Università degli Studi di Firenze

Dipartimento di Sistemi e Informatica
Dottorato di Ricerca in Informatica ed Applicazioni

Ph.D. Thesis in Computer Science

Specification and Analysis of Service-Oriented Applications

Francesco Tiezzi

Supervisor

Prof. Rosario Pugliese

Ph.D. Coordinator

Prof. Rocco De Nicola

April, 2009

Abstract

Service-oriented computing, an emerging paradigm for distributed computing based on the use of services, is calling for the development of tools and techniques to build safe and trustworthy systems, and to analyse their behaviour. Therefore, many researchers have proposed to use process calculi, a cornerstone of current foundational research on specification and analysis of concurrent, reactive, and distributed systems. In this thesis we illustrate this approach by focussing on COWS, a process calculus expressly designed for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We show that COWS can model all the phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution. We present the calculus and a number of methods and tools that have been devised to analyse COWS terms: a type system to check confidentiality properties, a bisimulation-based observational semantics to check interchangeability of services and conformance against service specifications, a temporal logic and a model checker to express and check functional properties of services. We also show COWS's expressiveness both for modelling imperative and orchestration constructs, and for encoding other process and orchestration languages. We illustrate our approach through many specific examples and two large case studies, from automotive and financial domains.

Acknowledgements

I would start by expressing my deep and sincere gratitude to my supervisor Rosario Pugliese for guiding and supporting me during my PhD studies. Thank you for the time you have spent on our discussions and for having always the right advice at the right time. I appreciated it more than I can say.

I'm indebt to my PhD coordinator Rocco De Nicola for having introduced me to the world of formal methods and for giving me the opportunity of working with the Concurrency and Mobility Group at the DSI of Università degli Studi di Firenze.

I would like to thank Mariangiola Dezani and Kohei Honda for accepting to read this thesis and for their precious comments and suggestions.

In the last three years, I had the privilege of working with great people who offered me not only their wide knowledge but also their trust and their support. A special thanks must go to Alessandro Lapadula. It was a pleasure to work with him for all these years; he is a good colleague as well as a nice friend. I have to say thanks to the people of ISTI-CNR of Pisa, Stefania Gnesi, Franco Mazzanti and Alessandro Fantechi, that have believed in our work and invested their time and resources in the development of a software tool for COWS. I'm also grateful to Laura Bocchi and José Luiz Fiadeiro, from the University of Leicester, and Federico Banti, from DSI, for their collaboration.

A particular thanks to Nobuko Yoshida from Imperial College of London, for her hospitality during my visit and for her precious contribution to my research work. She has also taught me to look the things from a different point of view. This thanks is also for the people belonging to her group, Martin, Raymond, Dimitris and Andi. I would say thanks also to Marco Carbone for our fruitful discussions and for his help during my stay in London.

I'm grateful to Università degli Studi di Firenze and to the EU project SENSORIA for their financial support.

I'm really grateful to all the people I met at DSI. Thanks to my roommates (in strict alphabetical order, Alessandro D., Alessandro L., Andrea, Antonio, Carlo, Carlotta, Daniele, Davide, Elena, Federico, Francesco B., Francesco C., Leonardo, Liliana, Lorenzo, Lucia, Maddalena, Massimiliano, Nicoletta, Paolo, Pierluigi, Sara, Stefano, Tania) for tolerating my 'noisy' presence and for not burning the big piles of papers on my desk (I will eventually read all of them!). Thanks also to Michele for his precious suggestions and great help, Simona for her kindness and ability to solve all my bureaucratic problems, and Betti, Piluc and Lorenzo for all advices, chats and coffee breaks.

I thank Mamo for his delicious ‘panino al lampredotto’ and our funny chats.

I would say thanks to my family for their (not only financial) support even if it is not completely clear to them what my job is. Last but not least, I thank my wife Sandra for her love and for keeping her promise to be there in good times and bad; this thesis would not have been at all possible without her support.

To my wife Sandra

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for *Service-Oriented Computing* (SOC) supporting automated use. This emerging paradigm finds its origin in object-oriented and component-based software development, and aims at enabling developers to build networks of interoperable and collaborative applications, regardless of the platform where the applications run and of the programming language used to develop them, through the use of independent computational units, called *services*. Services are loosely coupled reusable components, that are built with little or no knowledge about clients and other services involved in their operating environment. In the end, SOC systems deliver application functionalities as services to either end-user applications or other services.

There are by now some successful and well-developed instantiations of the general SOC paradigm, like e.g. Web Services and Grid Computing, that exploit the pervasiveness of Internet and related standards. However, current software engineering technologies for SOC remain at the descriptive level and lack rigorous formal foundations. In the design of SOC systems we are still experiencing a gap between practice (programming) and theory (formal methods and analysis techniques). The challenges come from the necessity of dealing at once with such issues as asynchronous interactions, concurrent activities, workflow coordination, business transactions, failures, resource usage, and security, in a setting where demands and guarantees can be very different for the many different components. Many researchers have hence put forward the idea of using *process calculi*, a cornerstone of current foundational research on specification and analysis of concurrent, reactive and distributed systems through mathematical — mainly algebraic and logical — tools. Indeed, due to their algebraic nature, process calculi convey in a distilled form the compositional programming style of SOC.

A major benefit of using process calculi, however, is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of SOC. In fact, it has been already argued that type systems, modal and temporal logics, and observational equivalences provide adequate tools to address topics

relevant to SOC (see e.g. [146, 195]). This 'proof technology' can eventually pave the way for the development of automatic property validation tools. Therefore, process calculi might play a central role in laying rigorous methodological foundations for specification and validation of SOC applications. Many process calculi for SOC have hence been proposed either by enriching well-established process calculi with specific constructs (e.g. the variants of π-calculus with transactions [32, 127, 128] and of CSP with compensation [52]) or by designing completely new formalisms (e.g. [131, 172, 50, 125, 104, 34, 35]).

This thesis focusses on one of these proposals, namely the process calculus COWS (*Calculus for Orchestration of Web Services*, [131]). The design of the calculus has been influenced by the principles underlying the OASIS standard for orchestration of web services WS-BPEL [166], and in fact COWS supports service instances with shared states, allows a process to play more than one partner role, permits programming stateful sessions by correlating different service interactions, and enables management of long-running transactions. COWS has also taken advantage of previous work on process calculi. Indeed, it combines in an original way constructs and features borrowed from well-known process calculi, e.g. non-binding input activities, asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

We put forward to use COWS as a linguistic formalism for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We present the calculus and a number of methods and tools that we have devised to analyse COWS terms: a type system to check confidentiality properties, a bisimulation-based observational semantics to check interchangeability of services and conformance against service specifications, a temporal logic and a model checker to express and check functional properties of services, and a symbolic characterisation of the operational semantics. We illustrate our approach and COWS's expressiveness through many specific examples and two large case studies, from automotive and financial domains.

1.1 About this thesis

In Chapter 2, we give an overview of SOC, by focussing on the standard language for orchestration of web services WS-BPEL [166]. Then, we point out major ambiguous features of the WS-BPEL specification by means of many examples, some of which are also exploited to test and compare the behaviour of three of the most known freely available WS-BPEL engines. We show that these ambiguities have led to engines implementing different semantics and, hence, complicate the task of developing WS-BPEL applications and undermine their portability across different platforms. To face these difficulties, we put forward using *formal methods* as a means to build up a framework to precisely describe the most significant aspects of a SOC application (e.g. a WS-BPEL program), to state and prove its properties, and to direct attention towards issues that might otherwise be overlooked. A formal approach also enables tailoring proof techniques and analytical

tools typical of process calculi to the needs of SOC applications. We briefly review some of these techniques and tools, and end the chapter with an informal presentation of the two case studies that will be used throughout the thesis for illustration purposes.

In Chapter 3, we introduce the formal language COWS used for specifying and analysing SOC applications. To gradually introduce the technicalities and distinctive features of COWS, we present its syntax and operational semantics in three steps. More specifically, firstly we consider the fragment of COWS without priority in parallel composition and linguistic constructs dealing with termination, then we enrich it with priority in parallel composition and, finally, we extend again the calculus by adding primitives for termination. For each of the three calculi we show many simple clarifying examples. After a discussion on the correspondence between WS-BPEL activities and COWS terms, we conclude with the specification in COWS of the two case studies informally described in the previous chapter.

In Chapter 4, we present some methods and tools to analyse COWS terms. Firstly, we introduce a type system for checking confidentiality properties, that uses types to express and enforce policies for regulating the exchange of data among services. Secondly, we describe a logical verification methodology for checking functional properties of SOC systems. The properties are described by means of SocL, a logic specifically designed to express in a convenient way peculiar aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses. Service behaviours, of course, are specified using COWS. The verification of SocL formulae over COWS specifications relies on an abstraction phase and is assisted by the on-the-fly model checker CMC. Thirdly, we study a bisimulation-based observational semantics for COWS, that is directly usable to check interchangeability of services and conformance against service specifications. We define natural notions of strong and weak open barbed bisimilarities and prove their coincidence with more manageable characterisations in terms of labelled bisimilarities. Finally, we define a symbolic characterisation of the operational semantics of COWS that avoids infinite representations of COWS terms due to the value-passing nature of communication in COWS and is more amenable for automatic manipulation by analytical tools, such as e.g. model and equivalence checkers.

In Chapter 5, we demonstrate COWS's expressiveness by showing some encodings of other formal languages for SOC: the three orchestration languages Orc, SCC and ws-calculus, the process calculus Lπ, and the modelling language SRML. Moreover, we present two COWS variants that permit modelling timed activities and dynamic service publication, discovery and negotiation. This way, the obtained linguistic formalism is capable of modelling all the phases of the life cycle of SOC applications.

In Chapter 6, we conclude with some final remarks and touch upon directions for future work.

Publications

This thesis is mainly the result of a collaboration with Dott. Alessandro Lapadula and Prof. Rosario Pugliese of Dipartimento di Sistemi e Informatica at Università degli Studi di Firenze. Besides them, other people that have contributed with their work to this thesis are Federico Banti, Laura Bocchi, Alessandro Fantechi, José Luiz Fiadeiro, Stefania Gnesi, Franco Mazzanti, and Nobuko Yoshida. Thus, part of the chapters has appeared in joint papers with them. In particular:

- Chapter 2 is based on [135, 137, 11];
- Chapter 3 is based on [131, 134, 174, 137, 11];
- Chapter 4 is based on [132, 174, 83, 175];
- Chapter 5 is based on [131, 134, 130, 31, 133, 136].

Some of the above papers have provided basis for the PhD thesis of Dott. Alessandro Lapadula that, besides COWS, presents other different approaches to model languages for web services orchestration: B*lite*, a lightweight language designed around some of WS-BPEL specific features, and ws-calculus, a calculus equipped with a typing discipline aiming at formalizing the relationship existing between WS-BPEL processes and the associated WSDL documents. This thesis, instead, focusses on COWS as specification language for SOC and introduces more techniques for analysing COWS terms.

Some of the work presented in this thesis has also been exploited by other researchers in [172, 173, 16, 205, 202].

Chapter 2

Background and motivations

In this introductory chapter, we set the scene of the whole thesis, by providing background notions from SOC and formal methods theory, and by outlining the main motivations behind this thesis.

After a general overview of the SOC paradigm, we introduce WS-BPEL, the standard language for orchestration of web services. We show many examples to point out major ambiguous features of the WS-BPEL specification and to test and compare the behaviour of three of the most known freely available WS-BPEL engines. Such ambiguities have led to engines implementing different semantics and, hence, complicate the task of developing WS-BPEL applications and undermine their portability across different platforms.

We then present an approach based on formal methods to face the above difficulties, which permits to build up a framework to precisely describe the most significant aspects of a SOC application (e.g. a WS-BPEL program), to state and prove its properties, and to direct attention towards issues that might otherwise be overlooked. A benefit of this approach is that it enables tailoring proof techniques and analytical tools typical of process calculi to the needs of SOC applications. Therefore, we outline some of the relevant tools for analysing process calculi terms, and give a glimpse of π-calculus, a cornerstone of current foundational research on specification and analysis of concurrent, reactive, and distributed systems.

We end the chapter with an informal presentation of the two case studies that will be used throughout the thesis for illustration purposes.

Structure of the chapter. The rest of the chapter is organized as follows. Section 2.1 presents basic elements of service-oriented architecture and computing. Section 2.2 provides an overview of WS-BPEL and shows many peculiar examples and the results of our experimentation with some WS-BPEL engines. Section 2.3 introduces formal methods as a mean to specify and analyse SOC application, with special concern on process calculi and the related theory. Section 2.4 presents two case studies, from automotive and financial domains.

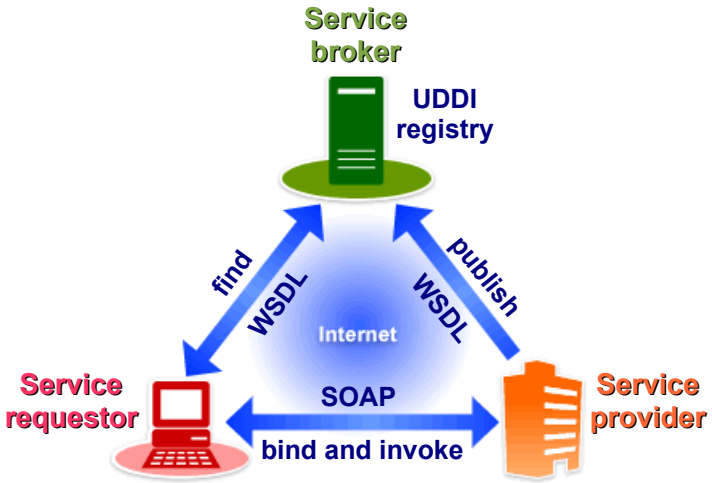

Figure 2.1: Service-Oriented Architecture

2.1 Service-Oriented Computing

Service-oriented computing (SOC) is emerging as an evolutionary paradigm for distributed and e-business computing that finds its origin in object-oriented and component-based software development. Early examples of technologies that are at least partly service-oriented are CORBA, DCOM, J2EE or .NET. Also, early adopters of the SOC approach have created their own service-oriented enterprise architectures based on messaging systems, such as IBM WebSphere.

A more recent successful instantiation of the SOC paradigm are *web services*. These are sets of operations (i.e. functionalities) that can be published, located and invoked through the Web via XML messages complying with given standard formats. To support the web service approach, several new languages and technologies have been designed and many international companies have invested a lot of efforts.

There is a common way to view the web service architecture. It focuses on three major roles:

- *Service provider*: The software entity that implements a service specification and makes it available on the Internet. Providers publish machine-readable service descriptions on registries to enable automated discovery and invocation.

- *Service requestor* (or *client*): The software entity that invokes a service provider. A service requestor can be an end-user application or another service.

- *Service broker*: A specific kind of service provider that allows automated publication and discovery of services by relying on a registry.

Figure 2.1 shows the three service roles and how they interact with each other. This

architecture, and the context of services use, imposes a series of constraints. Here are some key characteristics for effective use of services (see, e.g., [43]):

- *Coarse-grained*: Operations on services are frequently implemented to encompass more functionalities and operate on larger data sets, compared to those of fine-grained components as well as object-oriented interfaces.
- *Interface-based design*: Services implement separately defined interfaces. The benefit of this is that multiple services can implement a common interface and a service can implement multiple interfaces. The set of interfaces implemented by a service is called *service description*. In addition to the functions that the service performs, service descriptions should also include non-functional properties, such as e.g., response time, availability, reliability, security, and performance, that jointly represent the quality of the service. In this case, they are also called *service contracts*.
- *Discoverable*: Services need to be found at both design time and run time by service requestors.
- *Loosely coupled*: Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges.
- *Asynchronous*: In general, services use an asynchronous message passing approach. However, this is not required; in fact, some services may use synchronous message passing too.

Some of these criteria, such as interface-based design and discoverability, are also used in component-based development; however, it is the sum of these attributes that differentiates a service-based application from a component-based one. It is beneficial, for example, to make web services asynchronous to reduce the time a requestor spends waiting for responses. By making a service call asynchronous, with a separate return message, the requestor will be able to continue execution while the provider has a chance to respond. This is not to say that synchronous service behavior is wrong, just that experience has demonstrated that asynchronous service behavior is desirable, especially where communication costs are high or network latency is unpredictable, and provides the developer with a simpler scalability model [43].

To support the web service approach, many new languages, most of which based on XML, have been designed. The technologies that form the foundations of web services are SOAP, WSDL, and UDDI. Simple Object Access Protocol (SOAP, [38]) is responsible for encoding messages in a common XML format so that they can be understood at either end by all communicating services. Currently, SOAP is the principal XML-based standard for exchanging information between applications within a distributed environment. Web Service Description Language (WSDL, [66]) is responsible for describing the public interface of a specific web service. Through a WSDL description, that is an XML document,

a client application can determine the location of the remote web service, the functions it implements, as well as how to access and use each function. After parsing a WSDL description, a client application can appropriately format a SOAP request and dispatch it to the location of the web service. In this setting, Universal Description, Discovery, and Integration (UDDI [194]) is responsible for centralizing services into a common registry and providing easy *publish* and *find* functionalities. The relationships between SOAP, WSDL, and UDDI are depicted in Figure 2.1.

To move beyond the basic framework *describe-publish-interact* and to better appreciate the real value of web services, mechanisms for service composition and quality of service protocols are required. Several specifications have been proposed in these areas, most notably service composition languages such as Web Services Business Process Execution Language (WS-BPEL, [166]) and Web Services Choreography Description Language (WS-CDL, [118]), and standards for other important aspects of services such as WS-Transaction [165], WS-Security [164], and WS-Reliable Messaging [163].

In the web services literature [170], terms *orchestration* and *choreography* are both used to describe composition of web services. Orchestration describes how web services can interact with each other at the message level, including the business logic and the execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional, multi-step process model. Choreography tracks the sequence of messages that may involve multiple parties. For orchestration, the process is always controlled from the perspective of one of the business parties. Choreography is more collaborative in nature: it is defined according to a global perspective, where each party involved in the process describes the part that plays in the choreography.

A service orchestration combines services following a certain composition pattern to achieve a business goal or provide new service functions in general. For example, handling a purchase order is the summation of processes that calculate the final price for the order, select a shipper, and schedule the production and shipment for the order. It is worth emphasizing that service orchestrations may themselves become services, making composition a recursive operation. In the example above, handling a purchase order may become a service that is instantiated to serve each received purchase order separately from other similar requests. This is necessary because a client might be carrying on many simultaneous purchase order interactions with the same service.

Service descriptions are thus used as templates for creating service instances that deliver application functionality to either end-user applications or other instances. The technology supporting tightly coupled communication frameworks typically establishes an active connection between interacting entities that persists for the duration of a given business activity (or even longer). Because the connection remains active, context is inherently present, and correlation between individual transmissions of data is intrinsically managed by the technology protocol itself. Instead, the loosely coupled nature of SOC implies that a same service should be identifiable by means of different logic names and the connection between communicating instances cannot be assumed to persist for the du-

ration of a whole business activity. Therefore, there is no intrinsic mechanism for associating messages exchanged under a common context or as part of a common activity. Even the execution of a simple request-response message exchange pattern provides no built-in means of automatically associating the response message with the original request. It is up to each single message to provide a form of context thus enabling services to associate the message with others. This is achieved by embedding values in the message which, once located, can be used to correlate the message with others logically forming a same stateful interaction 'session'. A key observation is that *message correlation* is an essential part of messaging within SOC as it enables the persistence of activities' context and state across multiple message exchanges while preserving service statelessness and autonomy, and the loosely coupled nature of service-oriented systems.

A further key feature of languages for service composition is the recovery mechanism for long-running business transactions. In SOC environments, the ordinary assumptions about primitive operations in traditional databases (Atomicity, Consistency, Isolation and Durability, ACID) are not applicable in general because local locks and isolation cannot be maintained for the long periods (see [166], Section 12.3). Therefore, many languages for service composition rely on the concept of *compensation*, i.e. activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned.

In the next section we will focus our attention on WS-BPEL, an OASIS standard language for web service orchestration.

2.2 Overview of WS-BPEL and experimentation

We provide an overview of WS-BPEL and present some illustrative examples of WS-BPEL programs used to test and compare the behaviour of three of the most known freely available WS-BPEL engines. The section ends with an evaluation of the results of our experimentation.

2.2.1 A glimpse of WS-BPEL

WS-BPEL is essentially a linguistic layer on top of WSDL for describing the structural aspects of web service orchestration. In practice, and briefly, WSDL is a W3C standard that permits to express the functionalities offered and required by web services by defining, akin object interfaces in Object-Oriented Programming, the signatures of operations and the structure of messages for invoking them and returned by them. The WSDL document associated with a WS-BPEL program can then be exploited to verify the possibility of connecting different services.

In WS-BPEL, the logic of interaction between a service and its environment is described in terms of structured patterns of communication actions composed by means of control flow constructs that enable the representation of complex structures. Orchestra-

tion exploits state information that is maintained through shared variables and managed through message correlation. For the specification of orchestration, WS-BPEL provides many different activities that are distinguished between *basic activities* and *structured activities*.

The following basic activities are provided: <receive> and <reply>, to enable web service one-way and request-response operations; <invoke>, to invoke web service operations; <wait>, to delay execution for some amount of time; <assign>, to update the values of variables with new data; <throw>, to signal internal faults; <exit>, to immediately end the service instance; <empty>, to do nothing; <compensate> and <compensateScope>, to invoke compensation handlers; <rethrow>, to propagate faults; <validate>, to validate variables; and <extensionActivity>, to add new activity types.

The structured activities describe the control flow logic of a business process by composing basic and/or structured activities recursively. The following structured activities are provided: <sequence>, to process activities sequentially; <if>, to process activities conditionally; <while> and <repeatUntil>, to repetitively execute activities; <flow>, to process activities in parallel; <pick>, to perform activities selectively; <forEach>, to (sequentially or in parallel) perform multiple activities; and <scope>, to associate handlers for exceptional events to a primary activity.

Notably, synchronization dependencies among activities, other than by means of control flow constructs, can also be specified through *flow links* to form directed acyclic graphs. A flow link is a conditional transition that connects a 'source' activity to a 'target' activity. When a source activity completes, the associated *transition condition* is evaluated to determine the status of the *join condition* that acts on the flow link of the target activity. A target activity may only start when all its source activities complete and its join condition evaluates to true.

The handlers within a <scope> can be of four different kinds: <faultHandler>, to provide the activities in response to faults occurring during execution of the primary activity; <compensationHandler>, to provide the activities to compensate the successfully executed primary activity; <terminationHandler>, to control the forced termination of the primary activity; and <eventHandler>, to process message or timeout events occurring during execution of the primary activity. If a fault occurs during execution of a primary activity, the control is transferred to the corresponding fault handler and all currently running activities inside the scope are interrupted immediately without involving any fault/compensation handling behaviour. If another fault occurs during a fault/compensation handling, then it is re-thrown, possibly, to the immediately enclosing scope. Compensation handlers attempt to reverse the effects of previously successfully completed primary activities (scopes) and have been introduced to support Long-Running (Business) Transactions (LRTs). Compensation can only be invoked from within fault or compensation handlers starting the compensation either of a specific inner (completed) scope, or of all inner completed scopes in the reverse order of completion. Invoking a

compensation handler that is unavailable is equivalent to perform an empty activity.

A WS-BPEL program, also called (*business*) *process*, is a <process>, that is a sort of <scope> without compensation and termination handlers.

WS-BPEL uses the basic notion of *partner link* to directly model peer-to-peer relationships between services. This relationship is expressed at the WSDL level by specifying the roles played by each of the services in the interaction. However, the information provided by partner links is not enough to deliver messages to a business process. Indeed, since multiple instances of a same service can be simultaneously active because service operations can be independently invoked by several clients, messages need to be delivered not only to the correct partner, but also to the correct instance of the service that the partner provides. To achieve this, WS-BPEL relies on the business data exchanged rather than on specific mechanisms, such as WS-Addressing [103] or low-level methods based on SOAP headers. In fact, WS-BPEL exploits *correlation sets*, namely sets of *correlation variables* (called *properties* in WS-BPEL jargon), to declare the parts of a message that can be used to identify an instance. This way, a message can be delivered to the correct instance on the basis of the values associated to the correlation variables, independently of any routing mechanism.

2.2.2 An assessment of three WS-BPEL engines

We now present some illustrative examples of WS-BPEL programs and use them to test and compare the behaviour of three of the most known freely available WS-BPEL engines, namely ActiveBPEL [4], Apache ODE [9] and Oracle BPEL Process Manager [167] (the former two are open source projects, whereas the latter is distributed under the Oracle Technology Network Developer License)[1]. For our evaluation, we have taken into account fundamental features of WS-BPEL that remained unchanged since its initial version.

For the sake of readability, in this section WS-BPEL programs are presented by exploiting the graphical notations introduced in Figure 2.2, rather than the usual verbose textual form. We additionally use the following symbols:

- ⓘ to label an activity that initializes correlated variables;
- ⓤ to label a receive activity that does not use correlated variables;
- ⓒ to label an activity that checks correlated variables;
- ⓘⓒ to label an activity that initializes or checks correlated variables;
- ☎ to label an activity waiting for a message from a partner;
- ✔ to label a completed activity;

[1] ActiveBPEL and Oracle BPEL Process Manager are also part of (commercial) tool suites, namely ActiveVOS and Oracle SOA Suite, for designing, developing, testing, deploying and maintaining WS-BPEL applications.

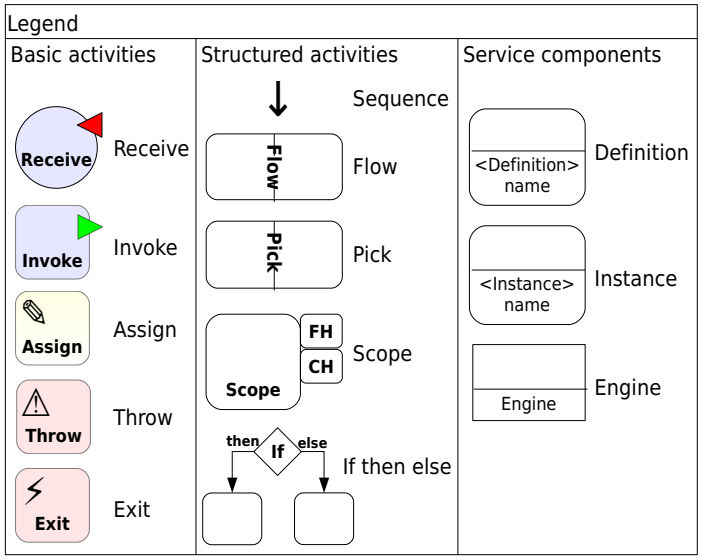

Figure 2.2: Basic/structured activities and service components

- ❇ to label a completed start activity that initiates a new instance of the service;
- ✘ to label a terminated activity due to the execution of `<exit>` or `<throw>` activities.

Example 2.2.1 (Message correlation) A client can request a log-on operation via LogOn, and can request some logging information via RequestLogInfo; this information can be asynchronously obtained by implementing the callback operation SendLogInfo (on the use of asynchronous request-response patterns in service-oriented applications see also Example 2.2.2). Correlation variables can be exploited to correlate, by means of their same contents, different service interactions logically forming a same 'session'. For example, consider the simple service LogOnService in Figure 2.3(a) providing 'log-on' and 'request-log-info' operations. Initially, to request a log-on a client must send its logID with some other data. Then, the service waits for a request from the client to provide some logging information[2]. After that, the service can reply (and terminate) by sending the requested information to the client. Notably, the WS-BPEL process in Figure 2.3(a) cannot ensure that the service does provide logging information properly. In fact, since the messages for operations LogOn and RequestLogInfo are uncorrelated, if concurrent instances are running then, e.g., successive invocations for the same instance can be mixed up and delivered to a wrong instance. This behavior can be prevented by simply correlating consecutive messages by means of some correlation data, e.g. logID, as in the modified service LogOnService of Figure 2.3(b).

[2]For the sake of simplicity, we assume here that the logging information are simply the data sent by the client through invocation of operation LogOn. In a more realistic scenario, of course, logging information could be internally computed by LogOnService or retrieved from a (possibly external) service.

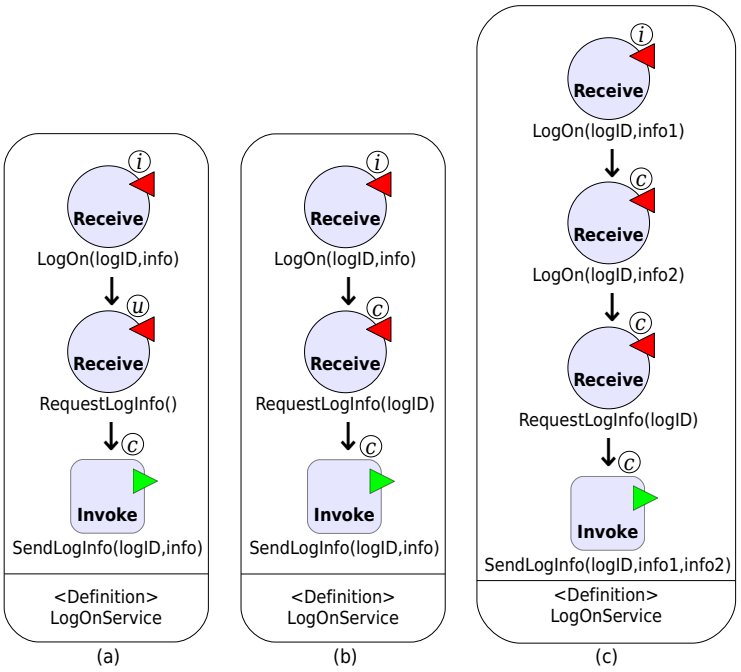

Figure 2.3: Message correlation

A special case is when the two initial receives are on the same partner and operation, as in Figure 2.3(c) where LogOnService requires some extra-information from the client, so that it waits for two consecutive log-on requests to let the client logging on the service. This is allowed by the WS-BPEL specification [166, Section 10.4] that, however, does not mention that possible conflicting receives could arise. This situation is illustrated in Figure 2.4(a), where it is assumed that a client process has performed two log-on requests with data info1 and info2 that, accordingly to the intended semantics of WS-BPEL, should trigger only one instantiation of the service. This is indeed the behaviour of ActiveBPEL and Apache ODE, that exploit the received data to correlate the two consecutive receives and, thus, to prevent creation of a wrong new instance. On the contrary, when executing this example, Oracle BPEL creates two instances, one for each received request as shown in Figure 2.4(b). An important consequence, and indeed an unexpected side effect, is that the created instances are in conflict and, then, will soon get stuck.

Example 2.2.2 (Asynchronous message delivering) In service-oriented systems communication paradigms are usually asynchronous (mainly for scalability reasons [43]), in the sense that there may be an arbitrary delay between the sending and the receiving of a message, the ordering in which messages are received may differ from that in which they were sent, and a sender cannot determine if and when a sent message will be received. We can guess from [166, Section 10.4], that this is also the case of WS-BPEL. To illustrate, consider the WS-BPEL process in Figure 2.5(a) representing a client logging on the previous service depicted in Figure 2.3(b). After the request for some user information

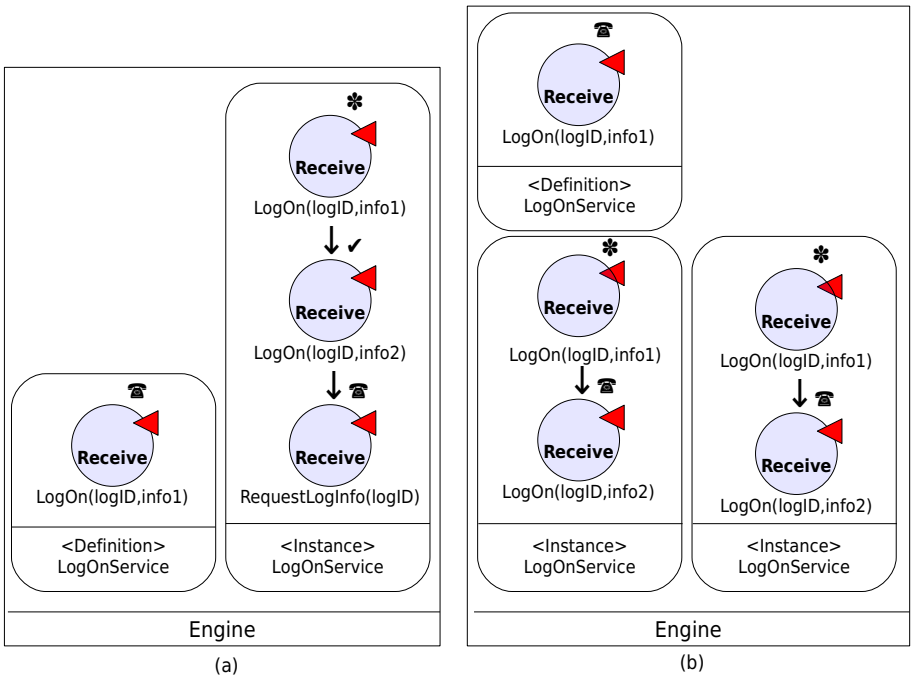

Figure 2.4: Message correlation: service instantiation

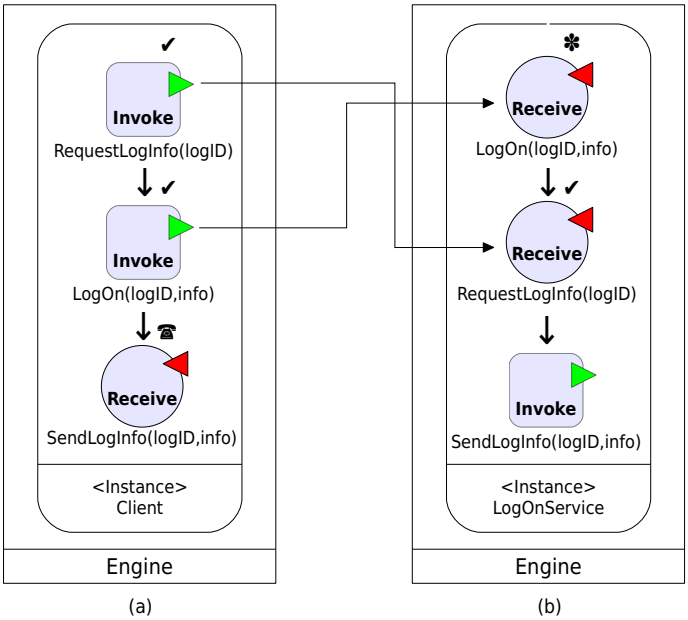

Figure 2.5: Asynchronous message delivering

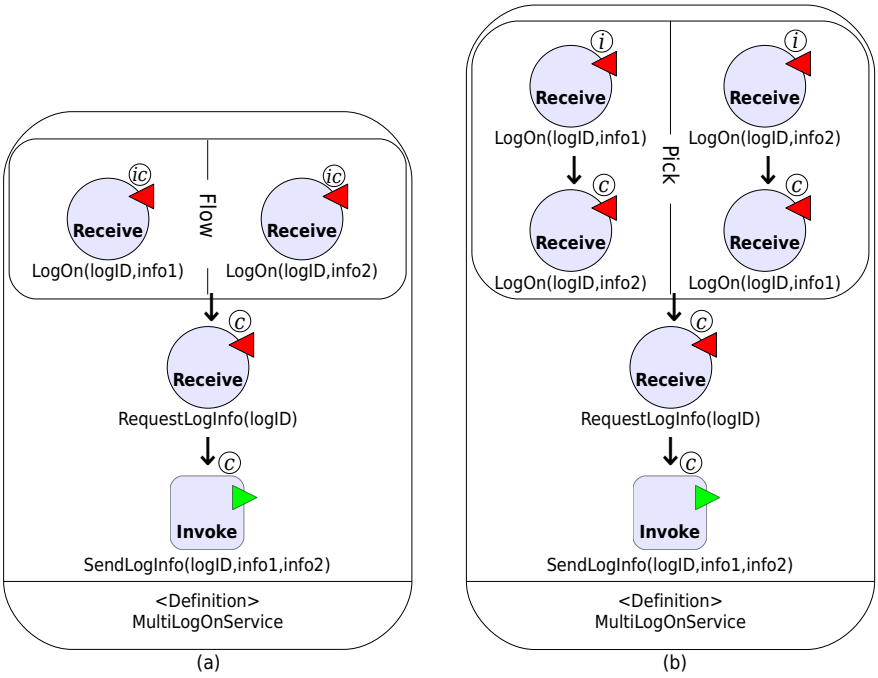

Figure 2.6: Multiple start activities

is performed by the first invoke activity, a service instance is created as a result of consumption of the request for logging on the service produced by the second invoke activity as depicted in Figure 2.5(b). Now, the first produced message is not considered expired and, thus, can be consumed by the newly created service instance. All the examined WS-BPEL engines *tacitly* agree with this communication paradigm, although no requirement is explicitly reported in the WS-BPEL specification.

Example 2.2.3 (Multiple start and conflicting receive activities) When defining services, the WS-BPEL specification allows for using multiple start activities [166, Section 10.4]. However, it is not clear how conflicting receive activities enabled at instantiation of such a service must be handled. To explain this point, consider a simple variant of service LogOnService, called MultiLogOnService, that allows two clients to log on the same service instance. Figure 2.6 illustrates two alternative definitions of MultiLogOnService with the same semantics: the one on the left hand side makes use of activity `<flow>`, while the one on the right hand side uses activity `<pick>`. In both definitions, the service waits for two log-on requests from clients and then, on demand by one of the two clients, provides logging information. After a message from a client, say client1, has been processed, an instance of the service is initiated as illustrated in Figure 2.7(a) (we only consider the case of the definition in Figure 2.6(a)). Now, the definition and the instance of the service compete for receiving the same message sent by another client that is correlated to that sent by client1 through the datum logID. In cases like this, the WS-BPEL specification requires the second message to be delivered to the existing instance, thus preventing creation of a

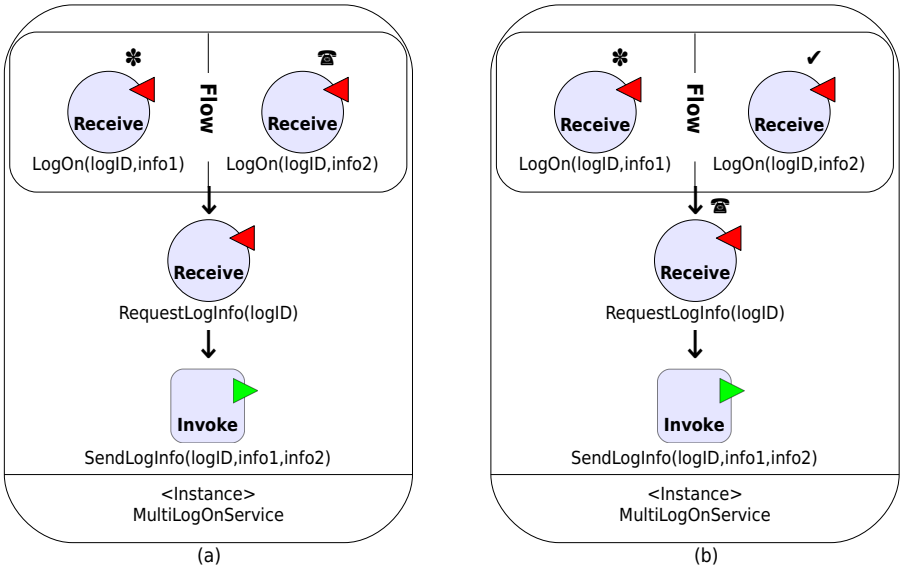

Figure 2.7: Multiple start activities: service instantiation

new instance. In fact, the instance in Figure 2.7(a) can only reduce to that of Figure 2.7(b).

In case of conflicting receives, the WS-BPEL specification document prescribes to raise the standard fault `bpel:conflictingReceive`, which seems to be somehow in contrast with what we have illustrated before. In fact, this situation readily occurs when a service exploits multiple start activities, because of race conditions on incoming messages among the service definition and the created instances. However, in such cases, it does not seem fair to raise a fault because the correlation data contained within each incoming message should be sufficient to decide if the message has to be delivered to a specific instance or to the service definition. This is indeed a tricky question that leads the three engines we have considered to behave differently. Indeed, Oracle BPEL always raises the fault `bpel:conflictingReceive`, ActiveBPEL exploits correlation to enforce creation of only one service instance (just like the example in Figure 2.7), whereas Apache ODE does not currently support multiple start activities.

Example 2.2.4 (Scheduling of parallel activities) While using the WS-BPEL engines, we have also experimented that they implement the flow activity in a different manner. For example, the expected behaviour of the WS-BPEL process in Figure 2.8(a) is that the three assignments are executed in an unpredictable order that may change in different executions. In fact, only Apache ODE implements this semantics, while the other two engines execute the assignments in an order fixed in advance, that is sequentially from left to right in case of ActiveBPEL (Figure 2.8(b)) and from right to left in case of Oracle BPEL (Figure 2.8(c)).

Example 2.2.5 (Forced termination) The WS-BPEL specification [166, Section 12.6] states: "The <`sequence`> and <`flow`> constructs *must* be terminated by terminating their

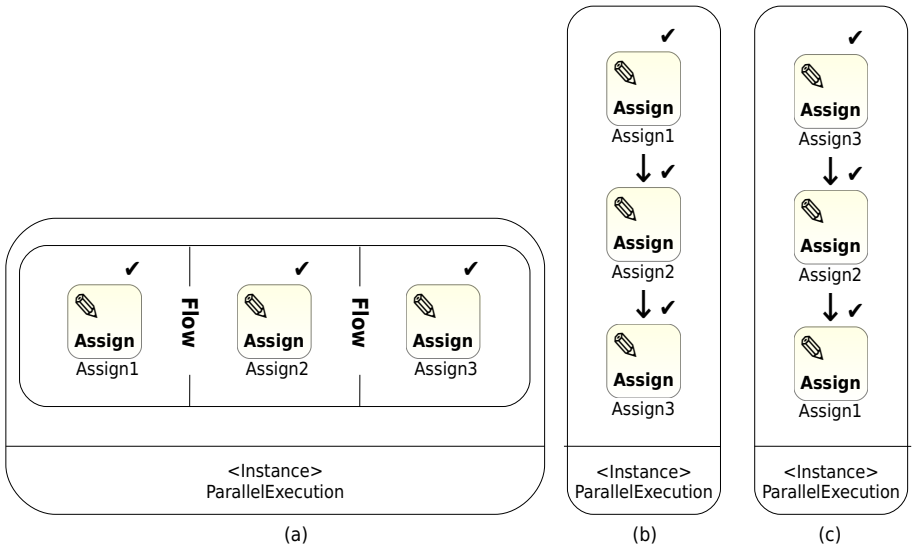

Figure 2.8: Scheduling of parallel activities

behavior and applying termination to all nested activities currently active within them". This sentence is ambiguous because it is not clear what "nested activities currently active" means in case of termination due to <exit> or <throw> activities. For example, consider a sequence of two assign activities. In Oracle BPEL, termination prompted by a parallel <exit> activity has no effect on the sequence (Figure 2.9(a)), while termination prompted by a parallel <throw> activity causes execution of only the first assign activity (Figure 2.9(b)). ActiveBPEL is more compliant to WS-BPEL for which all currently running activities must be terminated as soon as possible (Figure 2.9(c)) without any fault handling or compensation [166, Section 10.10]. However, differently from what the WS-BPEL specification seems to suggests, ActiveBPEL does not distinguish *short-lived activities* (i.e. sufficiently brief activities that may be allowed to complete) from basic activities and makes them terminate in the same way. Finally, Apache ODE is fully compliant with WS-BPEL, since a termination activity function is applied to the continuation that only retains short-lived activities.

Example 2.2.6 (Eager execution of activities causing termination) As shown in Example 2.2.5, to be compliant with the WS-BPEL requirement stating that termination activities must end immediately all currently running activities [166, Section 10.10], when defining the semantics of WS-BPEL, execution of activities <throw> and <exit> must have higher priority than execution of the remaining ones. Thus, for example, consider again a sequence of two assign activities. By executing a parallel <throw> activity, since assigns are not considered short-lived activities, the whole process can only reduce as shown in Figure 2.10(a). While ActiveBPEL agrees with this requirement, Oracle BPEL and Apache ODE do not implement any prioritized behavior for activities forcing termination. Thus, the above process can also evolve by firstly performing the first <assign>

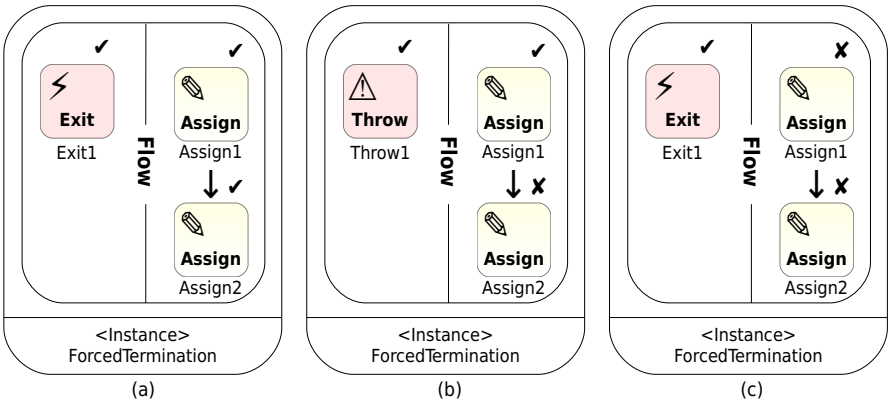

Figure 2.9: Forced termination

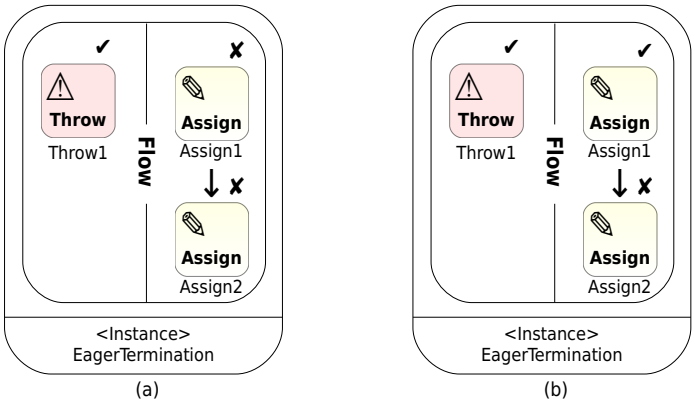

Figure 2.10: Eager execution of activities causing termination

activity and then the activity <throw>, as shown in Figure 2.10(b); this way, the first assign activity is not forced to terminate.

Example 2.2.7 (Handlers protection) The structured activity in Figure 2.11 consists of a process with two inner parallel activities, one of which being a scope whose primary activity is a sequence of a scope and a <throw> activity (Throw1), while the other parallel activity is a basic <throw> activity (Throw2). Suppose that the innermost scope performs its assignment Assign1 and completes. Then, the associated compensation handler CH (i.e. the activity Assign2) is installed. When execution of Throw1 rises a fault, then it is caught by the corresponding fault handler (that here is the default one and, hence, is not depicted in Figure 2.11) that simply activates the compensation consisting of execution of Assign2. This activity can be effectively executed since it is appropriately protected from the effect of execution of the parallel activity Throw2 (which has been enabled by completion of the scope enclosing Assign1).

We end by remarking two aspects of the compensation mechanism prescribed by the WS-BPEL specification [166, Sections 12.5 and 10.10]. First, compensation handlers

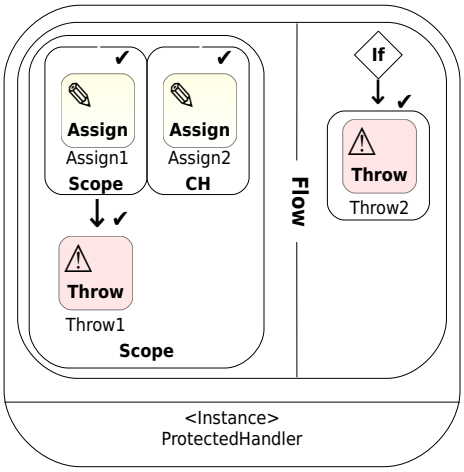

Figure 2.11: Handlers protection

of faultily terminated scopes should not be installed. Second, fault and compensation handlers should not be affected by the activities causing termination. Both aspects are not faithfully implemented in Oracle BPEL, while ActiveBPEL and Apache ODE meet these specific requirements and adhere to the intended WS-BPEL semantics.

2.2.3 Evaluation

The results of our experiments, summarized in Table 2.1, point out that the engines we have experimented with are not fully compliant with the intended semantics of WS-BPEL. This is also a consequence of the lack of a formal semantics for WS-BPEL, that would have disambiguated the intricate and complex features of the language. Therefore the work carried out in this thesis, and works with similar goals, other than as a guide for developing compliant implementations since the early stages, can be also used for making future versions of existing implementations more compatible.

We end our evaluation with some observations on the procedure to deploy WS-BPEL programs, although the description of the deployment is out of scope of the WS-BPEL specification document [166]. A WS-BPEL process is designed to be a reusable definition that can be deployed in different ways within different scenarios. In these respects the three tested engines pose different requirements. ActiveBPEL provides deployment information (i.e. partner link bindings and address information) in terms of abstract WS-BPEL elements (i.e. partner links and partner roles), while Apache ODE and Oracle BPEL Process Manager use proprietary defined elements to describe a deployment, regardless of whether the same elements are declared at WS-BPEL level. The integration of different deployment documents is then impossible to obtain, which is another factor that reduces the level of portability a programmer might expect.

	Oracle BPEL	ActiveBPEL	Apache ODE
Message correlation (Ex. 2.2.1)	+	+	+
Consecutive conflicting receives (Ex. 2.2.1)	−	+	+
Asynchronous message delivering (Ex. 2.2.2)	+	+	+
Multiple start (Ex. 2.2.3)	−	+	−
Scheduling of parallel activities (Ex. 2.2.4)	−	−	+
Short-lived activities (Ex. 2.2.5)	+	−	+
Forced Termination (Ex. 2.2.5)	−	+	+
Eager execution (Ex. 2.2.6)	−	+	−
Handlers protection and installation (Ex. 2.2.7)	−	+	+

Table 2.1: WS-BPEL compliance of the tested engines

2.3 Formal methods for Service-Oriented Computing

We have seen in the previous section that current software engineering technologies for development and composition of services, like WS-BPEL, remain at the descriptive level and do not integrate such techniques as, e.g., those developed for component-based software development. Formal reasoning mechanisms and analytical tools are still lacking for checking that the web services resulting from a composition meet desirable correctness properties and do not manifest unexpected behaviors.

To this aim, in the last few years, many researchers have exploited the studies on *process calculi* as a starting point to define a clean semantic model and lay rigorous methodological foundations for service-based applications and their composition. Process calculi, being defined algebraically, are inherently compositional and, therefore, convey in a distilled form the paradigm at the heart of SOC. This trend is witnessed by the many process calculi-like formalisms for orchestration and choreography. Most of these formalisms, however, do not suit for the analysis of currently available SOC technologies in their completeness because they only consider a few specific features separately, possibly by embedding *ad hoc* constructs within some well-established process calculi (see, e.g., the variants of π-calculus with transactions [32, 127, 128] and of CSP with compensation [52]). Besides these works, among the several proposals for orchestration and choreography languages based on formal frameworks inspired to process calculi, we want to mention [131, 47, 172, 50, 125, 104, 34, 57, 35]. Rather than as ordinary specification or programming languages, process calculi should be seen as experimental prototypes, whose primitives can inspire new computing paradigms.

A major benefit of using process calculi is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of SOC applications. For example, it has been shown that type systems, model checking and (bi)simulation analysis provide adequate tools to address topics relevant to the web services technology (see e.g. [146, 195]). In the end, this 'proof technology' can pave the way for the development of automatic property validation tools.

In the rest of the section, we outline some of the most relevant tools for analysing process calculi terms, and conclude with a glimpse of π-calculus, a cornerstone of current

foundational research on specification and analysis of concurrent, reactive, and distributed systems.

Type systems. Type systems have been shown to provide an important ingredient to statically detect (and prevent) execution errors of programs written in modern programming languages. Many of the SOC features (e.g. loose coupling, distributed and heterogeneous components) raise the need for the introduction of flexible and compositional type systems, capable of specifying the compositional behaviour of components in service-oriented applications and ensuring that the resulting compositions behave as intended across service composition in an open-ended way. While the sequential behaviour of services may be successfully approached using more traditional notions of types, the requirement of *compositionality* and *service awareness* raises additional challenges and seem to require fundamental changes in the underlying conceptual framework of type systems. However, to address topics relevant to the web services technology, type systems for process calculi might be a practical and scalable way to provide evidence that a large number of applications enjoy some given properties. Technically, one can prove the *type soundness* of a language as a whole, from which it follows that all *well-typed* applications do comply with the properties stated by their types.

Among the proposals appeared in literature, we want to mention two orthogonal research directions for developing typing systems for service compositionality. The first one concerns the way behavioural types can support and enforce high level properties of the *global service interactions*. The second one tackles the problem of formally representing the internal conversation rules between services, i.e. *service sessions*. A static approach to ensure global properties of service interactions has been introduced in [13, 14, 15] for λ^{req}, an extension of λ-calculus with primitive constructs for call-by-contract invocation. In particular, an automatic machinery, based on a type system and a model-checking technique, constructs a viable plan for the execution of services belonging to a given orchestration. Non-functional aspects are also included and enforced by means of a runtime security monitor. Service sessions are explored in [112, 207, 57, 3, 124, 114, 65] in terms of *session types*, an emerging powerful tool for taking into account behavioural and non-functional properties of conversational interactions. Session types permit to express and enforce many relevant policies for, e.g., constraining the sequences of messages accepted by services, ensuring service interoperability and compositionality, and guaranteeing absence of deadlock in service composition.

Logics. As for type systems, modal and temporal logics have long been used to represent properties of concurrent and distributed systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc. (see e.g. [116, 107, 151, 54, 53]). These logics have proved suitable to reason about complex software systems because they only provide abstract specifications of these systems and can thus be used for describing system properties rather than system behaviours.

Logical verification frameworks can support means for checking functional properties of services by abstracting away from the computational contexts in which they are operating. As an example, services can be abstractly considered as entities capable of accepting requests, delivering corresponding responses and, on-demand, cancelling requests. Some interesting abstract properties can be *availability* (if a service is always capable to accept a request), *reliability* (if when a request is accepted by a service, a final successful response is guaranteed), *responsiveness* (if a service always guarantees a response to each received request), etc. Many other interesting properties can express desirable attributes of services and SOC applications (see, e.g., [6]).

An important advantage is that the application of temporal logics to the analysis of systems is often supported by software tools (see e.g. [67, 68, 110]).

Behavioural equivalences. Process calculi can be used to describe both implementations of services and specifications of their expected behaviours. An important ingredient of these languages is therefore a notion of behavioural equivalence between processes. Behavioural equivalences [148, 77, 182] can be used in several ways: for example, to prove the soundness of a protocol implemented in the language, to prove some form of correspondence between a service written in a language and its encoding in another language, to optimize the semantic model representing a given service, or to provide a means to establish formal correspondences between different views (abstraction levels) of a service, e.g., the contract it has to honour and its true implementation. Ideally, equivalences should be congruences, i.e. should be preserved by all the contexts of the language. This is often obtained by closing the equivalences w.r.t. all the possible language contexts, which makes direct equivalence proofs particularly difficult. A standard device to avoid such a quantification consists in defining a labelled operational model so that, when a system evolves, the action performed is made apparent. In this way, the interaction with an external context is recorded in the labels and the universal quantification over language contexts can be dropped: equivalence proving is made more feasible.

Developing equivalences for SOC is a non trivial task. Indeed, it requires handling unexpected behaviours and (typically) asynchronous communication paradigms which complicates the observational theory (see, e.g., [143]). The behavioural theories introduced in [50] provide a means to establish formal correspondences between different views (choreography and orchestration) of service compositions. A notion of compliance among services instead is formalized in [39, 40, 126], where a special case of equivalence, called *subcontract preorder*, characterizes the possibility to replace services with subservices without breaking the correctness of the composition.

π-calculus by examples. COWS and several several process calculi for SOC have drawn their inspiration from the π-calculus [150], therefore we present here its syntax and informal semantics by examples. In fact, the design of many orchestration languages, such as e.g. XLANG [193], Microsoft BizTalk Server [70] and WS-BPEL, is admittedly inspired

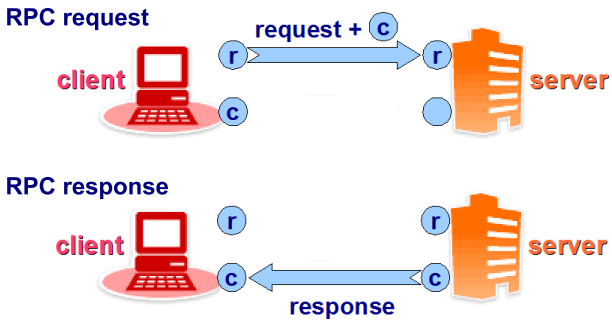

Figure 2.12: A remote procedure call interaction in π-calculus

to the programming metaphor offered by the π-calculus, based on message exchange in a distributed setting. Two books provide a complete account of π-calculus: *Communicating and Mobile Systems* by Milner [149] and *The π-calculus: A Theory of Mobile Processes* by Sangiorgi and Walker [182].

The basic computational step of π-calculus is the transmission of a communication channel between two processes; the receiver can then use the channel for further interactions with other parties. For example, Figure 2.12 depicts the process interfaces for an interaction based on the *Remote Procedure Call* (RPC) paradigm. An RPC interaction involves a client and a server, where after the first communication has taken place, the client waits for the server to elaborate a response. The communication channel at which the client waits for the server's response is c (abbreviation of 'callback') that, in the first communication along the channel r (abbreviation of 'request'), has been sent to the server.

The π-calculus term representing the client is $\bar{r}\langle d, c\rangle \,.\, c(x) \,.\, C$. The *output prefix* operator $\bar{r}\langle d, c\rangle \,.\, C'$ expresses that the request data d and the callback channel c are sent along the channel r and thereafter the process continues as C'. Similarly, the *input prefix* operator $c(x) \,.\, C$ means that a datum is received along the channel c and x is a placeholder for the received datum. After the input, the process will continue as C but with the received datum replacing x.

The server is rendered in π-calculus as $r(y, z) \,.\, \bar{z}\langle n\rangle$. It performs an input action $r(y, z)$ to receive an RPC request and replies by delivering the result of the call (i.e. the datum n) along the received channel (stored in z).

In the end, the whole system is

$$\bar{r}\langle d, c\rangle \,.\, c(x) \,.\, C \;\mid\; r(y, z) \,.\, \bar{z}\langle n\rangle$$

where the *parallel composition* operator $\_\mid\_$ allows the two processes to run concurrently and to interact with each other. Thus, the output and input actions along r can synchronise and communication can take place, leading to

$$c(x) \,.\, C \;\mid\; \bar{c}\langle n\rangle$$

where z is replaced by c in the server process. Now, by performing a second communication, this way along channel c, the system evolves to $C\{n/x\}$, that is the term obtained by

applying the substitution $\{n/x\}$ to the client continuation C.

A different kind of binding occurs in presence of the *restriction* operator: in $(\boldsymbol{\nu} c)\, P$ the name c is local to P. Coming back to the example above, suppose that initially c is a channel local to the client. The system now takes the form

$$(\boldsymbol{\nu} c)(\, \bar{r}\langle d, c\rangle \,.\, c(x) \,.\, C \,) \mid r(y, z) \,.\, \bar{z}\langle n\rangle$$

Thus, once a communication along r has taken place, c becomes a private channel shared between the two processes:

$$(\boldsymbol{\nu} c)(\, c(x) \,.\, C \mid \bar{c}\langle n\rangle \,)$$

To express infinite behaviours, a process may be *replicated*: a replicated term $!P$ represents an unbounded numbers of copies of P and it is (recursively) defined as $P \mid !P$. Re-consider our example

$$(\boldsymbol{\nu} c)(\, \bar{r}\langle d, c\rangle \,.\, c(x) \,.\, C \,) \mid !r(y, z) \,.\, \bar{z}\langle n\rangle$$

Here the server can receive an arbitrary number of procedure calls from clients. The above system can evolve as follows

$$(\boldsymbol{\nu} c)(\, c(x) \,.\, C \mid \bar{c}\langle n\rangle \,) \mid !r(y, z) \,.\, \bar{z}\langle n\rangle$$

Finally, another operator for defining π-calculus processes is the *choice*: $P + Q$ represents a process that can enact either P or Q. We see here an RPC client sending a second private communication channel e along which the server can deliver a message if some error occurs.

$$(\boldsymbol{\nu} c)(\boldsymbol{\nu} e)(\, \bar{r}\langle d, c, e\rangle \,.\, (c(x) \,.\, C \,+\, e(t) \,.\, E) \,) \mid !r(y, z, h) \,.\, (\tau \,.\, \bar{z}\langle n\rangle \,+\, \tau \,.\, \bar{h}\langle m\rangle)$$

In this case the RPC interaction produces the following system:

$$(\boldsymbol{\nu} c)(\boldsymbol{\nu} e)(\, (c(x) \,.\, C \,+\, e(t) \,.\, E) \mid (\tau \,.\, \bar{c}\langle n\rangle \,+\, \tau \,.\, \bar{e}\langle m\rangle) \,) \mid !r(y, z, h) \,.\, (\tau \,.\, \bar{z}\langle n\rangle \,+\, \tau \,.\, \bar{h}\langle m\rangle)$$

where the client can play two different behaviours depending on the response from the server, while the created server instance can proceed by performing one of the two *silent* actions τ, which can evolve without interaction with the environment. Suppose an error occurs (i.e. the τ on the right hand side of + is executed), then the system evolves to

$$(\boldsymbol{\nu} c)(\boldsymbol{\nu} e)(\, (c(x) \,.\, C \,+\, e(t) \,.\, E) \mid \bar{e}\langle m\rangle \,) \mid !r(y, z, h) \,.\, (\tau \,.\, \bar{z}\langle n\rangle \,+\, \tau \,.\, \bar{h}\langle m\rangle)$$

Now, the error can be communicated to the client by obtaining the following term

$$(\boldsymbol{\nu} c)(\boldsymbol{\nu} e)(\, E\{m/t\} \,) \mid !r(y, z, h) \,.\, (\tau \,.\, \bar{z}\langle n\rangle \,+\, \tau \,.\, \bar{h}\langle m\rangle)$$

Besides the standard π-calculus, roughly introduced above, in Chapter 3 we will refer also to some of its variants:

- asynchronous π-calculus (Aπ [111, 37, 7]): output actions cannot be used as prefixes and choice can only be guarded by receive activities; this way, communication is asynchronous in the sense that it is not possible for a sender to determine when its output is consumed by a synchronizing input;

- localised π-calculus (Lπ [147]): in each input prefix $a(b).P$ the name b may not occur free in P in input position; this way, only the output capability of names may be transmitted (see Remark 3.2.1 at page 43 for more details);

- π-calculus with polyadic synchronisation ($^e\pi$ [60]): channel names can be composite, i.e. they are vectors of names and interaction can take place only when such vectors match element-wise.

There is an ongoing debate, originating from the workflow community, about the relative merits of π-calculus, and more generally of process calculi, for modeling the service-oriented computing paradigm. For example, [179] has recently presented some challenges to model workflow in π-calculus. On the other hand, not trivial examples of applications of π-calculus as a formal foundation for modeling workflow have been investigated in [177, 176].

However, we are not completely convinced that all the features of π-calculus are well suited for expressing in a 'natural' way a number of workflow patterns like, e.g., the *cancelation patterns* [179]. In other words, exploiting directly π-calculus without introducing other orchestration primitives could be not-trivial and confusing, and could make it difficult to reason on the resulting modelled process. Thus, to fit with the specific requirements of the SOC paradigm, we should extend π-calculus with features borrowed from other process calculi such as, e.g., global scoping and non-binding input (from update calculus [168] and fusion calculus [169]), distinction between variables and values (from value-passing CCS [148], Applied π-calculus [1], Distributed π-calculus [108]), pattern-matching (from KLAIM [74]), prioritised activities (see, e.g., [69, 55, 171]), delimited forced termination (see [131]) and protection (inspired by [51]). In fact, COWS, the process calculus for SOC presented in details in the next chapter, results from a proper combination of all these features whose usefulness will be illustrated by means of several examples.

2.4 Case studies

In this section, we introduce two significant case studies defined within the EU project SENSORIA [185] that will be used throughout this thesis to illustrate our approach. The former [123] is a scenario in the area of automotive systems and describes some functionalities that will be likely available in the near future, while the latter [5] is from the financial domain.

2.4.1 An automotive case study

This case study involves a number of services that are discovered and bound at run-time according to levels of service specified at design time, so as to deliver the best available functionalities at agreed levels of quality.

We consider a scenario where vehicles are equipped with a multitude of sensors and actuators that provide the driver with services that assist in conducting the vehicle more safely. Driver assistance systems kick in automatically when the vehicle context renders it necessary. Due to the advances in mobile technology, automotive software installed in the vehicles can contact relevant specific services to deal with driver necessities.

Specifically, let us consider the case in which, while a driver is on the road with her/his car, the vehicle's *sensors monitor* reports a severe failure, which results in the car being no longer driveable. The car's *discovery* system then identifies garages, car rentals and towing truck services in the car's vicinity. At this point, the car's *reasoner* system chooses a set of adequate services taking into account personalised policies and preferences of the driver, e.g. balancing cost and delay, and tries to order them. Before being enable to order services, the owner of the car has to deposit a security payment, that will be given back if ordering the services fails. Other components of the in-vehicle service platform involved in this assistance activity are a *GPS* system, providing the car's current location, and an *orchestrator*, coordinating all the described services.

An UML-like activity diagram of the orchestration of services using UML4SOA, an UML Profile for service-oriented systems [141, 203, 142], is shown in Figure 2.13. The orchestrator is triggered by a signal from the sensors monitor (concerning, e.g., an engine failure) and consequently contacts the other components to locate and compose the various services to reach its goal. The process starts with a request from the orchestrator to the *bank* to charge the driver's credit card with the security deposit payment. This is modelled by the UML action CardCharge for charging the credit card whose number is provided as an output parameter of the action call. In parallel to the interaction with the bank, the orchestrator requests the current location of the car from the car's internal GPS system. The current location is modelled as an input to the RequestLocation action and subsequently used by the FindServices interaction which retrieves a list of services. If no service can be found, an action to compensate the credit card charge will be launched. For the selection of services, the orchestrator synchronises with the reasoner service to obtain the most appropriate services.

Service ordering is modelled by the UML actions OrderGarage, OrderTowTruck and RentCar. When the orchestrator makes an appointment with the garage, the diagnostic data are automatically transferred to the garage, which could then be able, e.g., to identify the spare parts needed to perform the repair. Then, the orchestrator makes an appointment with the towing service, providing the GPS data of the stranded vehicle and of the garage, to tow the vehicle to the garage. Concurrently, the orchestrator makes an appointment with the rental service, by indicating the location (i.e. the GPS coordinates either of the stranded vehicle or of the garage) where the car will be handed over to the driver.

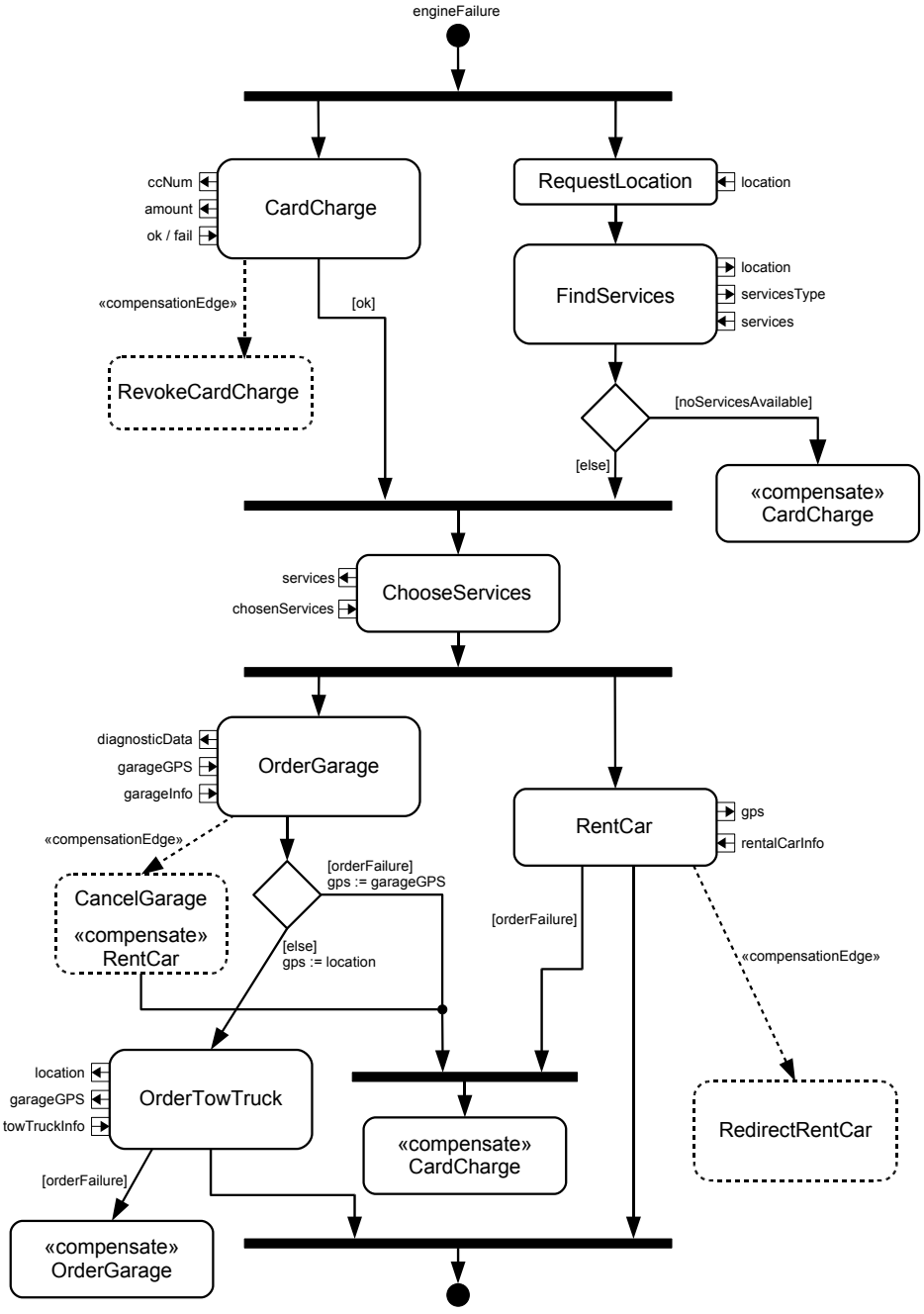

Figure 2.13: Orchestration in the automotive scenario

The workflow described in Figure 2.13 models the overall behaviour of the system. Besides interactions among services, it also includes activities using concepts developed for long running business transactions (in e.g. [98, 166]). These activities entail fault and compensation handling, kind of specific activities attempting to reverse the effects of previously committed activities, that are an important aspect of SOC applications. According to UML4SOA Profile, the installation of a compensation handler is modelled by an edge stereotyped ≪compensationEdge≫, and its activation by an activity stereotyped ≪compensate≫. Since each compensation handler is associated to a single UML activity, we omit drawing the enclosing scope construct. Moreover, we use dashed boxes to represent compensation handlers. Specifically, in the considered scenario:

- the security deposit payment charged to the driver's credit card must be revoked if either the discovery phase does not succeed or ordering the services fails, i.e. both garage/tow truck and car rental services reject the requests;
- if ordering a tow truck fails, the garage appointment has to be cancelled;
- if ordering a garage fails or a garage order cancellation is requested, the rental car delivery has to be redirected to the stranded car's actual location;
- instead, if ordering the car rental fails, it should not affect the tow truck and garage orders.

These requirements motivate the fact that ordering garage/tow truck and renting a car are modelled as activities running in parallel.

2.4.2 A finance case study

As in the previous section, we provide first an informal specification of the scenario, then a more detailed UML-based one.

The considered service is a credit (web) portal that provides the customer companies with the possibility to ask for a loan to a bank, and then orchestrates the necessary steps for processing the credit request, involving a preliminary evaluation by an employee, and subsequent evaluation by a supervisor before a contract proposal is sent to the customer.

Initially, the customer logins to the portal by providing his username and password, then he selects service Credit Request. In the next step, the customer uploads the necessary data for his request. More specifically, he firstly provides the desired credit amount, then the securities of the loan and his balance. The service checks the balance by resorting on a validation service and, in case the balance is not validated, it asks the user to provide it again.

When the request is completely filled by the customer, the service puts it in the list of tasks that the bank employees must accomplish. Then, an employee withdraws the request from the task list and fills his evaluation about it. The evaluation has a private part (only available for the bank purposes) and a public one which is available to the customer.

The private evaluation consists of the rating of the customer company and some additional information. The public evaluation consists of the decision about the request and, in case, the bank offer or the motivation for the rejection. The decision can be to reject the request, to accept it or to ask the customer for updating the request. According to the decision, the request processing may proceed in three different ways.

- If the request is rejected, the customer receives a message containing the response and its motivations, and then the process terminates.
- If an update is asked, a message is sent to the customer with the update request and its motivations. The customer may then decide to update the securities and/or the credit amount or refuse to update. In the latter case, the process terminates, while in the former one the updated request is processed again as from above.
- If the request is accepted, the service queues the contract (i.e. the request and the evaluation) in the list of tasks that the bank supervisors must accomplish. Then, a supervisor withdraws the contract from the task list and may update the public evaluation with its own decision. Again, the decision may be to ask for an update, to reject the request, or to accept it. The first two cases are processed as above, regarding the last one, the customer receives the offer and may answer positively or negatively. In both cases the process terminates. If the answer is positive, the process terminates positively and the contract is sent to an external service dealing with contracts for which customer and the bank have found an agreement.

At any moment the customer may require to abort the process. If this happens, the process terminates and, in case, the request is removed from task lists. As we will see later on, this last property requires execution of compensation activities to semantically rollback the action of queueing the request in the task list. This prevents an employee or a supervisor from examining an already aborted request.

We present now the UML specification of the scenario and its workflow. We rely again, for what regards the activity diagrams, on UML4SOA, a service-oriented profile of UML. However, due to the larger dimension of this scenario, to improve the understandability this time we follow all prescriptions of UML4SOA profile for the UML specification. In fact, within UML4SOA, the specific actions for service interaction are: send a message, receive a message and send&receive (a synchronous communication where a message is sent and then the service awaits for a reply). Actions have associated *pins*: the *link* pin specifies which is the partner of the interaction, the *input* and *output* pins specify the exchanged messages for send (only output pins), receive (only input pins), and send&receive (both input and output pins) actions. For instance, in Figure 2.14, a send&receive action is represented. The link pin, labelled by ≪link≫, specifies that service Authentication is the partner of the communication, the output pin, labelled by ≪snd≫, specifies that a message ID is sent and the input pin, labelled by ≪rcv≫, specifies that a message valid is received as an answer. The most important novelty in UML4SOA

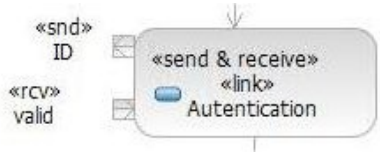

Figure 2.14: An example of (detailed) action in the profile UML4SOA

is the possibility to install compensations of executed activities that are executed in case of failure as discussed in the following.

Firstly, we illustrate the various services of the scenario, their orchestration and the kind of exchanged message. The customer initially logins to the Portal by sending his ID, i.e. his username and password. The customer identity is confirmed by an Authentication. For each successful login, Portal generates a sessionID, i.e. a datum univocally identifying a session. The value sessionID is used by the various actors for exchanging messages after the customer logins. Each message has, in fact, a body argument containing the exchanged data, and the sessionID as further argument identifying the corresponding session. This guarantees that messages referring to different requests are not erroneously mixed together. Portal then sends the requestID, i.e. the couple of sessionID and customer username, to service Information Upload, that starts a conversation with the customer in order to fill the request. The customer balance is validated by a Validation service. The filled request is then sent to Request Processing. Request Processing relies on employee and supervisor Task List services (shortened into empTaskList and supTaskList, respectively) for storing the request that is successively retrieved by, respectively, an employee and a supervisor, each of whom fills an Evaluation and forwards it to Request Processing. An Evaluation has two parts, i.e. Public Evaluation and Private Evaluation. The former is a tuple containing the strings Offer (the offer made to the customer), Motivation (the motivations of the rejection or of the request for an update) and Decision, which is equal to one of the values Accept, Reject or AskToUpdate. The latter is a tuple containing the strings rating and AdditionalInfo. Together, a Request and its Evaluation form a Contract. If either the employee or the supervisor asks to update the request, the related Contract is sent to service Information Update, that asks to the customer whether he wants to update the request and, in case, sends the updated request back to Request Processing . Finally, when an agreement between the bank and the customer is established, the related Contract is forwarded to a Contract Processing service.

We now specify the behavior of the involved services, i.e. Portal, Information Upload, Information Update and Request Processing, whose internal behavior is fundamental for a correct specification and implementation of the whole workflow.

The diagram of Figure 2.15 relates to the interaction between the customer and service Portal. The customer ID is sent to the portal that starts a login scope. Portal synchronously exchanges messages with service Authentication, sending the customer ID and receiving back the boolean valid. If valid=No, the service sends back to the customer a message signaling the failure of the login and then raises the exception failedLogin that terminates the process. If valid=Yes, the service generates (by means of action ≪create≫)

a new sessionID and sends it back to the customer. Portal receives the customer choice about the desired service (here we only consider service Credit Request) and invokes service Information Upload (shortened into InfoUpload) sending to it a message with the requestID. From then on, the customer communicates with InfoUpload.

After the login, service InfoUpload (see Figure 2.16) starts a conversation with the customer whose purpose is to produce a Request. The service workflow immediately forks in two parallel branches, one responsible for collecting the data of the Request, while the other one awaits for a message cancel from the customer, meaning that the customer wants to abort the process. In the last case, an exception abort is raised and the process terminates. The branch responsible of collecting the data of the Request first receives the desired amount from the customer. After that, the customer may choose to send first either his balance or the securities (shortened sec), hence the workflow forks in two parallel branches awaiting to receive the messages with this two data. Moreover, the branch responsible of receiving the balance, sends it to service Validation, that replies with a message containing the boolean valid. If valid=Yes, the workflow proceeds, otherwise, it sends a message to the customer, asking to resend the balance, and then cycles and awaits to receive a new message. After both the branches are completed, service InfoUpload terminates by invoking Request Processing (shortened into reqProcessing) and sending the Request to it.

Service Information Update (shortened into InfoUpdate) is similar to the previous service (see Figure 2.17) but, unlike InfoUpload, it starts already receiving a Contract containing the existing Request and the Motivation for the request of an update. The Motivation and the request of an update are sent to the customer and the service awaits to receive an Answer. If Answer=No the process terminates, otherwise the workflow forks in two parallel branches. In one branch, the service asks the customer if he wants to update the securities: if the answer is positive, the service awaits to receive the new securities and then reaches a join point with the other branch, otherwise it immediately reaches the join point. The other branch does the same activities but with the amount in place of the securities. After both the branches have reached the join point, the service terminates by sending the updated Request back to service reqProcessing. In parallel with the described branch, the service starts another branch awaiting for a cancel request from the customer, exactly as in the case of InfoUpload described above.

As above specified, both InfoUpload and InfoUpdate send a request to reqProcessing (see Figure 2.18). As for InfoUpload and InfoUpdate, the workflow initially forks in two branches, one responsible of the main interactions, while the other awaits to receive a cancel message from the customer and, in case, triggers an abort exception. Regarding the main branch, the received request is sent to service empTaskList that queues it. It is possible that the customer decides to cancel the request after this step has been performed. In this case, the request must be deleted from the task list, in order to prevent an employee to examine an already aborted request. Hence, the action of sending the request must be compensated with a delete action removing it from the task list. For this purpose, the ser-

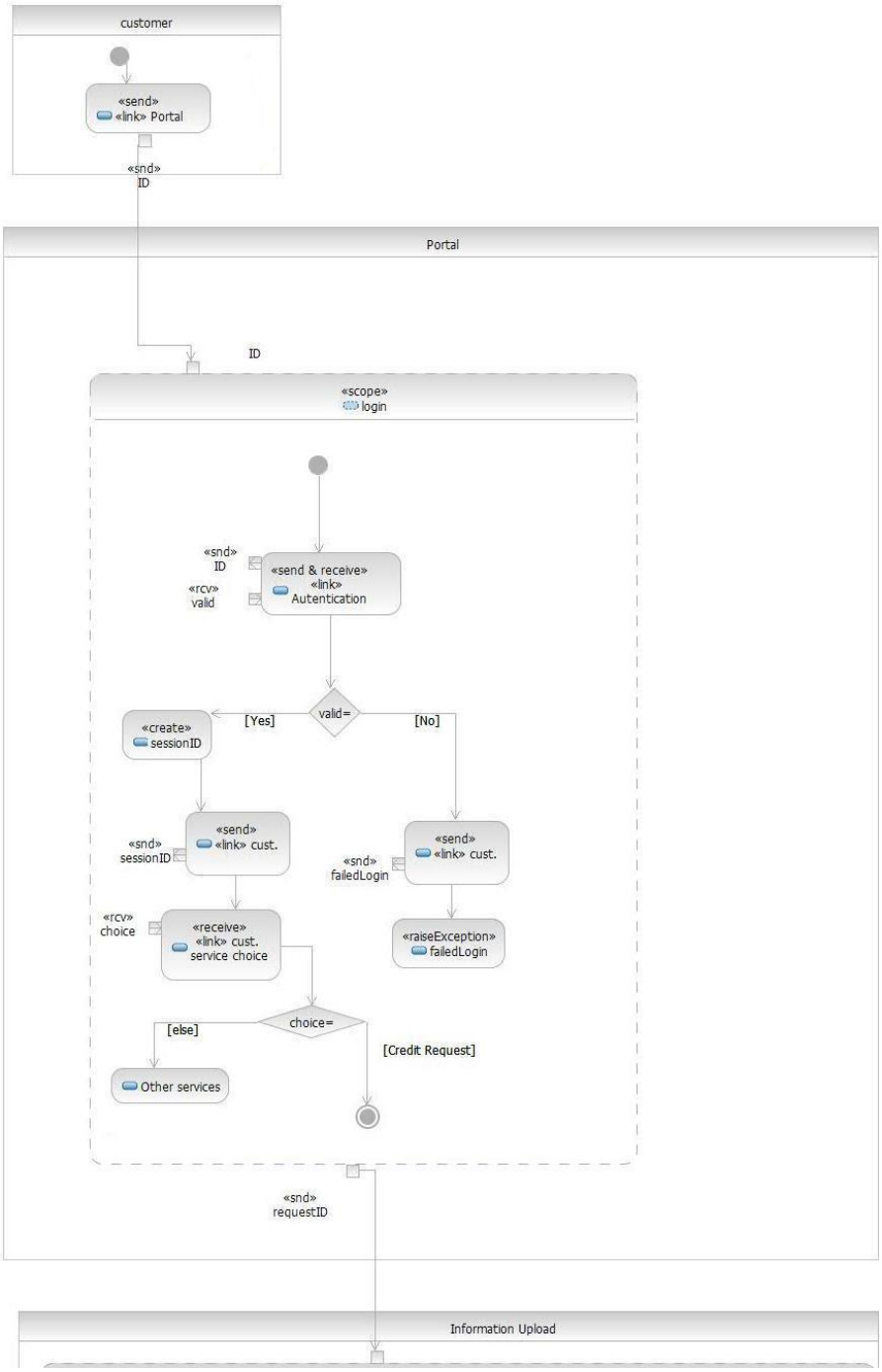

Figure 2.15: Service Portal activity diagram

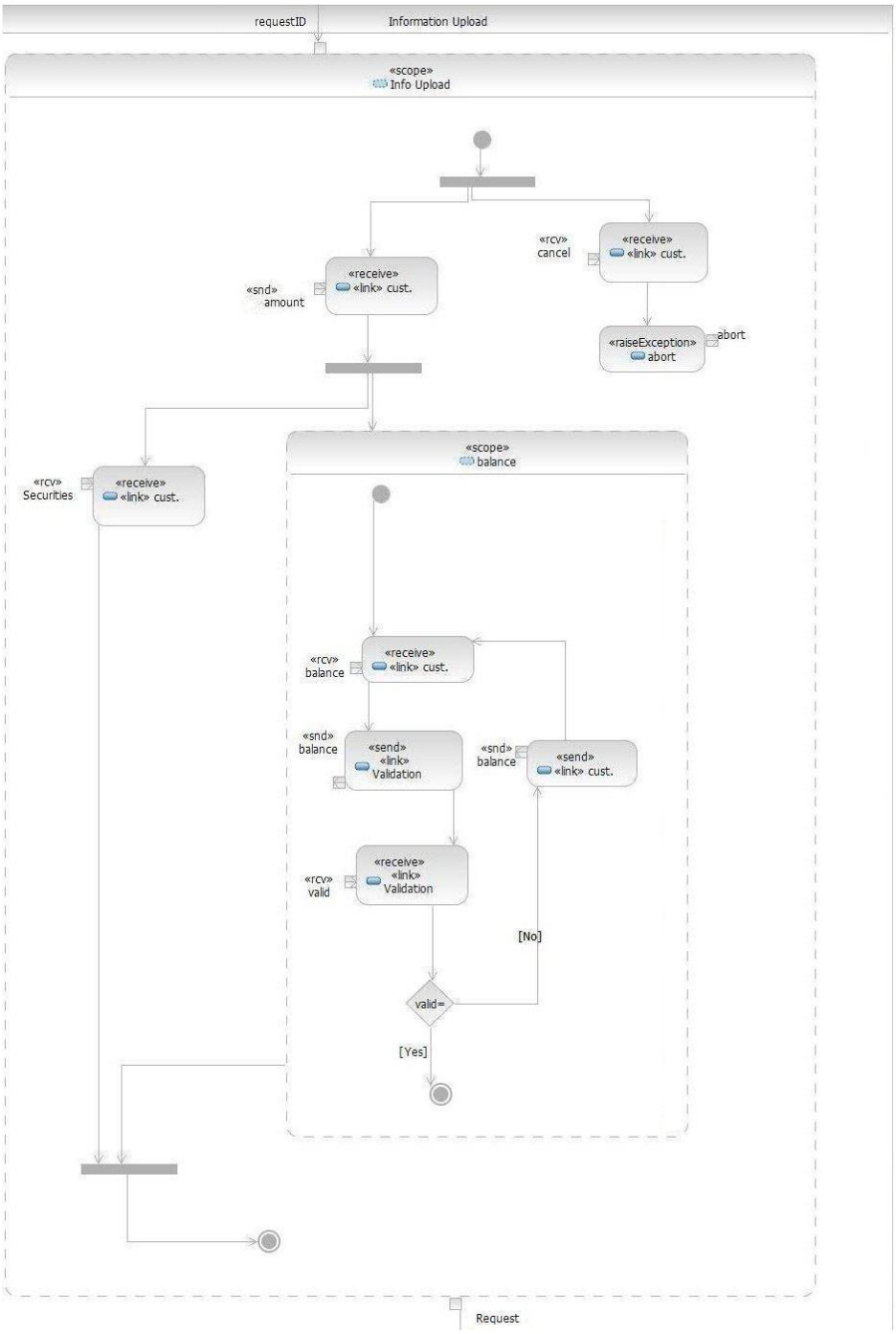

Figure 2.16: Service Information Upload activity diagram

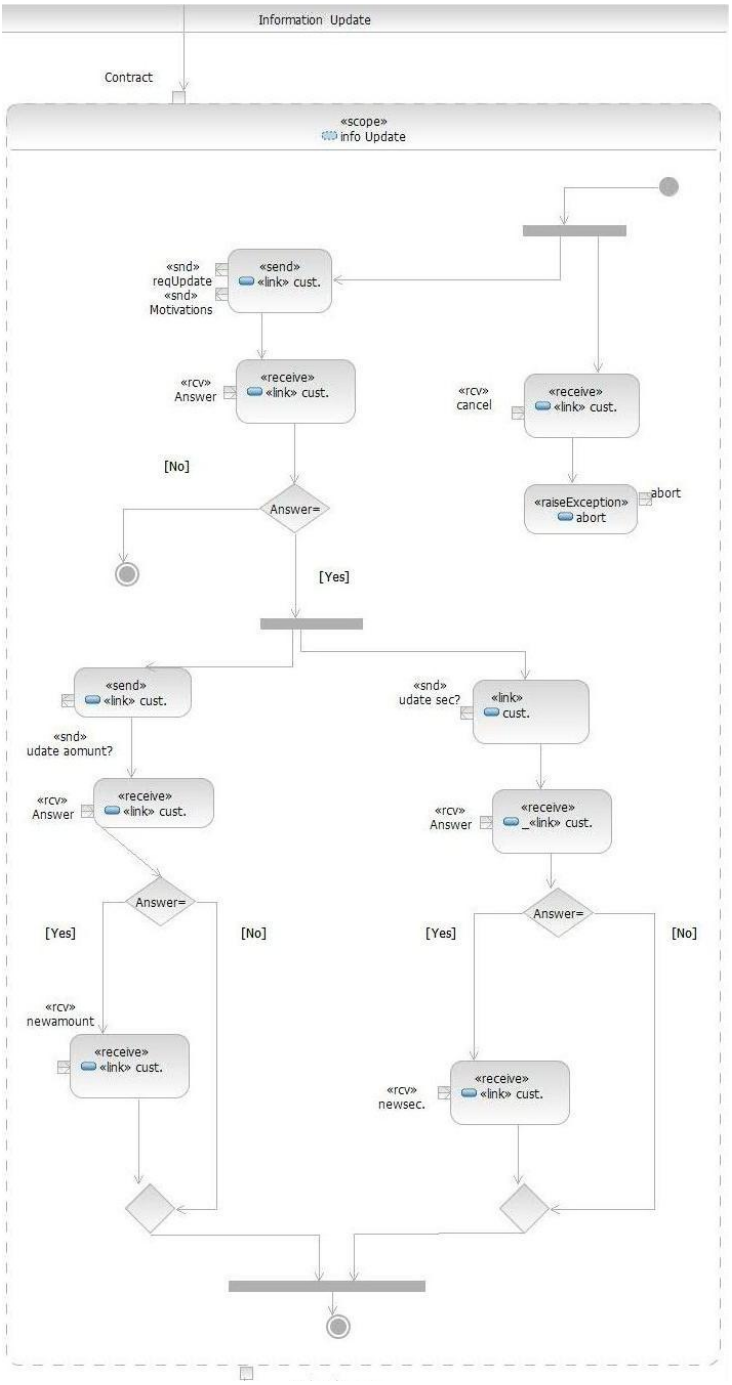

Figure 2.17: Service Information Update activity diagram

vice installs a 'compensation handler' consisting of an action sending the message Delete to service empTaskList. This message asks empTaskList to delete Request. Note that reqProcessing may not directly delete a request from a task list, since task lists are managed by services empTaskList and supTasklist that are autonomous from reqProcessing.

After sending the Request, reqProcessing awaits for the related Evaluation from an employee. The workflow then follows three alternative lines, according to the value of the argument Decision of Evaluation. If Decision=Reject, the service sends the Public Evaluation (shortened pubEvaluation), containing the decision and its motivation, to the customer and then terminates. If Decision=AskToUpdate, the service terminates by sending the Contract, containing the decision, its motivation and the request to InfoUpdate illustrated above. Finally, if Decision=Accept the second step of evaluation, similar to the described one, starts. The Contract is sent to supTaskList and the related compensation, asking for the deletion of the Contract from the supervisor task list, is installed. The service then awaits for the pubEvaluation by a supervisor. If Decision=Reject or Decision=AskToUpdate the service performs the same actions described above. If Decision=Accept, the service sends the pubEvaluation with the Decision and the bank Offer to the customer and awaits for his Answer. If Answer=No the process terminates, if Answer=Yes the service sends the Contract to a Contract Processing service and the process successfully terminates.

It still remains to examine the case when a customer asks to cancel the request while reqProcessing is running. As for services InfoUpload and InfoUpdate, the cancel message is received by a secondary branch of the process running in parallel with the main branch described above; then an exception abort is raised that eventually leads to process termination. However, before ending the process, some compensation activities may be required. The action ≪Compensate All≫ is executed; the meaning of this action is to execute all the installed compensations. Hence, if no compensation has yet been installed, the compensation activity is empty. If a request of deletion for the employee task list (and, possibly, for the supervisor task list) was installed, that compensation is executed.

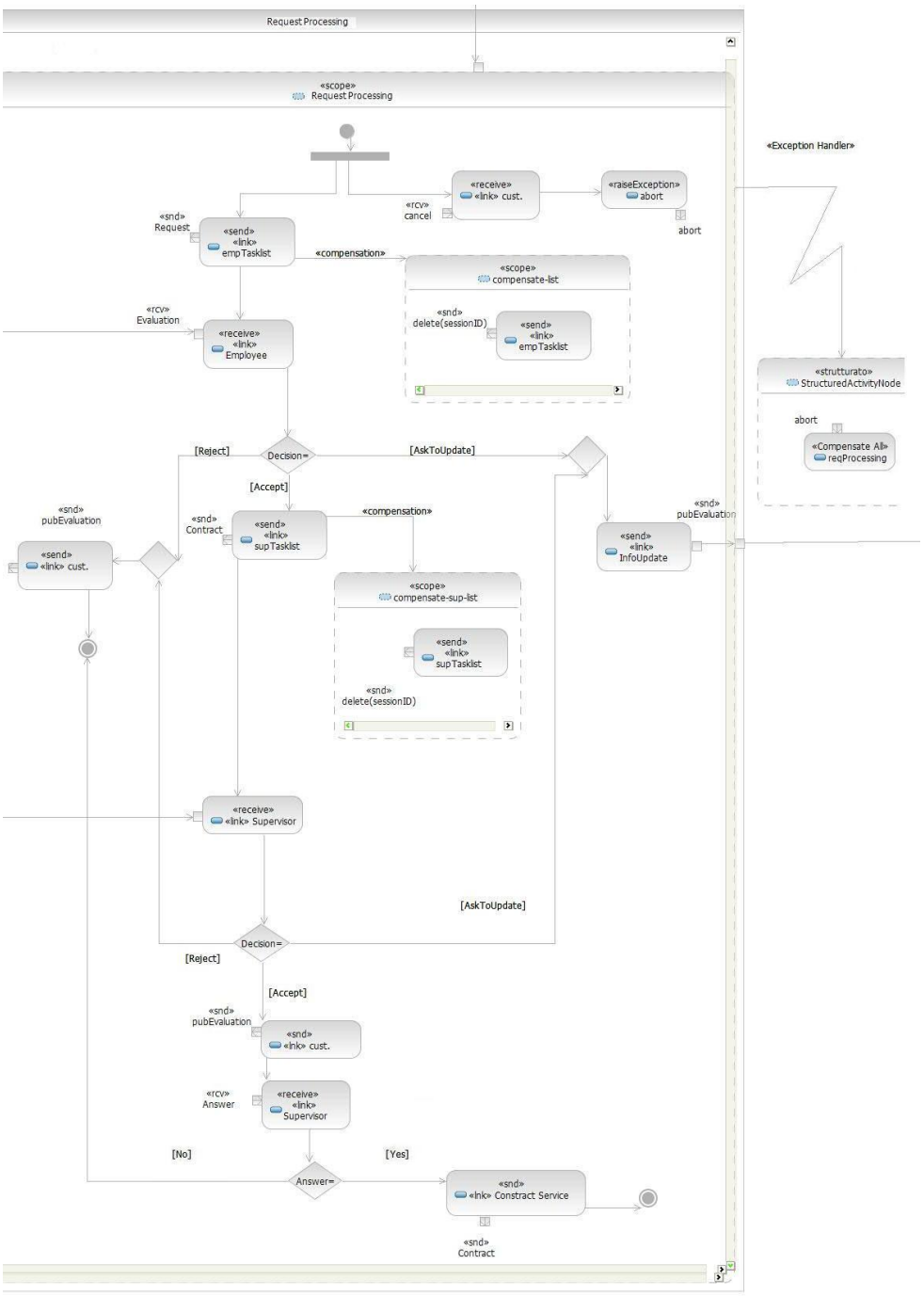

Figure 2.18: Service Request Processing activity diagram

Chapter 3

A Calculus for Orchestration of Web Services

In the previous chapter, we have discussed the motivations behind the necessity of rigorous formal foundations for current software engineering technologies for SOC, with special concern on the OASIS standard for orchestration of web services WS-BPEL.

Therefore, in this chapter, we introduce COWS as a formalism for specifying and orchestrating services while modelling their dynamic behaviour. COWS, in fact, falls within a main line of research (see e.g. [47, 50, 125, 130, 104, 34, 35, 52, 57, 127]) that aims at developing process calculi capable of capturing the basic aspects of service-oriented systems and, possibly, of supporting the analysis of qualitative and quantitative properties of services. The design of the calculus has been influenced by the principles underlying WS-BPEL, and in fact COWS supports service instances with shared states, allows a process to play more than one partner role, permits programming stateful sessions by correlating different service interactions, and enables management of long-running transactions. However, COWS intends to be a foundational model not specifically tight to web services' current technology. Thus, some WS-BPEL constructs, such as flow graphs and fault and compensation handlers, do not have a precise counterpart in COWS, rather they are expressed in terms of more primitive operators (see Sections 3.2.1.3 and 3.2.3.3). Of course, COWS has also taken advantage of previous work on process calculi. Indeed, it combines in an original way constructs and features borrowed from well-known process calculi, e.g. non-binding input activities, asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

We illustrate COWS's suitability for modelling SOC applications through many specific examples and the specification of the two case studies informally introduced at the end of the previous chapter.

Structure of the chapter. The rest of the chapter is organized as follows. Section 3.1 provides some insights into COWS's main features by means of an example. Section 3.2

presents COWS's syntax and operational semantics. Section 3.3 discusses the correspondence between WS-BPEL and COWS. Section 3.4 describes the more relevant parts of the COWS specifications of the two case studies. Section 3.5 touches upon more closely related work.

3.1 A 'Morra game' scenario

Before formally defining COWS, we provide some insights into its main features in a step-by-step fashion by means of an example. This is a service inspired by the well-known game Morra[1] and described at two different levels of abstraction.

Let us consider a service that allows its clients to play the Morra game. We consider the variation of Morra where two players, named "odds" and "evens", throw out a single hand, each showing zero to five fingers. If the sum of fingers shown by both players is an even number then the "evens" player wins; otherwise the "odds" player is the winner. The service collects the two throws (i.e. two integers), calculates the winner and sends the result back to the two players. A high-level specification of the service in COWS is:

$$\begin{array}{l} *\ [x_{id}, x_p, x_{num}, y_p, y_{num}] \\ \quad (\ odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \ \mid\ evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\ \qquad \mid\ x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \ \mid\ y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle\ ) \end{array} \tag{3.1}$$

The service receives throws from the players via two distinct endpoints, i.e. pairs $odds \bullet throw$ and $evens \bullet throw$, which can be interpreted as specific implementations of the *operation* name *throw* provided by *partner* names *odds* and *evens*. The players are required to provide the partner names, stored in variables x_p and y_p, that they will use to receive the result. The replication operator $*\_$, that spawns in parallel as many copies of its argument term as necessary, permits supporting creation of multiple instances to serve several matches simultaneously. A match is identified by a match-id, stored in x_{id}, that the partners need to provide when sending their throws. To avoid interferences between matches played simultaneously, match-ids should be unique (clients can use the delimitation operator $[\_]$ to guarantee uniqueness). Partner throws arrive randomly, thus any interaction with the service starts with one of the two *receive* activities $odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle$ or $evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle$, corresponding to reception of throws of the match identified by x_{id}, and terminates with the two *invoke* activities $x_p \bullet res!\langle x_{id,} win(x_{num}, y_{num}, 1)\rangle$ and $y_p \bullet res!\langle x_{id,} win(x_{num}, y_{num}, 0)\rangle$, used to reply with the result. We assume that $win(x, y, z)$ is a total function which, if x and y are integers between 0 and 5, returns the string w (abbreviation of 'winner') in case $(x + y)\ mod\ 2$ is equal to z, and the string l (abbreviation of 'loser') if $(x + y)\ mod\ 2$ is different from z; otherwise, the string *err* is returned.

[1] For further information about the Morra game visit the web site `http://en.wikipedia.org/wiki/Morra_(game)`.

Service (3.1) uses the delimitation operator to declare the scope of variables x_{id}, x_p, y_p, x_{num} and y_{num}. An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint match and causes replacement of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). Notably, the two receive activities are *correlated* by means of the *shared* variable x_{id}.

When an invocation for operation *throw* is processed, it must be checked if a service instance with the same match-id already exists, in which case the invocation is received by the instance, or if the service must produce a new instance. This is done through the *dynamic prioritised mechanism* of COWS, i.e. assigning the receives by instances (having a more defined pattern) a greater priority than the receives by the service definition.

Thus, for example, after an interaction with the following client

$$[z]\,(\,evens \bullet throw!\langle first, cbB, 1\rangle \mid cbB \bullet res?\langle first, z\rangle\,.\,\langle \text{rest of client B}\rangle\,)$$

service definition (3.1) runs in parallel with the instance identified by the match-id *first* (the instance is highlighted by a gray background)

$$\begin{aligned}
&* [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
&\quad (\,odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
&\qquad \mid x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle\,) \\
&\mid [x_p, x_{num}]\ (\,odds \bullet throw?\langle first, x_p, x_{num}\rangle \\
&\qquad\qquad \mid x_p \bullet res!\langle first, win(x_{num}, 1, 1)\rangle \mid cbB \bullet res!\langle first, win(x_{num}, 1, 0)\rangle\,)
\end{aligned}$$

Now, if another client performs the invocation $odds \bullet throw!\langle first, cbA, 2\rangle$, it will be processed by the already existing instance because, w.r.t. this invocation, the receive $odds \bullet throw?\langle first, x_p, x_{num}\rangle$ has greater priority than the receive $odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle$ (that has a less defined argument pattern). The long-running interaction between the Morra service definition (3.1) and the above clients is graphically represented in Figure 3.1, where each node represents a state, while each edge describes a communication action (e.g. label $evens \bullet throw\langle first, cbB, 1\rangle$ denotes taking place of a communication along endpoint $evens \bullet throw$ with matching values $\langle first, cbB, 1\rangle$).

For a lower level implementation, we wish to maximise the abilities of different services, while preserving the observable behaviour of the whole service w.r.t. the high-level specification. The main service is now composed of three entities as follows:

$$[req2f, req5f, resp2f, resp5f]\,(\,*M \mid *2F \mid *5F\,) \tag{3.2}$$

The delimitation operator is used here to declare that *req2f*, *req5f*, *resp2f* and *resp5f* are private operation names known to the three components *M*, *2F* and *5F*, and only to them. The three subservices are defined as follows:

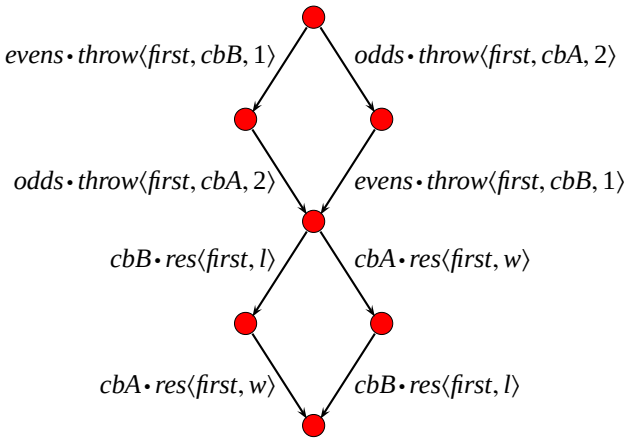

Figure 3.1: Long-running interaction between clients and high-level Morra specification

$$
\begin{aligned}
M \;\triangleq\;& [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
& (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
& \quad \mid [k]\,(\, m \bullet req2f!\langle x_{id}, x_{num}, y_{num}\rangle \mid m \bullet req5f!\langle x_{id}, x_{num}, y_{num}\rangle \\
& \qquad \mid [x_o, x_e]\, m \bullet resp2f?\langle x_{id}, x_o, x_e\rangle. \\
& \qquad\qquad (\, \mathbf{kill}(k) \mid \{\!|\, x_p \bullet res!\langle x_{id}, x_o\rangle \mid y_p \bullet res!\langle x_{id}, x_e\rangle \,|\!\}\,) \\
& \qquad \mid [x_o, x_e]\, m \bullet resp5f?\langle x_{id}, x_o, x_e\rangle. \\
& \qquad\qquad (\, \mathbf{kill}(k) \mid \{\!|\, x_p \bullet res!\langle x_{id}, x_o\rangle \mid y_p \bullet res!\langle x_{id}, x_e\rangle \,|\!\}\,)\,)\,) \\
2F \;\triangleq\;& [x]\,(\, m \bullet req2f?\langle x, 1, 1\rangle.\, m \bullet resp2f!\langle x, l, w\rangle \\
& \quad + \; m \bullet req2f?\langle x, 1, 2\rangle.\, m \bullet resp2f!\langle x, w, l\rangle \\
& \quad + \; m \bullet req2f?\langle x, 2, 1\rangle.\, m \bullet resp2f!\langle x, w, l\rangle \\
& \quad + \; m \bullet req2f?\langle x, 2, 2\rangle.\, m \bullet resp2f!\langle x, l, w\rangle\,) \\
5F \;\triangleq\;& [x, y, z]\,(\, m \bullet req5f?\langle x, y, z\rangle.\, m \bullet resp5f!\langle x, err, err\rangle \\
& \qquad + \; m \bullet req5f?\langle x, 0, 0\rangle.\, m \bullet resp5f!\langle x, l, w\rangle \\
& \qquad + \; m \bullet req5f?\langle x, 0, 1\rangle.\, m \bullet resp5f!\langle x, w, l\rangle \\
& \qquad + \; \ldots \; + \; m \bullet req5f?\langle x, 5, 5\rangle.\, m \bullet resp5f!\langle x, l, w\rangle\,)
\end{aligned}
$$

Service M is publicly invocable and can interact with players as well as with the 'internal' services $2F$ and $5F$. These latter two services, instead, can only be invoked by M and have the task of calculating the winner of a match. In particular, $2F$ performs a quick computation of simple matches where both players hold out either one or two fingers, while $5F$ performs a slower computation of standard 5-fingers matches (that exactly corresponds to the computation modelled by the function $win(\_)$). After the two initial receives, for e.g. performance and fault tolerance purposes, M invokes services $2F$ and $5F$ concurrently. Communication between M and the other two subservices relies on the match identifier (stored in x) as correlation data. When one of $2F$ and $5F$ replies, M immediately stops the other computation. This is done by executing the *kill* activity $\mathbf{kill}(k)$, that forces termination of all unprotected parallel terms inside the enclosing $[k]$, which stops the killing effect. Kill activities are executed *eagerly* w.r.t. the other parallel activ-

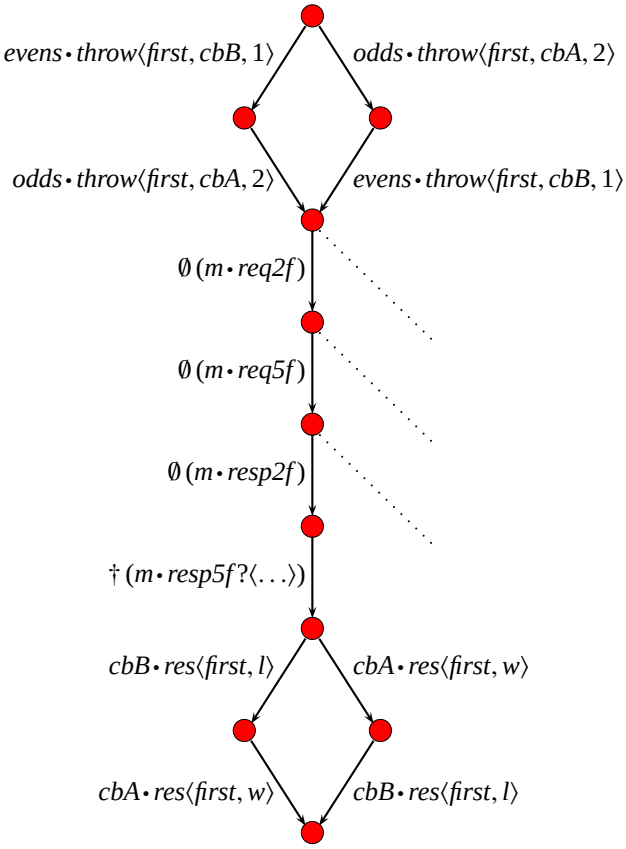

Figure 3.2: Long-running interaction between clients and low-level Morra specification

ities included within the enclosing scope, because relatively to these latter activities they take the greatest priority. However, critical activities can be protected from the effect of a forced termination by using the *protection* operator $\{\!|\_|\!\}$; this is indeed the case of the response $x_p \bullet res!\langle x_{id}, x_o\rangle$ in our example. Finally, M forwards the responses to the players and terminates.

Services *2F* and *5F* use the *choice* operator $\_ + \_$ to offer alternative behaviours: one of them can be selected by executing an invoke matching the receive leading the behaviour. If the throws are not integers between 0 and 5, *2F* does not reply, while *5F* returns the string *err*. Indeed, the receive $m \bullet req5f?\langle x, y, z\rangle$ is assigned less priority than the other receive activities, i.e. it is only executed when none of the other receives matches the two throws, thus avoiding to return *err* in case of admissible throws.

An interaction of specification (3.2) with the previous clients is shown in Figure 3.2, where $\emptyset(m \bullet o)$ represents an internal communication along the endpoint $m \bullet o$, while $\dagger(m \bullet o?\bar{w})$ denotes an invisible kill-action that terminates the activity $m \bullet o?\bar{w}$ (and its continuation). Dotted lines stand for alternative behaviours. Notably, after execution of a communication along $m \bullet resp2f$, the prioritised semantics of COWS permits executing only the kill action.

3.2 The language COWS

To gradually introduce the technicalities and distinctive features of COWS, we present its syntax and operational semantics in three steps. More specifically, in Section 3.2.1 we consider μCOWS^m (μCOWS *minus priority*), the fragment of COWS without priority in parallel composition and linguistic constructs dealing with termination. It retains all the other COWS's features, like e.g. global scope and pattern matching. In Section 3.2.2 we move on μCOWS (*micro* COWS), the calculus obtained by enriching μCOWS^m with priority in parallel composition. Finally, in Section 3.2.3 we study the full calculus, that extends μCOWS with primitives for termination. For each of the three calculi we show some simple clarifying examples.

3.2.1 μCOWS^m : the priority-, protection- and kill-free fragment of COWS

The fragment of COWS introduced in this section, namely μCOWS^m, dispenses with priority in parallel composition and linguistic constructs dealing with termination.

3.2.1.1 Syntax

The syntax of μCOWS^m is presented in Table 3.1. We use two countable disjoint sets: the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names* (ranged over by n, m, p, o, ...) mainly used to represent partners and operations. We also use a set of *expressions* (ranged over by ϵ), whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, values and variables.

Partner names and operation names can be combined to designate *endpoints*, written $p \bullet o$. In fact, alike channels in [60], an endpoint is not atomic but results from the composition of a partner name p and of an operation name o, which can also be interpreted as a specific implementation of o provided by p. This results in a very flexible naming mechanism that allows a service to be identified by means of different logic names (i.e. to play more than one partner role as in WS-BPEL). For example, the following service

$$p_{slow} \bullet o?\bar{w}.s_{slow} + p_{fast} \bullet o?\bar{w}.s_{fast}$$

accepts requests for the same operation o through different partners with distinct access modalities: process s_{slow} implements a slower service provided when the request is processed through the partner p_{slow}, while s_{fast} implements a faster service provided when the request arrives through p_{fast}. Additionally, it allows the names composing an endpoint to be dealt with separately, as in a request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to. For example, the ping service $p \bullet o_{req}?\langle x \rangle.x \bullet o_{res}!\langle \text{“}\texttt{I live}\text{”} \rangle$ will know at run-time the partner name for the reply activity. This mechanisms is also

s	::=	(services)	g	::=	(receive-guarded choice)
	$u \bullet u'!\bar{\epsilon}$	(invoke)		$\mathbf{0}$	(nil)
|	g	(receive-guarded choice)	|	$p \bullet o?\bar{w}.s$	(request processing)
|	$s \mid s$	(parallel composition)	|	$g + g$	(choice)
|	$[u]\, s$	(delimitation)			
|	$* s$	(replication)			

Table 3.1: μCOWS^m syntax

sufficiently expressive to support implementation of explicit locations: a located service can be represented by using a same partner for all its receiving endpoints. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, dynamically received names can only be used for service invocation (as in *localised π-calculus* [147]). Indeed, endpoints of receive activities are identified statically because their syntax only allows using names and not variables.

Remark 3.2.1 (Localised receive activities) As in localised π-calculus and differently from the standard π-calculus, COWS disallows 'input capability', i.e. the ability of services to receive a name and subsequently accept inputs along an endpoint containing such name. This choice is motivated, on the one hand, by the fact that the design of COWS has been influenced by the current (web) service technologies where endpoints of receive activities are statically determined (recall that service endpoints are not π-calculus channels) and, on the other hand, by the will to support an easier implementation of the calculus. However, the former is the major motivation. In fact, implementation problems due to input capability can be solved by relying on the theory of linear forwarders [99] as in `PiDuce` [64].

To model asynchronous communication, invoke activities cannot be used as prefixes and choice can only be guarded by receive activities (as in *asynchronous π-calculus* [7]). Indeed, in service-oriented systems, communication paradigms are usually asynchronous (mainly for scalability reasons [43]), in the sense that there may be an arbitrary delay between the sending and the receiving of a message, the ordering in which messages are received may differ from that in which they were sent, and a sender cannot determine if and when a sent message will be received.

In the sequel, w ranges over values and variables and u ranges over names and variables. Notation $\bar{\cdot}$ stands for tuples, e.g. $\bar{x}$ means $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$) where variables in the same tuple are pairwise distinct. We write $a, \bar{b}$ to denote the tuple obtained by concatenating the element a to the tuple $\bar{b}$. All notations shall extend to tuples componentwise. $\mathtt{n}$ ranges over communication endpoints that do not contain variables (e.g. $p \bullet o$), while $\mathtt{u}$ ranges over communication endpoints that may contain variables (e.g. $u \bullet u'$). Sometimes, we will use notation $\mathtt{n}$ and $\mathtt{u}$ for the tuples $\langle p, o \rangle$ and $\langle u, u' \rangle$, respectively, and

$*\mathbf{0}$	$\equiv$	$\mathbf{0}$	$*s$	$\equiv$	$s \mid *s$			
$s \mid \mathbf{0}$	$\equiv$	s	$s_1 \mid s_2$	$\equiv$	$s_2 \mid s_1$	$(s_1 \mid s_2) \mid s_3$	$\equiv$	$s_1 \mid (s_2 \mid s_3)$
$g + \mathbf{0}$	$\equiv$	g	$g_1 + g_2$	$\equiv$	$g_2 + g_1$	$(g_1 + g_2) + g_3$	$\equiv$	$g_1 + (g_2 + g_3)$
$[u]\,\mathbf{0}$	$\equiv$	$\mathbf{0}$	$[u_1]\,[u_2]\,s$	$\equiv$	$[u_2]\,[u_1]\,s$	$s_1 \mid [u]\,s_2$	$\equiv$	$[u]\,(s_1 \mid s_2)$ if $u \notin \mathrm{fu}(s_1)$

Table 3.2: μCOWS^m structural congruence

rely on the context to resolve any ambiguity. When convenient, we shall regard a tuple (hence, also an endpoint) simply as a set, writing e.g. $x \in \bar{y}$ to mean that x is an element of $\bar{y}$. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[\langle u_1, \ldots, u_n \rangle]\,s$ in place of $[u_1] \ldots [u_n]\,s$. We will write $I \triangleq s$ to assign a name I to the term s.

We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice.

The only *binding* construct is delimitation: $[u]\,s$ binds u in the scope s. In fact, to enable concurrent threads within each service instance to share (part of) the state, receive activities in COWS bind neither names nor variables. This is different from most process calculi and somewhat similar to update [168] and fusion [169] calculi. In COWS, however, inter-service communication give rise to substitutions of variables with values (alike [168]), rather than to fusions of names (as in [169]). The range of application of the substitutions generated by a communication is regulated by the delimitation operator, that additionally permits to generate fresh names (as the restriction operator of π-calculus). Thus, the occurrence of a name/variable is *free* if it is not under the scope of a delimitation for it. Bound and free names are also called *private* and *public* names, respectively. We denote by $\mathrm{fu}(t)$ the set of free names/variables that occur free in t. Two terms are *α-equivalent* if one can be obtained from the other by consistently renaming bound names/variables. As usual, we identify terms up to α-equivalence.

3.2.1.2 Operational semantics

The operational semantics of μCOWS^m is defined only for *closed* services, i.e. services without free variables. Formally, the semantics is given in terms of a structural congruence and of a labelled transition relation. The *structural congruence*, written $\equiv$, identifies syntactically different services that intuitively represent the same service. It is defined as the least congruence relation induced by the equational laws shown in Table 3.2. All the laws are straightforward. In particular, commutativity of consecutive delimitations implies that the order among the u_i in $[\langle u_1, \ldots, u_n \rangle]\,s$ is irrelevant, thus in the sequel we may use the simpler notation $[u_1, \ldots, u_n]\,s$. The last law permits to extend the scope of names (as in the π-calculus) and variables, thus enabling possible communication.

To define the labelled transition relation, we use two auxiliary functions. Firstly, we use the function $[\![\_]\!]$ for evaluating *closed* expressions (i.e. expressions without vari-

$$\mathcal{M}(x, v) = \{x \mapsto v\} \quad \mathcal{M}(v, v) = \emptyset \quad \mathcal{M}(\langle\rangle, \langle\rangle) = \emptyset \quad \frac{\mathcal{M}(w_1, v_1) = \sigma_1 \quad \mathcal{M}(\bar{w}_2, \bar{v}_2) = \sigma_2}{\mathcal{M}((w_1, \bar{w}_2), (v_1, \bar{v}_2)) = \sigma_1 \uplus \sigma_2}$$

Table 3.3: Matching rules

$$\frac{[\![\bar{\epsilon}]\!] = \bar{v}}{\mathtt{n}!\bar{\epsilon} \xrightarrow{\mathtt{n} \lhd \bar{v}} \mathbf{0}}\ (inv) \qquad \mathtt{n}?\bar{w}.s \xrightarrow{\mathtt{n} \rhd \bar{w}} s\ (rec) \qquad \frac{g \xrightarrow{\alpha} s}{g + g' \xrightarrow{\alpha} s}\ (choice)$$

$$\frac{s_1 \xrightarrow{\mathtt{n} \rhd \bar{w}} s_1' \qquad s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma}{s_1 \mid s_2 \xrightarrow{\sigma} s_1' \mid s_2'}\ (com) \qquad \frac{s_1 \xrightarrow{\alpha} s_1'}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (par)$$

$$\frac{s \xrightarrow{\sigma \uplus \{x \mapsto v\}} s'}{[x]\, s \xrightarrow{\sigma} s' \cdot \{x \mapsto v\}}\ (del_{com}) \qquad \frac{s \xrightarrow{\alpha} s' \quad u \notin \mathrm{u}(\alpha)}{[u]\, s \xrightarrow{\alpha} [u]\, s'}\ (del) \qquad \frac{s \equiv \xrightarrow{\alpha} \equiv s'}{s \xrightarrow{\alpha} s'}\ (str)$$

Table 3.4: μCOWS^m operational semantics

ables): it takes a closed expression and returns a value. It is not explicitly defined since the exact syntax of expressions is deliberately not specified. Secondly, we use the partial function $\mathcal{M}(\_, \_)$ for performing *pattern-matching* on semi-structured data and, thus, determining if a receive and an invoke over the same endpoint can synchronise. The rules defining $\mathcal{M}(\_, \_)$ are shown in Table 3.3. They state that two tuples match if they have the same number of fields and corresponding fields have matching values/variables. Variables match any value, and two values match only if they are identical. When tuples $\bar{w}$ and $\bar{v}$ do match, $\mathcal{M}(\bar{w}, \bar{v})$ returns a substitution for the variables in $\bar{w}$; otherwise, it is undefined. *Substitutions* (ranged over by σ) are functions mapping variables to values and are written as collections of pairs of the form $x \mapsto v$. Application of substitution σ to s, written $s \cdot \sigma$, has the effect of replacing every free occurrence of x in s with v, for each $x \mapsto v \in \sigma$, by possibly using α-conversion for avoiding v to be captured by name delimitations within s. We use $\emptyset$ to denote the empty substitution, $|\sigma|$ to denote the number of pairs in σ, and $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

The *labelled transition relation* $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Table 3.4, where label α is generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd \bar{v} \quad \mid \quad \mathtt{n} \rhd \bar{w} \quad \mid \quad \sigma$$

The meaning of labels is as follows: $\mathtt{n} \lhd \bar{v}$ and $\mathtt{n} \rhd \bar{w}$ denote execution of invoke and receive activities over the endpoint $\mathtt{n}$ with arguments $\bar{v}$ and $\bar{w}$, respectively; σ denotes execution of a communication with generated substitution σ to be still applied. $\emptyset$ denotes a *computational step* corresponding to taking place of communication without pending

substitutions. In the sequel, we will use $\mathrm{u}(\alpha)$ to denote the set of names and variables occurring in α, where $\mathrm{u}(\{x \mapsto v\}) = \{x\} \cup \mathrm{fu}(v)$ and $\mathrm{u}(\sigma_1 \uplus \sigma_2) = \mathrm{u}(\sigma_1) \cup \mathrm{u}(\sigma_2)$.

We comment on salient points. A service invocation can proceed only if the expressions in the argument can be evaluated (rule *(inv)*). This means, for example, that if it contains a variable x (in its endpoint or argument) it is stuck until x is not replaced by a value because of execution of a receive assigning a value to x. A receive activity offers an invocable operation along a given partner name (rule *(rec)*), and execution of a receive permits to take a decision between alternative behaviours (rule *(choice)*). Communication can take place when two parallel services perform matching receive and invoke activities (rule *(com)*). Communication generates a substitution that is recorded in the transition label (for subsequent application), rather than a silent transition as in most process calculi. Execution of parallel services is interleaved (rule *(par)*). When the delimitation of a variable x argument of a receive involved in a communication is encountered, i.e. the whole scope of the variable is determined, the delimitation is removed and the substitution for x is applied to the term (rule *(del$_{com}$)*). Variable x disappears from the term and cannot be reassigned a value (for this reason we say that COWS's variables are 'write once'). $[u]\, s$ behaves like s (rule *(del)*), except when the transition label α contains u. Rule *(str)* is standard and states that structurally congruent services have the same transitions.

We end with a property of the operational semantics regarding computational steps.

Property 3.2.1 *Let s be a μCOWS^m closed term. If $s \xrightarrow{\sigma} s'$, then $\sigma = \emptyset$ and s' is closed.*

The above property can be proved by a straightforward induction on the depth of the shortest inference for the transitions in the hypothesis, by exploiting the fact that s is closed.

3.2.1.3 Examples

We report here a few observations and examples aimed at clarifying the peculiarities of μCOWS^m.

Communication. Communication can exploit scope extension (last law of Table 3.2) to allow receive and invoke activities to interact. In fact, they can synchronise only if both are in the scopes of the delimitations that bind the variables argument of the receive. Thus, we must possibly extend the scopes of some variables, as in the following example:

$$\begin{aligned} &\mathit{odds} \bullet \mathit{throw}!\langle \mathit{first}, \mathit{cbA}, 2\rangle \mid [x_p, x_{num}]\,(\mathit{odds} \bullet \mathit{throw}?\langle \mathit{first}, x_p, x_{num}\rangle.\, s \mid s') \quad \equiv \\ &[x_p, x_{num}]\,(\mathit{odds} \bullet \mathit{throw}!\langle \mathit{first}, \mathit{cbA}, 2\rangle \mid \mathit{odds} \bullet \mathit{throw}?\langle \mathit{first}, x_p, x_{num}\rangle.\, s \mid s') \quad \xrightarrow{\emptyset} \\ &(s \mid s') \cdot \{x_p \mapsto \mathit{cbA}\,, x_{num} \mapsto 2\} \end{aligned}$$

Notice that the substitution $\{x_p \mapsto \mathit{cbA}\,, x_{num} \mapsto 2\}$ is applied to all terms delimited by $[x_p, x_{num}]$, not only to the continuation s of the service performing the receive. This is

different from most process calculi and accounts for the global scope of variables. This very feature permits to easily model the *delayed input* of fusion calculus [169], which is instead difficult to express in π-calculus.

Communication of private names. Communication of private names is standard and exploits scope extension as in π-calculus. To enable communication of private names, besides their scopes, we must possibly extend the scopes of some variables. Consider to modify the previous example by restricting the scope of the partner name *cbA* to the invoke activity, with *cbA* fresh in s and s'. Now, the communication can take place as follow:

$$\begin{array}{ll} [cbA]\,(odds \bullet throw!\langle first, cbA, 2\rangle) \mid [x_p, x_{num}]\,(odds \bullet throw?\langle first, x_p, x_{num}\rangle.\ s \mid s') & \equiv \\ [cbA]\,(odds \bullet throw!\langle first, cbA, 2\rangle \mid [x_p, x_{num}]\,(odds \bullet throw?\langle first, x_p, x_{num}\rangle.\ s \mid s')) & \equiv \\ [cbA, x_p, x_{num}]\,(odds \bullet throw!\langle first, cbA, 2\rangle \mid odds \bullet throw?\langle first, x_p, x_{num}\rangle.\ s \mid s') & \xrightarrow{\emptyset} \\ [cbA]\,(s \mid s') \cdot \{x_p \mapsto cbA\,, x_{num} \mapsto 2\} & \end{array}$$

XML messages. Nested tuples can be roughly used to represent XML documents, the standard format of messages exchanged among web services, by adopting the convention that the first field of each tuple acts as a 'tag'[2] (like originally proposed in the coordination language Linda [100]). For example, the following XML message representing a paper reference

```
<paper>
    <title> A Calculus for Orchestration of Web Services </title>
    <authors>
        <author> Lapadula </author>
        <author> Pugliese </author>
        <author> Tiezzi </author>
    </authors>
    <conference> ESOP </conference>
    <year> 2007 </year>
</paper>
```

could be rendered through the following COWS tuple

$$\begin{array}{l} \langle \texttt{paper}, \langle \texttt{title}, \text{A Calculus for Orchestration of Web Services} \rangle\,, \\ \quad \langle \texttt{authors}, \langle \texttt{author}, \text{Lapadula} \rangle\,, \langle \texttt{author}, \text{Pugliese} \rangle\,, \langle \texttt{author}, \text{Tiezzi} \rangle\,\rangle\,, \\ \quad \langle \texttt{conference}, \text{ESOP} \rangle\,, \\ \quad \langle \texttt{year}, 2007 \rangle\,\rangle \end{array}$$

Thus, to extract the title and the name of the second author of the paper above, one can use the following pattern (as argument of a receive activity):

[2] Element attributes could be rendered in a similar way.

$$\langle \texttt{paper}, \langle \texttt{title}, x_{title} \rangle, \langle \texttt{authors}, \_, \langle \texttt{author}, x_{secName} \rangle, \_ \rangle, \_, \_ \rangle$$

where, for simplicity sake, we assume that the *don't care* symbol _ matches any value/tuple.

Service instances and message correlation. The replication operator, that spawns in parallel as many copies of its argument term as necessary (law $*\, s \equiv s \mid *\, s$ of Table 3.2), permits specifying *persistent* services, i.e. services capable of creating multiple instances to serve several requests simultaneously.

The loosely coupled nature of SOC implies that the connection between communicating instances should not be assumed to persist for the duration of a whole business activity. Therefore, it is up to each single message to provide a form of context that enables services to associate the message with others. This is achieved by embedding values, called *correlation data*, in the content of the message itself. Pattern-matching is the mechanism for locating such data important to identify service instances for the delivering of messages.

Consider, for example, the following (persistent) service definition running in parallel with two clients:

$$\begin{array}{l}
(\, odds \bullet throw!\langle first, cbA, 2\rangle \mid odds \bullet throw!\langle second, cbA, 3\rangle \mid s_A \,) \\
\mid (\, evens \bullet throw!\langle first, cbB, 1\rangle \mid s_B \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}]\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle .\, evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle .\, s
\end{array}$$

After a computational step, due to the interaction between the service definition and the client A, a new instance identified by the correlation datum *first* runs in parallel with the other terms:

$$\begin{array}{l}
(\, odds \bullet throw!\langle second, cbA, 3\rangle \mid s_A \,) \\
\mid (\, evens \bullet throw!\langle first, cbB, 1\rangle \mid s_B \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}]\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle .\, evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle .\, s \\
\quad \mid [y_p, y_{num}]\, evens \bullet throw?\langle first, y_p, y_{num}\rangle .\, s \cdot \{x_{id} \mapsto first\,, x_p \mapsto cbA\,, x_{num} \mapsto 2\}
\end{array}$$

If, again, the client A invokes the service, a second instance, identified by the correlation datum *second*, is created:

$$\begin{array}{l}
s_A \\
\mid (\, evens \bullet throw!\langle first, cbB, 1\rangle \mid s_B \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}]\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle .\, evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle .\, s \\
\quad \mid [y_p, y_{num}]\, evens \bullet throw?\langle first, y_p, y_{num}\rangle .\, s \cdot \{x_{id} \mapsto first\,, x_p \mapsto cbA\,, x_{num} \mapsto 2\} \\
\quad \mid [y_p, y_{num}]\, evens \bullet throw?\langle second, y_p, y_{num}\rangle .\, s \cdot \{x_{id} \mapsto second\,, x_p \mapsto cbA\,, x_{num} \mapsto 3\}
\end{array}$$

Now, the client B invokes the service and, since the sent message contains the correlation datum *first*, the interaction takes place with the first service instance (indeed,

$\mathcal{M}(\langle second, y_p, y_{num}\rangle, \langle first, cbB, 1\rangle)$ does not hold):

$$
\begin{array}{l}
s_A \\
\mid\ s_B \\
\mid\ *\,[x_{id}, x_p, x_{num}, y_p, y_{num}]\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle.\, evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle.\, s \\
\quad \mid\ (s \cdot \{x_{id} \mapsto first\,, x_p \mapsto cbA\,, x_{num} \mapsto 2\}) \cdot \{y_p \mapsto cbB\,, y_{num} \mapsto 1\} \\
\quad \mid\ [y_p, y_{num}]\, evens \bullet throw?\langle second, y_p, y_{num}\rangle.\, s \cdot \{x_{id} \mapsto second\,, x_p \mapsto cbA\,, x_{num} \mapsto 3\}
\end{array}
$$

Therefore, although two instances waiting for a message along the endpoint $evens \bullet throw$ were available when B invoked the service, the message sent by B has been delivered to the correct instance. This behaviour is achieved simply by allowing the two receive activities of the service definition to share the variable x_{id}, used to store the correlation datum.

It is worth noticing that, as witnessed by the above example, this correlation mechanism is flexible enough for allowing a single message to participate in *multiparty conversations* (indeed, the above conversation involves one provider service and two clients).

Notice also that, differently from other correlation-based formal languages for SOC, such as WS-CALCULUS [130], SOCK [104] and B*lite* [135], correlation variables in COWS are not syntactically distinguished by other data variables. In fact, correlation variables can be recognized by their use (as in the example above, where x_{id} is used as argument of two consecutive receives). This is due to the fact that COWS intends to be a foundational model (specifically, a process calculus), with a small number of simple primitives.

Imperative constructs. We present how some higher level imperative constructs can be rendered in μCOWS^m.

Suppose to add a *matching with assignment* construct $[w = \epsilon]$ to COWS basic activities. Hence, we can also write terms of the form $[w = \epsilon].s$ whose intended semantics is that, if w and ϵ do match, a substitution is returned that will eventually assign to the variable in w the corresponding value of ϵ, and service s can proceed. In COWS, this meaning can be rendered through the following encoding

$$\langle\!\langle [w = \epsilon].s \rangle\!\rangle \;=\; [\mathtt{n}]\,(\mathtt{n}!\langle\epsilon\rangle \mid \mathtt{n}?\langle w\rangle.\langle\!\langle s\rangle\!\rangle) \tag{3.3}$$

for $\mathtt{n}$ fresh. The new construct generalizes standard assignment because it allows values to occur on the left of =, in which case it behaves as a matching mechanism. Similarly, we can encode *conditional choice* as follows:

$$\langle\!\langle \textbf{if } (\epsilon) \textbf{ then } \{s_1\} \textbf{ else } \{s_2\} \rangle\!\rangle \;=\; [\mathtt{n}]\,(\mathtt{n}!\langle\epsilon\rangle \mid (\mathtt{n}?\langle \textbf{true}\rangle.\langle\!\langle s_1\rangle\!\rangle + \mathtt{n}?\langle \textbf{false}\rangle.\langle\!\langle s_2\rangle\!\rangle)\,) \tag{3.4}$$

where **true** and **false** are the values that can result from evaluation of ϵ.

Like the receive activity, matching with assignment does not bind the variables on the left of =, thus it cannot reassign a value to them if a value has already been assigned. Therefore, the behaviour of matching with assignment may differ from standard assignment, even when the former uses only variables on the left of = as the latter does. For

example, activity $[x = 1]$ will not necessarily generate substitution $\{x \mapsto 1\}$. In fact, when it will be executed, x could have been previously replaced by a value v in which case execution of the activity corresponds to checking if v and 1 do match. For similar reasons, activity $[x = x + 1]$ does not have the effect of increasing the value of x by 1, but that of checking if the value of x and that of $x + 1$ do match, which always fails.

Standard variables (that can be repeatedly assigned) can be rendered as services providing 'read' and 'write' functionalities. When the service variable is initialized (i.e. the first time the 'write' operation is used), an instance is created that is able to provide the value currently stored. When this value must be read (resp. updated), the current instance terminates and a new instance is created which stores the current (resp. new) value. Here is the specification:

$$\begin{aligned} \mathit{Var}_{\mathtt{x}} \triangleq\ & [x_v, x_a]\, \mathtt{x} \bullet \mathit{write}?\langle x_v, x_a\rangle. \\ & [\mathtt{n}]\,(\,\mathtt{n}!\langle x_v, x_a\rangle \\ & \quad \mid *\,[x, y]\,\mathtt{n}?\langle x, y\rangle.\,(\,y!\langle\rangle \\ & \qquad \mid [x', y']\,(\,\mathtt{x} \bullet \mathit{read}?\langle y'\rangle.\,[\mathtt{m}]\,(\,\mathtt{n}!\langle x, \mathtt{m}\rangle \mid \mathtt{m}?\langle\rangle.\,y'!\langle x\rangle\,) \\ & \qquad\quad + \mathtt{x} \bullet \mathit{write}?\langle x', y'\rangle.\,\mathtt{n}!\langle x', y'\rangle\,)\,)\,) \end{aligned}$$

where $\mathtt{x}$ is a public partner name. Service $\mathit{Var}_{\mathtt{x}}$ provides two operations: *read*, for getting the current value; *write*, for replacing the current value with a new one. To access the service, a user must invoke these operations by providing a communication endpoint for the reply and, in case of *write*, the value to be stored. The *write* operation can be invoked along the public partner $\mathtt{x}$, which corresponds, the first time, to initialization of the variable. Thus, $\mathit{Var}_{\mathtt{x}}$ uses the delimited endpoint $\mathtt{n}$ in which to store the current value of the variable. This last feature is exploited to implement operations *read* and *write* in terms of re-instantiation. Notably, notation $\mathit{Var}_{\mathtt{x}} \triangleq s$ is used here to assign the name $\mathit{Var}_{\mathtt{x}}$ to the service s and to indicate that the name $\mathtt{x}$ occurs free in s. Thus, Var_n is a family of names, one for each name n identifying the variable service invocable by using the partner name n.

Now, suppose temporarily that standard variables, ranged over by X, Y,..., may occur in the syntax of COWS anywhere a variable can. We can remove them by using the following encodings. If ϵ contains standard variables $X_1, \ldots, X_n$, we can let

$$\begin{aligned} \langle\!\langle \epsilon \rangle\!\rangle_{\mathtt{m},\mathtt{n}} = [\mathtt{n}_1, \ldots, \mathtt{n}_k]\,(\,& \mathtt{x}_1 \bullet \mathit{read}!\langle \mathtt{n}_1\rangle \mid \cdots \mid \mathtt{x}_k \bullet \mathit{read}!\langle \mathtt{n}_k\rangle \\ & \mid [x_1, \ldots, x_k]\,(\,\mathtt{n}_1?\langle x_1\rangle.\ \cdots\ .\,\mathtt{n}_k?\langle x_k\rangle.\,\mathtt{m}!\langle \epsilon \cdot \{X_i \mapsto x_i\}_{i\in\{1,\ldots,k\}}, \mathtt{n}\rangle\,)\,) \end{aligned}$$

where $\{X_i \mapsto x_i\}$ denotes substitution of X_i with x_i, endpoint $\mathtt{m}$ returns the result of evaluating ϵ, and endpoint $\mathtt{n}$ permits to receive an acknowledgment when the resulting value is assigned to a service variable (of course, we are assuming that $\mathtt{m}$, $\mathtt{n}$, $\mathtt{n}_i$ and x_i are fresh). Basically, the encoded term reads the current values of the standard variables within ϵ (by means of partner names $\mathtt{x}_i$s associated to standard variables X_is) and uses them to evaluate ϵ. With this encoding of expression evaluation, the encoding of matching with assignment

becomes

$$\begin{aligned}\langle\!\langle [w = \epsilon].s \rangle\!\rangle &= [\mathtt{m}, \mathtt{n}]\,(\langle\!\langle \epsilon \rangle\!\rangle_{\mathtt{m},\mathtt{n}} \mid \mathtt{m}?\langle w, \mathtt{n}\rangle.\langle\!\langle s \rangle\!\rangle)\\ \langle\!\langle [X = \epsilon].s \rangle\!\rangle &= [\mathtt{n}]\,(\langle\!\langle \epsilon \rangle\!\rangle_{\mathtt{x}\bullet write,\mathtt{n}} \mid \mathtt{n}?\langle\rangle.\langle\!\langle s \rangle\!\rangle)\end{aligned} \tag{3.5}$$

where w is a value v or a variable x, while X is a standard variable. In the sequel, we will write $[\bar{w} = \bar{\epsilon}].s$, where $\bar{w} = \langle w_1, \ldots, w_k\rangle$ and $\bar{\epsilon} = \langle \epsilon_1, \ldots, \epsilon_k\rangle$, with $\bar{w}$ and $\bar{\epsilon}$ that may contain standard variables, for the sequence of assignments $[w_1 = \epsilon_1].\cdots.[w_k = \epsilon_k].s$. The encodings of the remaining constructs, where standard variables may directly occur, are

$$\begin{aligned}\langle\!\langle [X]\ s \rangle\!\rangle &= [\mathtt{x}]\,(\mathit{Var}_{\mathtt{x}} \mid \langle\!\langle s \rangle\!\rangle)\\ \langle\!\langle X \bullet u!\bar{\epsilon} \rangle\!\rangle &= [x, \mathtt{n}]\,(\,\mathtt{x}\bullet \mathit{read}!\langle \mathtt{n}\rangle \mid \mathtt{n}?\langle x\rangle.\,\langle\!\langle x \bullet u!\bar{\epsilon} \rangle\!\rangle\,)\\ \langle\!\langle u \bullet X!\bar{\epsilon} \rangle\!\rangle &= [x, \mathtt{n}]\,(\,\mathtt{x}\bullet \mathit{read}!\langle \mathtt{n}\rangle \mid \mathtt{n}?\langle x\rangle.\,\langle\!\langle u \bullet x!\bar{\epsilon} \rangle\!\rangle\,)\\ \langle\!\langle u \bullet u'!\langle \epsilon_1, \ldots, \epsilon_k\rangle \rangle\!\rangle &= [x_1, \mathtt{n}_1, \mathtt{m}_1, \ldots, x_k, \mathtt{n}_k, \mathtt{m}_k]\\ &\quad (\langle\!\langle \epsilon_1 \rangle\!\rangle_{\mathtt{n}_1,\mathtt{m}_1} \mid \ldots \mid \langle\!\langle \epsilon_k \rangle\!\rangle_{\mathtt{n}_k,\mathtt{m}_k}\\ &\qquad \mid \mathtt{n}_1?\langle x_1, \mathtt{m}_1\rangle.\cdots.\mathtt{n}_k?\langle x_k, \mathtt{m}_k\rangle.\,u \bullet u'!\langle x_1, \ldots, x_k\rangle\,)\\ &\quad \text{if } u \text{ and } u' \text{ do not contain standard variables}\\ \langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle &= [x_1, \ldots, x_k]\ p \bullet o?\bar{w}\cdot\{X_i \mapsto x_i\}_{i\in\{1,..,k\}}.\\ &\qquad [\mathtt{n}_1, \ldots, \mathtt{n}_k]\,(\,\mathtt{x}_1 \bullet \mathit{write}!\langle x_1, \mathtt{n}_1\rangle \mid \ldots \mid \mathtt{x}_k \bullet \mathit{write}!\langle x_k, \mathtt{n}_k\rangle\\ &\qquad\qquad \mid \mathtt{n}_1?\langle\rangle.\cdots.\mathtt{n}_k?\langle\rangle.\langle\!\langle s \rangle\!\rangle\,)\\ &\quad \text{if } \bar{w} \text{ contains standard variables } X_1, \ldots, X_k \text{ and}\\ &\qquad x_1, \ldots, x_k \text{ are fresh}\end{aligned} \tag{3.6}$$

This way, occurrences of standard variables can be completely removed. It is worth noticing that in the encoding of $p \bullet o?\bar{w}.s$, standard variables X_i occurring within $\bar{w}$ are replaced by auxiliary fresh variables x_i that are then used to update the corresponding standard variables. This means that standard variables are not used for correlation purposes. This choice is motivated by the fact that, differently from standard variables, correlation variables are write-once variables.

Sequential composition can be encoded alike in CCS [148, Chapter 8]. However, due to the asynchrony of invoke activities, the notion of well-termination must be relaxed w.r.t. CCS. Firstly, we settle that terms may indicate their termination by exploiting the invoke activity $x_{done} \bullet o_{done}!\langle\rangle$, where x_{done} is a distinguished variable and o_{done} is a distinguished name. Secondly, we say that a term s is *well-terminating* if, for every reduct s' of s and fresh partner p, $s' \cdot \{x_{done} \mapsto p\} \xrightarrow{p \bullet o_{done} \lhd \langle\rangle}$ implies that if $s' \cdot \{x_{done} \mapsto p\} \xrightarrow{\alpha}$ then $\alpha = p' \bullet o \lhd \bar{v}$, for some p', o and $\bar{v}$. Notably, well-termination does not demand a term to terminate, but only that whenever the term can perform activity $p \bullet o_{done}!\langle\rangle$, then it terminates except for, possibly, some parallel pending invoke activities. As usual, the encoding of sequential composition relies on the assumption that all calculus operators (in particular, parallel composition) can be rendered as to preserve well-termination. Therefore, if

we only consider well-terminating terms, then, for a fresh p, we can let:

$$\langle\!\langle s_1; s_2 \rangle\!\rangle \;=\; [p]\,(\,\langle\!\langle s_1 \cdot \{x_{done} \mapsto p\} \rangle\!\rangle \mid p \bullet o_{done}?\langle\rangle.\,\langle\!\langle s_2 \rangle\!\rangle\,) \tag{3.7}$$

Finally, iterative constructs can be encoded by exploiting the previous encodings:

$$\langle\!\langle \textbf{while}\ (\epsilon)\ \{s\} \rangle\!\rangle \;=\; [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.\,\textbf{if}\ (\epsilon)\ \textbf{then}\ \{\langle\!\langle s \rangle\!\rangle; \mathtt{n}!\langle\rangle\}\ \textbf{else}\ \{x_{done} \bullet o_{done}!\langle\rangle\}\,) \tag{3.8}$$

Services' execution modalities. In the previous examples and in that presented in Section 3.1 we have shown that persistent services create one specific instance to serve each received request. Once created, service instances can be executed concurrently or sequentially, and may also share (part of) the state. The following examples illustrate how different services' execution modalities, e.g. concurrent vs. sequential execution, local vs. shared state, can be modelled in μCOWS^m (see [104] for an account of this topic in another process language for SOC).

In μCOWS^m, a service can be modelled by a term of the form $*\,[\bar{u}\,]\,s$, where tuple $\bar{u}$ contains all the free variables of s. The use of replication enables providing as many concurrent instances as needed, while that of delimitation permits modelling the state (by restricting the scope of variables). This means that the previous term corresponds to a service whose instances are *concurrently executed without a shared state*. For instance, consider the following service definition:

$$*\,[x_1, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s$$

If we put it in parallel with the invocation $p \bullet o!\langle v_1 \rangle$, the resulting system can evolve as follows:

$$\begin{array}{l} *\,[x_1, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s \;\mid\; p \bullet o!\langle v_1 \rangle \;\xrightarrow{\emptyset}\\ *\,[x_1, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s \;\mid\; [x_2, \ldots, x_n]\; s \cdot \{x_1 \mapsto v_1\} \end{array}$$

Each time an invocation is processed, a new service instance with private variables $x_2, \ldots, x_n$ is activated. For example, if we have two concurrent invocations, we get

$$\begin{array}{l} *\,[x_1, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s \;\mid\; p \bullet o!\langle v_1 \rangle \;\mid\; p \bullet o!\langle v_2 \rangle \;\xrightarrow{\emptyset}\; \xrightarrow{\emptyset}\\ *\,[x_1, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s \;\mid\; [x_2, \ldots, x_n]\; s \cdot \{x_1 \mapsto v_1\} \;\mid\; [x_2, \ldots, x_n]\; s \cdot \{x_1 \mapsto v_2\} \end{array}$$

The resulting system is composed of the service definition and of two different instances, each with its own state.

To allow instances of a same service to be *concurrently executed while sharing (part of) the state*, we move the delimitations of the variables to be shared outside the scope of replication. Thus, if $x_1, \ldots, x_k$ are shared and $x_{k+1}, \ldots, x_n$ are not, the previous example can be modified as follows:

$$[x_1, \ldots, x_k]\; *\; [x_{k+1}, \ldots, x_n]\; p \bullet o?\langle x_1 \rangle.s$$

After a parallel request $p \bullet o!\langle v_1 \rangle$ has been processed, we have:

$$[x_2, \ldots, x_k]\,(\,*\,[x_{k+1}, \ldots, x_n]\,p \bullet o?\langle v_1 \rangle.s \cdot \{x_1 \mapsto v_1\} \mid [x_{k+1}, \ldots, x_n]\,s \cdot \{x_1 \mapsto v_1\}\,)$$

In this case, since x_1 is shared both by the service definition and by its instances, new instances can be created only if the service definition receives requests along $p \bullet o$ with the same value (i.e. v_1) as the first invocation. In general, however, instantiation variables, such as x_1, are not shared, in order to allow service invocations with different arguments to trigger instance creation. To model this behaviour, we can simply leave instantiation variables within the scope of replication. Consider for example the term:

$$[x_2]\ *\ [x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s$$

If requests $p \bullet o!\langle v_1 \rangle$ and $p \bullet o!\langle v_2 \rangle$ are put in parallel, the resulting system can evolve as follows:

$$\begin{array}{l} [x_2]\ *\ [x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s \mid p \bullet o!\langle v_1 \rangle \mid p \bullet o!\langle v_2 \rangle \xrightarrow{\emptyset} \xrightarrow{\emptyset} \\ [x_2]\,(\,*\,[x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s \mid [x_3]\,s \cdot \{x_1 \mapsto v_1\} \mid [x_3]\,s \cdot \{x_1 \mapsto v_2\}\,) \end{array}$$

After two computational steps, two instances, each with a local state (i.e. the variable x_3) and sharing variable x_2, are activated.

Suppose now we want to model the fact that service instances can only be *sequentially executed without sharing a state*. We can exploit the sequential operator ';' (see encoding (3.7)) and a fresh endpoint $\mathtt{n}$ (to signal termination of an instance). For example, consider the term:

$$[\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.\,[x_1, \ldots, x_k]\,(\,(p \bullet o?\langle x_1 \rangle.s)\,;\,\mathtt{n}!\langle\rangle\,)\,)$$

After processing a parallel request $p \bullet o!\langle v_1 \rangle$, the resulting system becomes

$$[\mathtt{n}]\,(\,*\,\mathtt{n}?\langle\rangle.\,[x_1, \ldots, x_k]\,(\,(p \bullet o?\langle x_1 \rangle.s)\,;\,\mathtt{n}!\langle\rangle\,) \mid [x_2, \ldots, x_k]\,(s \cdot \{x_1 \mapsto v_1\}\,;\,\mathtt{n}!\langle\rangle)\,)$$

Now, another request cannot be processed, and creation of a new service instance is disabled, until the existing instance emits the termination signal $\mathtt{n}!\langle\rangle$. This guarantees that at most one service instance is executed at a time.

Finally, by combining all the previous patterns, we can also model services whose instances are *sequentially executed and share (part of) a state*, as the following term shows:

$$[\mathtt{n}, x_2]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.\,[x_1, x_3]\,(\,(p \bullet o?\langle x_1 \rangle.s)\,;\,\mathtt{n}!\langle\rangle\,)\,)$$

Flow graphs. In business process management, flow graphs[3] provide a direct and intuitive way to structure workflow processes, where activities executed in parallel can be synchronized by settling dependencies, called (flow) links, among them. At the beginning

[3]Here, we refer to the corresponding notion of WS-BPEL rather than to similar synchronization constructs of some process calculi (see e.g. [125]) or to the homonymous graphical notation used for representing processes and their interconnection structure (see, e.g., [148, 150]).

$$
\begin{array}{rcll}
s & ::= & \ldots \mid [\overline{fl}\,]\,ls \mid \sum_{i\in I} p_i \bullet o_i?\bar{w}_i.s_i & \text{(services)} \\
ls & ::= & (jc) \overset{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{\epsilon}) \mid s \Rightarrow (\overline{fl}, \bar{\epsilon}) \mid ls \mid ls & \text{(linked services)} \\
jc & ::= & \mathbf{true} \mid \mathbf{false} \mid fl \mid \neg jc \mid jc \vee jc \mid jc \wedge jc & \text{(join conditions)} \\
sjf & ::= & yes \mid no & \text{(supp. join failure)}
\end{array}
$$

$$
\langle\!\langle [\overline{fl}\,]\,ls \rangle\!\rangle = [\bar{x}_{fl}]\,\langle\!\langle ls \rangle\!\rangle \qquad \langle\!\langle ls_1 \mid ls_2 \rangle\!\rangle = \langle\!\langle ls_1 \rangle\!\rangle \mid \langle\!\langle ls_2 \rangle\!\rangle \qquad \langle\!\langle s \Rightarrow (\overline{fl}, \bar{\epsilon}) \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle; [\bar{x}_{fl} = \bar{\epsilon}]
$$

$$
\langle\!\langle (jc) \overset{yes}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{\epsilon}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\, \langle\!\langle s \rangle\!\rangle; [\bar{x}_{fl} = \bar{\epsilon}] \,\}\ \mathbf{else}\ \{\, [outLinkOf(s) = \overline{\mathbf{false}}] \,\}
$$

$$
\langle\!\langle (jc) \overset{no}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{\epsilon}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\, \langle\!\langle s \rangle\!\rangle; [\bar{x}_{fl} = \bar{\epsilon}] \,\}\ \mathbf{else}\ \{\, \mathbf{throw}(\phi_{join\_f}) \,\}
$$

$$
\begin{aligned}
\langle\!\langle \textstyle\sum_{i\in\{1..n\}} p_i \bullet o_i?\bar{w}_i.s_i \rangle\!\rangle = {} & p_1 \bullet o_1?\bar{w}_1.\,[\textstyle\bigcup_{j\in\{2..n\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\,\langle\!\langle s_1 \rangle\!\rangle \\
& + \ldots + p_n \bullet o_n?\bar{w}_n.\,[\textstyle\bigcup_{j\in\{1..n-1\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\,\langle\!\langle s_n \rangle\!\rangle
\end{aligned}
$$

Table 3.5: Syntax and encoding of flow graphs

of a parallel execution, all involved links are inactive and only those activities with no synchronization dependencies can execute. Once all incoming links of an activity are active (i.e., they have been assigned either a positive or negative state), a guard, called *join condition*, is evaluated. When an activity terminates, the status of the outgoing links, which can be positive, negative or undefined, is determined through evaluation of a *transition condition*. When an activity in the flow graph cannot execute (i.e., the join condition fails), a *join failure* fault is emitted to signal that some activities have not completed. An attribute called 'suppress join failure' can be set to *yes* to ensure that join condition failures do not throw the join failure fault (this way obtaining the so-called *Dead-Path Elimination* effect [166]).

To express the constructs above, we extend the syntax of μCOWS^m as illustrated in the upper part of Table 3.5. A *flow graph activity* $[\overline{fl}\,]\,ls$ is a delimited *linked service*, where the activities within ls can synchronize by means of the flow links in $\overline{fl}$, rendered as (boolean) variables. A linked service is a service equipped with a set of incoming flow links that forms the *join condition*, and a set of outgoing flow links that represents the *transition condition*. Incoming flow links and join condition are denoted by $(jc) \overset{sjf}{\Rightarrow}$. Outgoing links are represented by $\Rightarrow (\overline{fl}_{i\in I}, \bar{\epsilon}_{i\in I})$ where each pair (fl_i, ϵ_i) is composed of a flow link fl_i and the corresponding transition (boolean) condition ϵ_i. Attribute sjf permits suppressing possible join failures. Input-guarded summation replaces binary choice, because we want all the branches of a multiple choice to be considered at once.

Again, we show that in fact it is not necessary to extend the syntax because flow graphs can be easily encoded by exploiting the capability of μCOWS^m of modelling a state shared among a group of activities. The most interesting cases of the encoding are shown in the lower part of Table 3.5. The encoding uses the auxiliary function $outLinkOf(s)$, that

returns the tuple of outgoing links in s and is inductively defined as follows:

$$
\begin{array}{l}
\mathit{outLinkOf}([\overline{fl}]\, ls) = \mathit{outLinkOf}(ls) \\
\mathit{outLinkOf}(\sum_{i\in\{1..n\}} p_i \bullet o_i?\bar{w}_i.s_i) = \mathit{outLinkOf}(s_1), \ldots, \mathit{outLinkOf}(s_n) \\
\mathit{outLinkOf}((jc) \stackrel{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{\epsilon})) = \mathit{outLinkOf}(s)\,,\ \bar{x}_{fl} \\
\mathit{outLinkOf}(s \Rightarrow (\overline{fl}, \bar{\epsilon})) = \mathit{outLinkOf}(s)\,,\ \bar{x}_{fl} \\
\mathit{outLinkOf}(ls_1 \mid ls_2) = \mathit{outLinkOf}(ls_1)\,,\ \mathit{outLinkOf}(ls_2) \\
\mathit{outLinkOf}(\mathbf{0}) = \mathit{outLinkOf}(u \bullet u'!\bar{\epsilon}) = \langle\rangle \\
\mathit{outLinkOf}(s_1 \mid s_2) = \mathit{outLinkOf}(s_1)\,,\ \mathit{outLinkOf}(s_2) \\
\mathit{outLinkOf}([u]\, s) = \mathit{outLinkOf}(*\, s) = \mathit{outLinkOf}(s)
\end{array}
$$

Basically, flow graphs are rendered as delimited services, while flow links are rendered as variables. A join condition is encoded as a boolean condition within a conditional construct, where the transition conditions are rendered as the assignment $[\bar{x}_{fl} = \bar{\epsilon}]$. In case attribute 'suppress join failure' is set to *no*, a join condition failure produces a fault signal that can be caught by a proper fault handler (see Section 3.2.3.3 for an account of fault handling with COWS). Choice among (linked) services is implemented in such a way that, when a branch is selected, the links outgoing from the activities of the discarded branches are set to **false**. The same rationale underlies the new encoding of conditional choice that becomes as follows

$$
\begin{array}{rcl}
\langle\!\langle \textbf{if } (\epsilon) \textbf{ then } \{s_1\} \textbf{ else } \{s_2\} \rangle\!\rangle & = & \textbf{if } (\epsilon) \textbf{ then } \{\, [\mathit{outLinkOf}(s_2) = \overline{\textbf{false}}].\, \langle\!\langle s_1 \rangle\!\rangle \,\} \\
& & \textbf{else } \{\, [\mathit{outLinkOf}(s_1) = \overline{\textbf{false}}].\, \langle\!\langle s_2 \rangle\!\rangle \,\}
\end{array}
$$

3.2.2 μCOWS : the protection- and kill-free fragment of COWS

The fragment of COWS presented in this section, namely μCOWS, dispenses with those activities dealing with termination, i.e. kill and protection. In other words, μCOWS extends μCOWS^m with priority in the parallel composition.

3.2.2.1 Syntax and operational semantics

The syntax of μCOWS and the set of laws defining the structural congruence coincide with that of μCOWS^m, shown in Tables 3.1 and 3.2, respectively. Instead, the labelled transition relation $\xrightarrow{\alpha}$ is the least relation over μCOWS services induced by the rules in Table 3.6. The rules in the upper part of the table are directly borrowed from μCOWS^m (Table 3.4), while those in the lower part are the new rules used to deal with priority in the parallel composition. Labels are now generated by the following grammar:

$$
\alpha \quad ::= \quad \mathtt{n} \lhd \bar{v} \quad | \quad \mathtt{n} \rhd \bar{w} \quad | \quad \mathtt{n}\, \sigma\, \ell\, \bar{v}
$$

The new label $\mathtt{n}\,\sigma\,\ell\,\bar{v}$, which replaces the label σ, denotes execution of a communication over $\mathtt{n}$ with matching values $\bar{v}$, generated substitution having ℓ pairs, and substitution σ to be still applied. Now, *computational steps* are denoted by label of the form

$$\frac{[\![\bar{\epsilon}]\!] = \bar{v}}{\mathrm{n}!\bar{\epsilon} \xrightarrow{\mathrm{n} \lhd \bar{v}} \mathbf{0}}\ (inv) \qquad \mathrm{n}?\bar{w}.s \xrightarrow{\mathrm{n} \rhd \bar{w}} s\ (rec) \qquad \frac{g \xrightarrow{\alpha} s}{g + g' \xrightarrow{\alpha} s}\ (choice)$$

$$\frac{s \xrightarrow{\alpha} s' \quad u \notin \mathrm{u}(\alpha)}{[u]\, s \xrightarrow{\alpha} [u]\, s'}\ (del) \qquad \frac{s \equiv \xrightarrow{\alpha} \equiv s'}{s \xrightarrow{\alpha} s'}\ (str)$$

$$\frac{s_1 \xrightarrow{\mathrm{n} \rhd \bar{w}} s_1' \quad s_2 \xrightarrow{\mathrm{n} \lhd \bar{v}} s_2' \quad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \quad \mathrm{noConf}(s_1 \mid s_2, \mathrm{n}, \bar{v}, |\sigma|)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma|\sigma|\bar{v}} s_1' \mid s_2'}\ (com_2)$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \qquad \alpha \neq \mathrm{n}\sigma\ell\bar{v}}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (par_2) \qquad \frac{s_1 \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s_1' \qquad \mathrm{noConf}(s_2, \mathrm{n}, \bar{v}, \ell)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s_1' \mid s_2}\ (par_{com})$$

$$\frac{s \xrightarrow{\mathrm{n}\sigma \uplus \{x \mapsto v\}\ell\bar{v}} s'}{[x]\, s \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s' \cdot \{x \mapsto v\}}\ (del_{com2})$$

Table 3.6: μCOWS operational semantics

$\mathrm{n}\emptyset\ell\bar{v}$. Notation $\mathrm{u}(\alpha)$, indicating the set of names and variables occurring in α, is such that $\mathrm{u}(\mathrm{n}\sigma\ell\bar{v}) = \mathrm{u}(\sigma)$.

The definition of the labelled transition relation exploits a new auxiliary predicate $\mathrm{noConf}(s, \mathrm{n}, \bar{v}, \ell)$, with ℓ natural number. The predicate, defined in Table 3.7, holds true if s cannot immediately perform a receive over the endpoint n matching $\bar{v}$ and generating a substitution σ with $|\sigma| < \ell$.

We comment on salient points. In μCOWS, the communication label $\mathrm{n}\sigma\ell\bar{v}$, produced by rule *($com_2$)*, carries information about the communication that has taken place (i.e. the endpoint, the transmitted values, the generated substitution and its length) used to check the presence of conflicting receives in parallel components. Indeed, if more than one matching is possible, the receive that needs fewer substitutions is selected to progress (rules *(com_2)* and *(par_{com})*). This mechanism permits to correlate different service communications thus implicitly creating interaction sessions and can be exploited to model the precedence of a service instance over the corresponding service specification when both can process the same request (see Section 3.2.2.2 for some examples). Rule *(del_{com2})* is similar to *(del_{com})* (shown in Table 3.4) but deals with labels generated by communications subject to priority. Notably, during the inference of a transition labelled by $\mathrm{n}\sigma\ell\bar{v}$, the length of the substitution to be applied decreases (rule *(del_{com2})*), while the length ℓ of the initial substitution does never change, which makes it suitable to check, in any moment,

$$
\begin{array}{l}
\mathrm{noConf}(\mathtt{u}!\bar{\epsilon}, \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(\mathbf{0}, \mathtt{n}, \bar{v}, \ell) \;=\; \mathbf{true} \\[2ex]
\mathrm{noConf}(\mathtt{n}'?\bar{w}.s, \mathtt{n}, \bar{v}, \ell) \;=\; \begin{cases} \mathbf{false} & \text{if } \mathtt{n}' = \mathtt{n} \wedge\ |\mathcal{M}(\bar{w}, \bar{v})| < \ell \\ \mathbf{true} & \text{otherwise} \end{cases} \\[2ex]
\mathrm{noConf}(g + g', \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(g, \mathtt{n}, \bar{v}, \ell) \;\wedge\; \mathrm{noConf}(g', \mathtt{n}, \bar{v}, \ell) \\[2ex]
\mathrm{noConf}(s \mid s', \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) \;\wedge\; \mathrm{noConf}(s', \mathtt{n}, \bar{v}, \ell) \\[2ex]
\mathrm{noConf}([u]\, s, \mathtt{n}, \bar{v}, \ell) \;=\; \begin{cases} \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) & \text{if } u \notin \mathtt{n} \\ \mathbf{true} & \text{otherwise} \end{cases} \\[2ex]
\mathrm{noConf}(* \, s, \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell)
\end{array}
$$

Table 3.7: There are not conflicting receives along $\mathtt{n}$ matching $\bar{v}$

existence of better matching, i.e. of parallel receives with greater priority. Execution of parallel services is interleaved (rule *(par$_2$)*), but when a communication is performed. In such case, the progress of the receive activity with greater priority must be ensured.

3.2.2.2 Examples

We present now some examples that point out the peculiarities of μCOWS.

Multiple start activities. Web services could be able of receiving multiple messages in a statically unpredictable order and in such a way that the first incoming message triggers creation of a service instance which subsequent messages are routed to. This would require all those receive activities that can be immediately executed (according to [166], Section 16.3, there are *multiple start activities*) to share a non-empty set of variables (the so-called *correlation set*).

Consider, for example, an excerpt of the high-level specification of the Morra game scenario presented in Section 3.1:

$$
\begin{array}{l}
(\, odds \bullet throw!\langle first, cbA, 2\rangle \;\mid\; [x_A]\, cbA \bullet res?\langle first, x_A\rangle.\; s_A \,) \\
\mid\; (\, evens \bullet throw!\langle first, cbB, 1\rangle \;\mid\; [x_B]\, cbB \bullet res?\langle first, x_B\rangle.\; s_B \,) \\
\mid\; * \, [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
\qquad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \;\mid\; evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\qquad\quad \mid\; x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \;\mid\; y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle \,)
\end{array}
$$

After an interaction with the client B, an instance running in parallel with the service

definition is created.

$$
\begin{array}{l}
(\, odds \bullet throw!\langle first, cbA, 2\rangle \mid [x_A]\, cbA \bullet res?\langle first, x_A\rangle.\ s_A \,) \\
\mid (\, [x_B]\, cbB \bullet res?\langle first, x_B\rangle.\ s_B \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
\quad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\quad\quad \mid x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle \,) \\
\mid [x_p, x_{num}] \\
\quad (\, odds \bullet throw?\langle first, x_p, x_{num}\rangle \\
\quad\quad \mid x_p \bullet res!\langle first, win(x_{num}, 1, 1)\rangle \mid cbB \bullet res!\langle first, win(x_{num}, 1, 0)\rangle \,)
\end{array}
$$

Now, the service definition and the created instance, being both able to receive the same tuple $\langle first, cbA, 2\rangle$ along the endpoint $odds \cdot throw$, compete for the request $odds \bullet throw!\langle first, cbA, 2\rangle$ (i.e., in WS-BPEL jargon, two *conflicting* receive activities are enabled). However, μCOWS's (prioritized) semantics, in particular rule *(com$_2$)* in combination with rule *(par$_{com}$)*, allows only the existing instance to evolve. Indeed, suppose to try to infer the transition corresponding to the interaction between client A and the service definition. Then, the generated substitution would have length 3 and, hence, the predicate $\mathrm{noConf}(s_{inst}, odds \bullet throw, \langle first, cbA, 2\rangle, 3)$, where s_{inst} is the created instance, would not hold. In fact, the instance can perform a receive matching the same message and producing a substitution with fewer pairs (it has length 2). This way, the creation of a new instance is prevented and the only feasible computation leads to the following term:

$$
\begin{array}{l}
(\, [x_A]\, cbA \bullet res?\langle first, x_A\rangle.\ s_A \,) \\
\mid (\, [x_B]\, cbB \bullet res?\langle first, x_B\rangle.\ s_B \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
\quad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\quad\quad \mid x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle \,) \\
\mid (\, cbA \bullet res!\langle first, win(2, 1, 1)\rangle \mid cbB \bullet res!\langle first, win(2, 1, 0)\rangle \,)
\end{array}
$$

At the end, the result of the match is sent to both players and the term evolves to:

$$
\begin{array}{l}
(\, s_A \cdot \{x_A \mapsto w\} \,) \\
\mid (\, s_B \cdot \{x_B \mapsto l\} \,) \\
\mid *\, [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
\quad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\quad\quad \mid x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle \,)
\end{array}
$$

It is worth noticing that the above considerations still hold if we use choice rather than parallel to compose the start activities of the Morra service, as shown below:

$$
\begin{array}{l}
*\, [x_{id}, x_p, x_{num}, y_p, y_{num}] \\
\quad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle.\ evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\quad\quad +\ evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle.\ odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \,)\ ;\ \ldots
\end{array}
$$

No conflict predicate. Rules *(com$_2$)* and *(par$_{com}$)* use the *no conflict* predicate noConf$(_, \mathtt{n}, \bar{v}, \ell)$ for checking the presence of concurrent conflicting receives. When these rules must be used to infer a transition, a preventive α-conversion may be necessary. Indeed, condition noConf$(\mathtt{n}?\bar{w}.s, \mathtt{n}, \bar{v}, \ell)$ might single out patterns that could not really match the transmitted values. These false alarms would block the inference (but allow us to stay on the 'safe' side).

For instance, consider the following term:

$$\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle \mid [m]\,\mathtt{n}?\langle m\rangle \tag{3.9}$$

Apparently, both receive activities match the invoke activity, but only $\mathtt{n}?\langle x\rangle$ can synchronise with $\mathtt{n}!\langle m\rangle$, because the argument of $\mathtt{n}?\langle m\rangle$ is a restricted name, thus it is certainly different from the name transmitted by the invoke. However, if we try to naively infer the transition corresponding to the synchronisation between $\mathtt{n}!\langle m\rangle$ and $\mathtt{n}?\langle x\rangle$, we fail due to rules *(com$_2$)* or *(par$_{com}$)*. In fact, noConf$([m]\,\mathtt{n}?\langle m\rangle, \mathtt{n}, \langle m\rangle, 1)$ does not hold because $\mathcal{M}(m, m)$ produces the substitution $\emptyset$, that is smaller than $\{x \mapsto m\}$, that is produced by $\mathcal{M}(x, m)$.

However, the wanted transition can be inferred by first applying α-conversion. In fact, (3.9) can be re-written as follows:

$$\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle \mid [m']\,\mathtt{n}?\langle m'\rangle$$

Now, it is clear that $\mathtt{n}?\langle m'\rangle$ is not a conflicting receive, because $\mathcal{M}(m', m)$ is undefined.

The same observations hold for the term:

$$[m]\,(\,\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle\,) \mid \mathtt{n}?\langle m\rangle$$

Again, α-conversion is necessary for inferring the correct transitions. Instead, if in (3.9) we replace delimitation of m with that of $\mathtt{n}$, the correct transition can be directly inferred because noConf$([\mathtt{n}]\,\mathtt{n}?\langle m'\rangle, \mathtt{n}, \bar{v}, \ell)$ holds **true**.

Default behaviour. The priority mechanism of μCOWS can be used for coordination, other than for orchestration, purposes. For example, in the service *5F* of Section 3.1 reported below

$$\begin{aligned}[x, y, z]\,(\,& m \bullet \mathit{req5f}?\langle x, y, z\rangle.\, m \bullet \mathit{resp5f}!\langle x, \mathit{err}, \mathit{err}\rangle \\ &+\ m \bullet \mathit{req5f}?\langle x, 0, 0\rangle.\, m \bullet \mathit{resp5f}!\langle x, l, w\rangle \\ &+\ m \bullet \mathit{req5f}?\langle x, 0, 1\rangle.\, m \bullet \mathit{resp5f}!\langle x, w, l\rangle \\ &+\ \ldots\ +\ m \bullet \mathit{req5f}?\langle x, 5, 5\rangle.\, m \bullet \mathit{resp5f}!\langle x, l, w\rangle\,)\end{aligned}$$

the priority mechanism enables implementing a sort of 'default' behaviour. Indeed, when the service is invoked along the endpoint $m \bullet \mathit{req5f}$ with a correct tuple of values, i.e. the second and third elements are integers between 0 and 5, a tuple containing the strings w and l is returned along $m \bullet \mathit{resp5f}$. For example, if *5F* is invoked by $m \bullet \mathit{req5f}!\langle \mathit{id}, 0, 0\rangle$, although the invocation and the receive $m \bullet \mathit{req5f}?\langle x, y, z\rangle$ do match, the priority mechanism

ensures that the service replies with $m \bullet resp5f!\langle id, l, w\rangle$. The above receive can progress only when a throw is not admissible, e.g. $m \bullet req5f!\langle id, 0, 7\rangle$, and allows the service to reply with a tuple containing two string *err*.

Implementing sessions by means of correlation mechanisms. In a service-oriented scenario, service definitions are used as templates for creating service instances that deliver application functionality to either end-user applications or other instances. Sessions and correlation are used in this scenario to allow service instances to have conversations by means of different communication mechanisms. Session-based communication takes place along *private* channels and is, usually, regulated by session types [112, 207]. Correlation, instead, is a mechanism that simply permits delivering messages to the proper service instances by means of their same contents.

Here, we show that correlation can be used to suitably implement sessions. To this aim, suppose to temporarily extend μCOWS syntax with session constructs inspired by those introduced in [112] as shown in the upper part of Table 3.8, where c, c', ... denote *channels* and l, l_1, ... denote *branching labels*. The initiation of a session is requested, via a name a, by a construct **request** $a(c)$ **in** s, and causes the generation of a fresh channel c that shall be used by s for later communications. Conversely, **accept** $a(c)$ **in** s permits receiving the request for the initiation of a session via a and generating a new channel c, which shall be used for communications in s. Activities $c![\bar{\epsilon}]$ and $c?(\bar{w})$ **in** s denote (asynchronous) data sending and receiving via a channel c of a session, respectively. $c \lhd l$ and $c \rhd \{l_1 : s_1 \parallel \ldots \parallel l_r : s_r\}$ denote label selection and label branching (where l_1,..., l_r are assumed to be pairwise distinct) via a channel c, respectively. They mime method invocation in object-based programming. Finally, **throw** $c[c']$ and **catch** $c(c')$ **in** s denote session channel sending and receiving, and permit to pass a channel, that is being used in a session, to another process (this feature is called 'delegation'), thus allowing complex nested structured communications. Constructs **request** $a(c)$ **in** s, **accept** $a(c)$ **in** s, and **catch** $c'(c)$ **in** s bind channel c in s.

Now we show that, by exploiting private names as correlation data, the introduced session constructs can be easily encoded in μCOWS, thus it is not necessary to extend its syntax. This means that session-based communication can be implemented by using correlation mechanisms. The most interesting cases of the encoding are reported in the lower part of Table 3.8 (in the remaining cases, the encoding acts as an homomorphism). The distinguished operation names o_{init} and o_{start} are used for initialising and starting sessions, respectively, while the distinguished endpoints `req` and `acc` are used for modelling the two sides of sessions, i.e. the requestor- and acceptor-side, respectively. For simplicity sake, we assume that bound channels are pairwise distinct and different from the free ones (as usual, this condition is not restrictive and can always be fulfilled by possibly using α-conversion).

The encoding function $\langle\!\langle \cdot \rangle\!\rangle_C$, that is inductively defined on the extended syntax, is parameterized by a set of channels C. The encoding of an extended service s is given

$$
\begin{array}{rcll}
s & ::= & \ldots & \text{(services)} \\
& | & \textbf{request}\ a(c)\ \textbf{in}\ s & \text{(session request)} \\
& | & \textbf{accept}\ a(c)\ \textbf{in}\ s & \text{(session acceptance)} \\
& | & c![\bar{\epsilon}] & \text{(data sending)} \\
& | & c?(\bar{w})\ \textbf{in}\ s & \text{(data reception)} \\
& | & c \lhd l & \text{(label selection)} \\
& | & c \rhd \{l_1 : s_1 \parallel \ldots \parallel l_r : s_r\} & \text{(label branching)} \\
& | & \textbf{throw}\ c[c'] & \text{(channel sending)} \\
& | & \textbf{catch}\ c(c')\ \textbf{in}\ s & \text{(channel reception)}
\end{array}
$$

$$\langle\!\langle \textbf{request}\ a(c)\ \textbf{in}\ s \rangle\!\rangle_C = [c]\,(\, a \bullet o_{init}!\langle c\rangle \mid a \bullet o_{start}?\langle c\rangle.\langle\!\langle s\rangle\!\rangle_C\,)$$

$$\langle\!\langle \textbf{accept}\ a(c)\ \textbf{in}\ s \rangle\!\rangle_C = [x_c]\,(\, a \bullet o_{init}?\langle x_c\rangle.(\, a \bullet o_{start}!\langle x_c\rangle \mid \langle\!\langle s\rangle\!\rangle_{C\cup\{c\}}\,)\,)$$

$$\langle\!\langle c![\bar{\epsilon}]\rangle\!\rangle_C = \begin{cases} \texttt{req}!(x_c, \bar{\epsilon}) & \text{if } c \in C \\ \texttt{acc}!(c, \bar{\epsilon}) & \text{otherwise} \end{cases} \qquad \langle\!\langle c?(\bar{w})\ \textbf{in}\ s\rangle\!\rangle_C = \begin{cases} \texttt{acc}?(x_c, \bar{w}).\langle\!\langle s\rangle\!\rangle_C & \text{if } c \in C \\ \texttt{req}?(c, \bar{w}).\langle\!\langle s\rangle\!\rangle_C & \text{otherwise} \end{cases}$$

$$\langle\!\langle c \lhd l\rangle\!\rangle_C = \begin{cases} \texttt{req}!\langle x_c, l\rangle & \text{if } c \in C \\ \texttt{acc}!\langle c, l\rangle & \text{otherwise} \end{cases}$$

$$\langle\!\langle c \rhd \{l_1 : s_1 \| \ldots \| l_r : s_r\}\rangle\!\rangle_C = \begin{cases} \texttt{acc}?\langle x_c, l_1\rangle.\langle\!\langle s_1\rangle\!\rangle_C + \ldots + \texttt{acc}?\langle x_c, l_r\rangle.\langle\!\langle s_r\rangle\!\rangle_C & \text{if } c \in C \\ \texttt{req}?\langle c, l_1\rangle.\langle\!\langle s_1\rangle\!\rangle_C + \ldots + \texttt{req}?\langle c, l_r\rangle.\langle\!\langle s_r\rangle\!\rangle_C & \text{otherwise} \end{cases}$$

$$\langle\!\langle \textbf{throw}\ c[c']\rangle\!\rangle_C = \begin{cases} \texttt{req}!\langle x_c, \langle\!\langle c'\rangle\!\rangle_C\rangle & \text{if } c \in C \\ \texttt{acc}!\langle c, \langle\!\langle c'\rangle\!\rangle_C\rangle & \text{otherwise} \end{cases}$$

$$\langle\!\langle \textbf{catch}\ c(c')\ \textbf{in}\ s\rangle\!\rangle_C = \begin{cases} \texttt{acc}?\langle x_c, \langle\!\langle c'\rangle\!\rangle_C\rangle.\langle\!\langle s\rangle\!\rangle_C & \text{if } c \in C \\ \texttt{req}?\langle c, \langle\!\langle c'\rangle\!\rangle_C\rangle.\langle\!\langle s\rangle\!\rangle_C & \text{otherwise} \end{cases} \qquad \begin{array}{l} \langle\!\langle c\rangle\!\rangle_{C\cup\{c\}} = x_c \\ \langle\!\langle c\rangle\!\rangle_C = c \quad \text{if } c \notin C \end{array}$$

Table 3.8: Syntax and encoding of session constructs

by a μCOWS service $\langle\!\langle s\rangle\!\rangle_C$ with $C = \emptyset$; as the encoding proceeds, C is used to record the channels that were initially bound by a session acceptance. In fact, the crux of the encoding is mapping each session request in a μCOWS term that generates a fresh name (by means of delimitation operator) and sends it to the encoding of the corresponding session acceptance. Then, data sending and reception, label selection and branching, and channel sending and reception use the new private name as first element of messages for correlation purposes. A subtle point of the encoding is that whenever a session channel is bound by a session acceptance, then it is mapped to a μCOWS variable, otherwise it is mapped to a μCOWS name. Moreover, in the former case reception takes place along `acc` and sending along `req`, and vice versa in the latter case; this way, internal synchronization within a single side of a session is avoided.

In the above encoding, if acc and req are not distinguished (and reserved) endpoints, we could write malicious services that can steal messages exchanged over a session and use the correlation datum to pass themselves off as one of the two session parties. For example, consider an already initialised session where the requestor party can perform the activities $c![5, \text{“}foo\text{”}] \mid s$ and the acceptor the activity $c?(x, y)$ **in** $c \lhd l_{ok}$. The corresponding encoded term is as follows:

$$[c]\,(\,(\mathtt{acc}!\langle c, 5, \text{“}foo\text{”}\rangle \mid \langle\!\langle s \rangle\!\rangle_{\emptyset}) \;\mid\; [x, y]\,\mathtt{acc}?\langle c, x, y\rangle.\,\mathtt{req}!\langle c, l_{ok}\rangle\,) \qquad (*)$$

Now, if we simply put in parallel the above term with $[z_1, z_2, z_3]\,\mathtt{acc}?\langle z_1, z_2, z_3\rangle.s'$, that uses improperly channel acc, i.e.

$$\begin{array}{l}[c]\,(\,(\mathtt{acc}!\langle c, 5, \text{“}foo\text{”}\rangle \mid \langle\!\langle s \rangle\!\rangle_{\emptyset}) \;\mid\; [x, y]\,\mathtt{acc}?\langle c, x, y\rangle.\,\mathtt{req}!\langle c, l_{ok}\rangle\,) \\ \mid\; [z_1, z_2, z_3]\,\mathtt{acc}?\langle z_1, z_2, z_3\rangle.s'\end{array}$$

due to the prioritised semantics of parallel composition, the communication takes place in the proper way and the term correctly evolves to

$$[c]\,(\,\langle\!\langle s \rangle\!\rangle_{\emptyset} \;\mid\; \mathtt{req}!\langle c, l_{ok}\rangle\,) \;\mid\; [z_1, z_2, z_3]\,\mathtt{acc}?\langle z_1, z_2, z_3\rangle.s'$$

By the way, in μCOWS^m, since parallel composition does not have a prioritised semantics, the receive $\mathtt{acc}?\langle z_1, z_2, z_3\rangle$ can steal the message by synchronising with $\mathtt{acc}!\langle c, 5, \text{“}foo\text{”}\rangle$.

Come back to μCOWS and put in parallel the term $(*)$ with $[z]\,\mathtt{acc}?\langle z, 5, \text{“}foo\text{”}\rangle.\,\mathtt{req}!\langle z, l_{no}\rangle$, that is a malicious service able to guess the content of the exchanged message (except for the correlation datum that, indeed, is a restricted name), i.e.

$$\begin{array}{l}[c]\,(\,(\mathtt{acc}!\langle c, 5, \text{“}foo\text{”}\rangle \mid \langle\!\langle s \rangle\!\rangle_{\emptyset}) \;\mid\; [x, y]\,\mathtt{acc}?\langle c, x, y\rangle.\,\mathtt{req}!\langle c, l_{ok}\rangle\,) \\ \mid\; [z]\,\mathtt{acc}?\langle z, 5, \text{“}foo\text{”}\rangle.\,\mathtt{req}!\langle z, l_{no}\rangle\end{array}$$

Then, a communication bug can occur and lead to the following term

$$[c]\,(\,\langle\!\langle s \rangle\!\rangle_{\emptyset} \;\mid\; [x, y]\,\mathtt{acc}?\langle c, x, y\rangle.\,\mathtt{req}!\langle c, l_{ok}\rangle \;\mid\; \mathtt{req}!\langle c, l_{no}\rangle\,)$$

where $\mathtt{req}!\langle c, l_{no}\rangle$ is enabled instead of $\mathtt{req}!\langle c, l_{ok}\rangle$.

This demonstrates that, in general, service communication based on correlation can be prone to security issues. Such problems are due to the fact that messages do not contain enough ‘private information’ to be delivered in a secure way. However, we can easily cope with this problem by using the restricted correlation datum more than once within each message. Therefore, before to apply the encoding to a term, we analyse it to calculate the maximal length of messages. Then, let r be this length, we will encode each message $\bar{\epsilon}$ along the session channel c with the message $(\underbrace{c, \ldots, c}_{r+1}, \bar{\epsilon})$. In this way, the previous example can be rewritten as follows:

$$\begin{array}{l}[c]\,(\,(\mathtt{acc}!\langle c, c, c, 5, \text{“}foo\text{”}\rangle \mid \langle\!\langle s \rangle\!\rangle_{\emptyset}) \;\mid\; [x, y]\,\mathtt{acc}?\langle c, c, c, x, y\rangle.\,\mathtt{req}!\langle c, c, c, l_{ok}\rangle\,) \\ \mid\; [z_1, z_2, z_3]\,\mathtt{acc}?\langle z_1, z_2, z_3, 5, \text{“}foo\text{”}\rangle.\,\mathtt{req}!\langle z_1, z_2, z_3, l_{no}\rangle\end{array}$$

This time, the correct receive activity $\mathtt{acc}?\langle c, c, c, x, y\rangle$ always wins the race condition with the malicious one. However, also this last encoding does not work for some session protocols, since it requires that when a party is ready to send a message, the other party is ready to receive it. We leave the generalisation of this encoding for future work.

Of course, we could solve the above security problems also by using private endpoints, as it is done in session-based calculi [112, 57, 35], i.e. the encoding of the session request creates a private endpoint to receive data and sends it to the encoding of the corresponding session acceptance, that creates in its turn a private endpoint to receive data and sends it to the other party. However, here we want to point out that correlation can be exploited in a powerful and flexible way.

'Blind date' session joining. We present here a service capable of arranging matches of 2-players online games, such as e.g. morra or rock/paper/scissors[4] (here called *rps* for short). To join a match, a player has only to provide the kind of game and its partner name; players do not need to know on advance any further information, such as e.g. the identifier of the match or the identifier of the other player. Thus, the arrangement of matches is completely transparent to players (for this reason, we call 'blind date' this particular kind of session joining). Such service can be rendered in COWS as follows:

$$\begin{aligned} \mathit{masterService} \triangleq\ & *\,[x_{game}, x_{player1}, x_{player2}] \\ & \quad \mathit{master} \bullet \mathit{join}?\langle x_{game}, x_{player1}\rangle. \\ & \quad \mathit{master} \bullet \mathit{join}?\langle x_{game}, x_{player2}\rangle. \\ & \qquad [\mathit{matchId}]\,(\, x_{player1} \bullet \mathit{start}!\langle \mathit{matchId}\rangle \mid x_{player2} \bullet \mathit{start}!\langle \mathit{matchId}\rangle\,) \end{aligned}$$

Consider now the following players

$$\mathit{Player}_A \triangleq \mathit{master} \bullet \mathit{join}!\langle \mathit{morra}, p_A\rangle \mid [x_{id}]\, p_A \bullet \mathit{start}?\langle x_{id}\rangle.\, \langle \text{rest of } \mathit{Player}_A\rangle$$

$$\mathit{Player}_B \triangleq \mathit{master} \bullet \mathit{join}!\langle \mathit{rps}, p_B\rangle \mid [x_{id}]\, p_B \bullet \mathit{start}?\langle x_{id}\rangle.\, \langle \text{rest of } \mathit{Player}_B\rangle$$

$$\mathit{Player}_C \triangleq \mathit{master} \bullet \mathit{join}!\langle \mathit{morra}, p_C\rangle \mid [x_{id}]\, p_C \bullet \mathit{start}?\langle x_{id}\rangle.\, \langle \text{rest of } \mathit{Player}_C\rangle$$

and the system

$$\mathit{Player}_A \mid \mathit{Player}_B \mid \mathit{Player}_C \mid \mathit{masterService}$$

If Player_A requests to join to a match, since there are not matches under arrangement, *masterService* initialises a new match and the system evolves to:

$$\begin{aligned} & [x_{id}]\, p_A \bullet \mathit{start}?\langle x_{id}\rangle.\, \langle \text{rest of } \mathit{Player}_A\rangle \\ & \mid \mathit{Player}_B \mid \mathit{Player}_C \mid \mathit{masterService} \\ & \mid [x_{player2}]\, \mathit{master} \bullet \mathit{join}?\langle \mathit{morra}, x_{player2}\rangle. \\ & \qquad [\mathit{matchId}]\,(\, p_A \bullet \mathit{start}!\langle \mathit{matchId}\rangle \mid x_{player2} \bullet \mathit{start}!\langle \mathit{matchId}\rangle\,) \end{aligned}$$

[4] For an account of rock/paper/scissors visit `http://en.wikipedia.org/wiki/Rock-paper-scissors`.

Now, if $Player_B$ invokes *masterService*, a second match instance is created:

$$
\begin{array}{l}
[x_{id}]\, p_A \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_A\rangle \\
\mid\ [x_{id}]\, p_B \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_B\rangle \\
\mid\ Player_C \mid masterService \\
\mid\ [x_{player2}]\, master \bullet join?\langle morra, x_{player2}\rangle. \\
\qquad\qquad [matchId]\,(\, p_A \bullet start!\langle matchId\rangle \mid x_{player2} \bullet start!\langle matchId\rangle\,) \\
\mid\ [x_{player2}]\, master \bullet join?\langle rps, x_{player2}\rangle. \\
\qquad\qquad [matchId']\,(\, p_B \bullet start!\langle matchId'\rangle \mid x_{player2} \bullet start!\langle matchId'\rangle\,)
\end{array}
$$

When $Player_C$ invokes *masterService*, he is joined to the existing morra match:

$$
\begin{array}{l}
[x_{id}]\, p_A \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_A\rangle \\
\mid\ [x_{id}]\, p_B \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_B\rangle \\
\mid\ [x_{id}]\, p_C \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_C\rangle \\
\mid\ masterService \\
\mid\ [matchId]\,(\, p_A \bullet start!\langle matchId\rangle \mid p_C \bullet start!\langle matchId\rangle\,) \\
\mid\ [x_{player2}]\, master \bullet join?\langle rps, x_{player2}\rangle. \\
\qquad\qquad [matchId']\,(\, p_B \bullet start!\langle matchId'\rangle \mid x_{player2} \bullet start!\langle matchId'\rangle\,)
\end{array}
$$

Finally, since a morra match has been completely arranged, $Player_A$ and $Player_C$ can start to play, while $Player_B$ continues to wait an *rps* player:

$$
\begin{array}{l}
[matchId]\,(\, \langle \text{rest of } Player_A\rangle \mid \langle \text{rest of } Player_C\rangle\,) \\
\mid\ [x_{id}]\, p_B \bullet start?\langle x_{id}\rangle.\, \langle \text{rest of } Player_B\rangle \\
\mid\ masterService \\
\mid\ [x_{player2}]\, master \bullet join?\langle rps, x_{player2}\rangle. \\
\qquad\qquad [matchId']\,(\, p_B \bullet start!\langle matchId'\rangle \mid x_{player2} \bullet start!\langle matchId'\rangle\,)
\end{array}
$$

Of course, *masterService* can be easily tailored to arrange match for n-players online games with $n > 2$ (e.g. poker, bridge, ...). Notably, a player can play concurrently in more than one match. However, for the sake of simplicity, we assume here that a player can ask to play to the same game more than once only if each time it waits the response to the previous request, otherwise it can be assigned more than once to the same match.

3.2.3 COWS

COWS is obtained by enriching μCOWS with two primitive operators to enable fault and compensation handling and guarantee transactional properties of services.

3.2.3.1 Syntax

The syntax of COWS is presented in Table 3.9 (the new constructs are highlighted by a gray background). In COWS, besides the sets of values and variables, we also use the set of *(killer) labels* (ranged over by $k, k', \ldots$). Notably, expressions do not include killer labels that, hence, are *non-communicable* values. This way the scope of killer labels

s	::=		(services)	g ::=	(receive-guarded choice)
		$\mathbf{kill}(k)$	(kill)	$\mathbf{0}$	(nil)
	$\mid$	$u \bullet u'!\bar{\epsilon}$	(invoke)	$\mid$ $p \bullet o?\bar{w}.s$	(request processing)
	$\mid$	g	(receive-guarded choice)	$\mid$ $g + g$	(choice)
	$\mid$	$s \mid s$	(parallel composition)		
	$\mid$	$\{\!\{s\}\!\}$	(protection)		
	$\mid$	$[e]\, s$	(delimitation)		
	$\mid$	$* s$	(replication)		

Table 3.9: COWS syntax

$\{\!\{\mathbf{0}\}\!\}$	$\equiv$	$\mathbf{0}$	$[k]\,\mathbf{0}$	$\equiv$	$\mathbf{0}$
$\{\!\{\{\!\{s\}\!\}\}\!\}$	$\equiv$	$\{\!\{s\}\!\}$	$[k_1]\,[k_2]\,s$	$\equiv$	$[k_2]\,[k_1]\,s$
$\{\!\{[e]\, s\}\!\}$	$\equiv$	$[e]\,\{\!\{s\}\!\}$	$s_1 \mid [k]\, s_2$	$\equiv$	$[k]\,(s_1 \mid s_2)$ if $k \notin \mathrm{fk}(s_1) \cup \mathrm{fk}(s_2)$

Table 3.10: COWS structural congruence (additional laws)

cannot be dynamically extended and the activities whose termination would be forced by execution of a kill can be statically determined.

We still use w to range over values and variables, u to range over names and variables, while we use e to range over *elements*, namely killer labels, names and variables. Delimitation now is a binder also for killer labels. $\mathrm{fe}(t)$ denotes the set of free elements in t, and $\mathrm{fk}(t)$ denotes the set of free killer labels in t. A closed service is a COWS term without free variables and killer labels.

3.2.3.2 Operational semantics

The structural congruence $\equiv$ for COWS, besides the laws in Table 3.2, additionally includes the laws in Tables 3.10. Notably, the last law of Table 3.10 prevents extending the scope of a killer label k when it is free in s_1 or s_2 (this avoids involving s_1 in the effect of a kill activity inside s_2 and is essential to statically determine which activities can be terminated by a kill). Thus, this law can be used to garbage-collect killer labels, e.g. $[k]\,\mathtt{n}!\bar{\epsilon} \equiv [k]\,(\mathtt{n}!\bar{\epsilon} \mid \mathbf{0}) \equiv \mathtt{n}!\bar{\epsilon} \mid [k]\,\mathbf{0} \equiv \mathtt{n}!\bar{\epsilon} \mid \mathbf{0} \equiv \mathtt{n}!\bar{\epsilon}$.

To define the labelled transition relation, we need two new auxiliary functions. The function $halt(\_)$ takes a service s as an argument and returns the service obtained by only retaining the protected activities inside s. $halt(\_)$ is defined inductively on the syntax of services. The most significant case is $halt(\{\!\{s\}\!\}) = \{\!\{s\}\!\}$. In the other cases, $halt(\_)$ returns $\mathbf{0}$, except for parallel composition, delimitation and replication operators, for which it acts as an homomorphism.

$$halt(\mathbf{kill}(k)) = halt(\mathtt{u}!\bar{\epsilon}) = halt(g) = \mathbf{0} \qquad halt(\{\!\{s\}\!\}) = \{\!\{s\}\!\}$$

$$halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2) \qquad halt([e]\, s) = [e]\, halt(s) \qquad halt(* s) = * halt(s)$$

$\text{noKill}(s, e) = \mathbf{true}$ if $\text{fk}(e) = \emptyset$	$\text{noKill}(s \mid s', k) = \text{noKill}(s, k) \wedge \text{noKill}(s', k)$
$\text{noKill}(\mathbf{kill}(k), k) = \mathbf{false}$	$\text{noKill}([e]\, s, k) = \text{noKill}(s, k)$ if $e \neq k$
$\text{noKill}(\mathbf{kill}(k'), k) = \mathbf{true}$ if $k \neq k'$	$\text{noKill}([k]\, s, k) = \mathbf{true}$
$\text{noKill}(\mathtt{u}!\bar{\epsilon}, k) = \text{noKill}(g, k) = \mathbf{true}$	$\text{noKill}(\{\!\{s\}\!\}, k) = \text{noKill}(* s, k) = \text{noKill}(s, k)$

Table 3.11: There are no active $\mathbf{kill}(k)$

Then, in Table 3.11, we inductively define the predicate $\text{noKill}(s, e)$, that holds true if either e is not a killer label or $e = k$ and s cannot immediately perform a free kill activity $\mathbf{kill}(k)$. Moreover, the predicate $\text{noConf}(s, \mathtt{n}, \bar{v}, \ell)$, defined for μCOWS by the rules in Table 3.7, is extended to COWS by adding the following rules:

$$\text{noConf}(\mathbf{kill}(k), \mathtt{n}, \bar{v}, \ell) = \mathbf{true} \qquad \text{noConf}(\{\!\{s\}\!\}, \mathtt{n}, \bar{v}, \ell) = \text{noConf}(s, \mathtt{n}, \bar{v}, \ell)$$

$$\text{noConf}([e]\, s, \mathtt{n}, \bar{v}, \ell) = \begin{cases} \text{noConf}(s, \mathtt{n}, \bar{v}, \ell) & \text{if } e \notin \mathtt{n} \\ \mathbf{true} & \text{otherwise} \end{cases}$$

The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over COWS services induced by the rules in Table 3.12. The rules in the upper part of the table are directly borrowed from μCOWS (Table 3.6), while those in the lower part are the new rules used to deal with forced termination. Labels are now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd \bar{v} \quad | \quad \mathtt{n} \rhd \bar{w} \quad | \quad \mathtt{n}\, \sigma\, \ell\, \bar{v} \quad | \quad k \quad | \quad \dagger$$

The meaning of the new labels is as follows: k denotes execution of a request for terminating a term from within the delimitation $[k]$, and $\dagger$ denotes a computational step corresponding to taking place of forced termination. In the sequel, we use $\text{e}(\alpha)$ to denote the set of elements occurring in α (it is defined similarly to $\text{u}(\alpha)$, Section 3.2.1.2, page 46).

Let us now comment on the operational rules. Activity $\mathbf{kill}(k)$ forces termination of all unprotected parallel activities (rules *(kill)* and (par_{kill})) inside an enclosing $[k]$, that stops the killing effect by turning the transition label k into $\dagger$ (rule $(del_{kill\,1})$). Existence of such delimitation is ensured by the assumption that the semantics is only defined for closed services. Critical activities can be protected from killing by putting them into a protection $\{\!\{\_\}\!\}$; this way, $\{\!\{s\}\!\}$ behaves like s (rule *(prot)*). Similarly, $[e]\, s$ behaves like s (rule (del_2)), except when the transition label α contains e, in which case α must correspond either to a communication assigning a value to e (rule $(del_{com\,2})$) or to a kill activity for e (rule $(del_{kill\,1})$), or when a free kill activity for e is active in s, in which case only actions corresponding to kill activities can be executed (rules $(del_{kill\,2})$ and $(del_{kill\,3})$), that also apply when the third premise of (del_2) does not hold, i.e. $\alpha = k$ or $\alpha = \dagger$). This means that kill activities are executed *eagerly* with respect to the activities enclosed within the delimitation of the corresponding killer label. Execution of parallel services is interleaved (rule (par_3)), but when a kill activity or a communication is performed. Indeed, the former must trigger termination of all parallel services (according to rule (par_{kill})), while the latter must ensure that the receive activity with greater priority progresses (rules (com_2) and (par_{com})).

$$\frac{[\![\bar{\epsilon}]\!] = \bar{v}}{\mathrm{n}!\bar{\epsilon} \xrightarrow{\mathrm{n} \lhd \bar{v}} \mathbf{0}}\ (inv) \qquad \mathrm{n}?\bar{w}.s \xrightarrow{\mathrm{n} \rhd \bar{w}} s\ (rec) \qquad \frac{g \xrightarrow{\alpha} s}{g + g' \xrightarrow{\alpha} s}\ (choice)$$

$$\frac{s_1 \xrightarrow{\mathrm{n} \rhd \bar{w}} s_1' \quad s_2 \xrightarrow{\mathrm{n} \lhd \bar{v}} s_2' \quad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \quad \mathrm{noConf}(s_1 \mid s_2, \mathrm{n}, \bar{v}, |\sigma|)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma|\sigma|\bar{v}} s_1' \mid s_2'}\ (com_2)$$

$$\frac{s \xrightarrow{\mathrm{n}\sigma \uplus \{x \mapsto v\}\,\ell\,\bar{v}} s'}{[x]\, s \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s' \cdot \{x \mapsto v\}}\ (del_{com\,2}) \qquad \frac{s \equiv \xrightarrow{\alpha} \equiv s'}{s \xrightarrow{\alpha} s'}\ (str)$$

$$\frac{s_1 \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s_1' \qquad \mathrm{noConf}(s_2, \mathrm{n}, \bar{v}, \ell)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma\ell\bar{v}} s_1' \mid s_2}\ (par_{com})$$

$$\mathbf{kill}(k) \xrightarrow{k} \mathbf{0}\ (kill) \qquad \frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}\ (prot)$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq k, \mathrm{n}\sigma\ell\bar{v}}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (par_3) \qquad \frac{s_1 \xrightarrow{k} s_1'}{s_1 \mid s_2 \xrightarrow{k} s_1' \mid halt(s_2)}\ (par_{kill})$$

$$\frac{s \xrightarrow{k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'}\ (del_{kill\,1}) \qquad \frac{s \xrightarrow{k} s' \quad k \neq e}{[e]\, s \xrightarrow{k} [e]\, s'}\ (del_{kill\,2})$$

$$\frac{s \xrightarrow{\dagger} s'}{[e]\, s \xrightarrow{\dagger} [e]\, s'}\ (del_{kill\,3}) \qquad \frac{s \xrightarrow{\alpha} s' \quad e \notin \mathrm{e}(\alpha) \quad \alpha \neq k, \dagger \quad \mathrm{noKill}(s, e)}{[e]\, s \xrightarrow{\alpha} [e]\, s'}\ (del_2)$$

Table 3.12: COWS operational semantics

3.2.3.3 Examples

We conclude with some examples aimed at clarifying the peculiar features of COWS.

An 'efficient' implementation of standard variables. Service Var_{x}, described in Section 3.2.1.3 (page 50), implements a standard variable, which can be repeatedly assigned, by creating a new service instance to store the current value any time that operations *read* or *write* are invoked. This is due to the fact that the two offered operations are provided by two receives composed by using the choice operator. In fact, in case of reading, since the stored value does not change, it is not necessary to re-instantiate the service. This

behaviour can now easily implemented by exploiting kill and protection operators as follows:

$$\begin{aligned} \mathit{Var}_{\mathtt{x}} \triangleq\ & [x_v, x_a]\, \mathtt{x} \bullet \mathit{write}?\langle x_v, x_a\rangle. \\ & [\mathtt{n}]\,(\,\mathtt{n}!\langle x_v, x_a\rangle \\ & \quad \mid *\,[x,y]\,\mathtt{n}?\langle x,y\rangle.\,(y!\langle\rangle \mid [k]\,(*\,[y']\,\mathtt{x} \bullet \mathit{read}?\langle y'\rangle.\,\{\!|y'!\langle x\rangle|\!\} \\ & \qquad\qquad \mid [x',y']\,\mathtt{x} \bullet \mathit{write}?\langle x',y'\rangle. \\ & \qquad\qquad\quad (\,\mathbf{kill}(k) \mid\ \{\!|\mathtt{n}!\langle x',y'\rangle|\!\}\,)\,)\,)\,) \end{aligned}$$

Now, read access to the variable does not require re-instantiation of the whole service, since the receive along $\mathtt{x} \bullet \mathit{read}$ is replicated. Instead, when the stored value must be updated, the current instance is terminated, by executing $\mathbf{kill}(k)$, and a new instance storing the new value is created (alike the memory cell service of [34]). Delimitation $[k]$ is used to confine the effect of the kill activity to the current instance, while protection $\{\!|\_|\!\}$ avoids forcing termination of pending replies and of the invocation that will trigger the new instance.

Protected kill activity. The following simple example illustrates the effect of executing a kill activity within a protection block:

$$[k]\,(\{\!|s_1 \mid \{\!|s_2|\!\} \mid \mathbf{kill}(k)|\!\} \mid s_3) \mid s_4 \xrightarrow{\dagger} [k]\,\{\!|\,\{\!|s_2|\!\}\,|\!\} \mid s_4$$

where, for simplicity, we assume that $\mathit{halt}(s_1) = \mathit{halt}(s_3) = \mathbf{0}$. In essence, $\mathbf{kill}(k)$ terminates all parallel services inside delimitation $[k]$ (i.e. s_1 and s_3), except those that are protected at the same nesting level of the kill activity (i.e. s_2).

Interplay between communication and kill activity. Kill activities can break communication, as the following example shows:

$$\mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k)) \xrightarrow{\dagger} \mathtt{n}!\langle v\rangle \mid [k]\,[x]\,\mathbf{0}$$

In fact, due to the priority of the kill activity over communication, this is the only possible evolution of the above term. Communication can however be guaranteed by protecting the receive activity, as follows

$$\begin{aligned} & \mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\} \mid \mathbf{kill}(k)) && \xrightarrow{\dagger} \\ & \mathtt{n}!\langle v\rangle \mid [k]\,[x]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\} && \equiv \\ & [x]\,(\mathtt{n}!\langle v\rangle \mid [k]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\}) && \xrightarrow{\mathtt{n}\,\emptyset\,1\,\langle v\rangle} \\ & [k]\,\{\!|s \cdot \{x \mapsto v\}|\!\} \end{aligned}$$

Notably, priority of kill activities over communication acts only with respect to the activities enclosed within the delimitation of the corresponding killer labels (i.e. priority is *local* to killer label scopes). For instance, if we re-write the above example as follows:

$$[y]\,\mathtt{n}?\langle y\rangle.s' \mid \mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k))$$

communication between $\mathtt{n}!\langle v\rangle$ and $\mathtt{n}?\langle x\rangle$ is still preempted by $\mathbf{kill}(k)$, while communication with $\mathtt{n}?\langle y\rangle$ can take place and lead to

$$s' \cdot \{y \mapsto v\} \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k))$$

Non-communicability of killer labels. We require killer labels not to be communicable to avoid a service be capable to indiscriminately stop the execution of other services' activities. However, when desired, this behaviour can be modelled in COWS. Consider, for example, the following term where two parallel services share the private name *stop*:

$$[stop]\,(\,s_1 \mid s_2\,) \mid s_3$$

where $s_1 \triangleq [k]\,(\mathtt{n}?\langle stop\rangle.\,\mathbf{kill}(k) \mid s_1')$ and $s_2 \triangleq \mathtt{n}!\langle stop\rangle \mid s_2'$. In s_1, the activity $\mathbf{kill}(k)$ is prefixed by the receive $\mathtt{n}?\langle stop\rangle$ that does not allow forced termination to take place until the 'termination signal' *stop* is received. In fact, if a communication between s_1 and s_2 takes place along the endpoint $\mathtt{n}$, the term evolves to

$$[stop]\,(\,[k]\,(\mathbf{kill}(k) \mid s_1') \mid s_2'\,) \mid s_3$$

Now, due to the priority of the kill activity over communication, the term $[k]\,(\mathbf{kill}(k) \mid s_1')$ can only perform a kill activity and evolve, e.g., to $[k]\,halt(s_1')$.

Delimitation of killer labels. We require killer labels to be delimited to avoid a single service be capable to stop all the other parallel services which would be unreasonable in a service-oriented setting. Indeed, suppose a service s can perform a $\mathbf{kill}(k)$ with k undelimited in s. The killing effect could not be stopped, thus, due to a transition labelled by k, the whole service s would be terminated (but for protected activities). Moreover, the effect of $\mathbf{kill}(k)$ could not be confined to s, thus, if there are other parallel services, the whole service composition might be terminated by $\mathbf{kill}(k)$.

Fault and compensation handlers. In the SOC approach, fault handling is strictly related to the notion of *compensation*, namely the execution of specific activities (attempting) to reverse the effects of previously executed activities. Here, we consider a WS-BPEL compensation protocol[5]. To begin with, we extend COWS syntax as shown in the upper part of Table 3.13. The *scope* activity $[s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i$ permits explicitly grouping activities together. The declaration of a scope activity contains a unique scope identifier i, a service s representing the normal behaviour, an optional list of fault handlers $s_1, \ldots, s_n$, and a compensation handler s_c. The *fault generator* activity

[5]This protocol only permits to compensate on specified inner scopes and does not provide an automatic reverse compensation mechanism à la Sagas [98]. This latter mechanism, however, can be realized in COWS by relying on queues; for an account of this protocol implemented in COWS we refer to [12] and [129].

$$
\begin{array}{rcll}
s & ::= & \ldots & \text{(services)} \\
 & | & \mathbf{throw}(\phi) & \text{(fault generator)} \\
 & | & \mathbf{compensate}(i) & \text{(compensate)} \\
 & | & [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i & \text{(scope)}
\end{array}
$$

$$
\begin{array}{rcl}
\llparenthesis [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i \rrparenthesis_k & = & \\
\multicolumn{3}{r}{[\phi_1, \ldots, \phi_n]\,(\,\llparenthesis \mathbf{catch}(\phi_1)\{s_1\} \rrparenthesis_k \mid \ldots \mid \llparenthesis \mathbf{catch}(\phi_n)\{s_n\} \rrparenthesis_k} \\
\multicolumn{3}{r}{\mid [k_i]\,\llparenthesis s \rrparenthesis_{k_i}\,;\,(\,x_{done} \bullet o_{done}!\langle\rangle \mid [k']\,\{\!|\mathtt{undo}?\langle i\rangle.\llparenthesis s_c \rrparenthesis_{k'}|\!\}\,)\,)} \\
\llparenthesis \mathbf{catch}(\phi)\{s\} \rrparenthesis_k & = & \mathtt{throw}?\langle\phi\rangle.[k']\,\llparenthesis s \rrparenthesis_{k'} \\
\llparenthesis \mathbf{compensate}(i) \rrparenthesis_k & = & \mathtt{undo}!\langle i\rangle \mid x_{done} \bullet o_{done}!\langle\rangle \\
\llparenthesis \mathbf{throw}(\phi) \rrparenthesis_k & = & \{\!|\mathtt{throw}!\langle\phi\rangle|\!\} \mid \mathbf{kill}(k)
\end{array}
$$

Table 3.13: Syntax and encoding of fault and compensation handling

throw(ϕ) can be used by a service to rise a fault signal ϕ. This signal will trigger execution of activity s', if a construct of the form **catch**(ϕ)$\{s'\}$ exists within the same scope. The *compensate* activity **compensate**(i) can be used to invoke a compensation handler of an inner scope named i that has already completed normally (i.e. without faulting). Compensation can only be invoked from within a fault or a compensation handler. Here, we fix two syntactic constraints: handlers do not contain scope activities and, as in WS-BPEL (see [166, Section 12.4.3.1]), for each **compensate**(i) occurring in a service there exists at least an inner scope i. Notably, an activity $[s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i$ acts as a binder for $\phi_1, \ldots, \phi_n$; in this way, a scope can only catch and handle faults coming from its enclosed activities.

Now we show that fault and compensation handling can be easily encoded in COWS, thus it is not necessary to extend its syntax. The most interesting cases of the encoding are shown in the lower part of Table 3.13 (in the remaining cases, the encoding acts as an homomorphism). The two distinguished endpoints `throw` and `undo` are used for exchanging fault and compensation signals, respectively. Each scope identifier i or fault signal ϕ can be used to activate scope compensation or fault handling, respectively.

The encoding $\llparenthesis \cdot \rrparenthesis_k$ is parameterized by the label k that identifies the closest enclosing scope, if any. The parameter is used when encoding a fault generator, to launch a kill activity that forces termination of all the remaining activities of the enclosing scope, and when encoding a scope, to delimit the field of action of inner kill activities. The compensation handler s_c of scope i is installed when the normal behaviour s successfully completes, but it is activated only when signal $\mathtt{undo}!\langle i\rangle$ occurs. Similarly, if during normal execution a fault ϕ occurs, a signal $\mathtt{throw}!\langle\phi\rangle$ triggers execution of the corresponding fault handler (if any). Installed compensation handlers are protected from killing by means of $\{\!|\_|\!\}$. Notably, the compensate activity can immediately terminate (thus enabling possible sequential compositions, see encoding (3.7) at page 52); this, of course, does not mean that

the corresponding handler is terminated.

At this moment, it is not completely clear to us if it is possible to define a sound encoding of fault and compensation constructs into π-calculus. However, if we assume that it can be defined, we believe that the encoded terms would be very large and, probably, the encoding would be untenable (e.g. non-compositional or semantically not complete). This motivates our choice of considering a calculus equipped with specific primitives to deal with fault and compensation handling. Anyway, as a future work we plan to further investigate the expressive power of such primitives.

A shipping service scenario. We consider an extended version of the shipping service described in the official specification of WS-BPEL [166, Section 15.1]. This example allows us to illustrate most of the language features and previously introduced high-level constructs, including message correlation, shared variables, control flow structures, fault and compensation handling.

The shipping service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The service offers two types of shipment: shipments where the items are held and shipped together and shipments where the items are shipped piecemeal until the order is fulfilled. A possible computation of a scenario where the shipping service interacts with a customer service is shown in the customized UML sequence diagram of Figure 3.3. The shipping service is specified in COWS as follows:

$$
\begin{array}{l}
* \ [x_{cust}, x_{id}, x_c, x_{items}] \\
\quad ship \bullet req?\langle x_{cust}, x_{id}, x_c, x_{items}\rangle . \\
\quad \textbf{if } (x_c) \ \textbf{then } \{ \ x_{cust} \bullet notice!\langle x_{id}, x_{items}\rangle \ \} \\
\qquad \textbf{else } \{ \ [\ s_{ship} \ : \ \textbf{catch}(\phi_{noItems})\{ \ \textbf{compensate}(price) \\
\qquad\qquad\qquad\qquad\qquad\qquad | \ x_{cust} \bullet err!\langle x_{id}, \text{“}sorry\text{”}\rangle \ \} \ : \mathbf{0} \ ]_{non-complete} \ \}
\end{array}
$$

where the normal behaviour s_{ship} is

$$
\begin{array}{l}
[x_{ratio}] \, (\, [\, s_{priceCalc} \ : \ s_{priceComp} \,]_{price} \ ; \\
\qquad [while] \, (\, ship \bullet while!\langle 0\rangle \\
\qquad\qquad | \ * \, [x_{shipped}] \, ship \bullet while?\langle x_{shipped}\rangle . \\
\qquad\qquad\qquad \textbf{if } (x_{shipped} < x_{items}) \ \textbf{then } \{ \ [x_{count}] \\
\qquad\qquad\qquad\qquad [x_{count} = rand(x_{items} - x_{shipped})]. \\
\qquad\qquad\qquad\qquad \textbf{if } (x_{count} \leqslant 0) \ \textbf{then } \{ \\
\qquad\qquad\qquad\qquad\qquad [x_{ratio} = x_{shipped} \ / \ x_{items}]. \textbf{throw}(\phi_{noItems}) \ \} \\
\qquad\qquad\qquad\qquad \textbf{else } \{ \ x_{cust} \bullet notice!\langle x_{id}, x_{count}\rangle \\
\qquad\qquad\qquad\qquad\qquad | \ ship \bullet while!\langle x_{shipped} + x_{count}\rangle \ \} \ \} \ ) \ )
\end{array}
$$

The partner name *ship* is associated with the shipping service, the operation name *req* is used to receive the shipping request, and the tuple of variables $\langle x_{cust}, x_{id}, x_c, x_{items}\rangle$ is used for the request shipping message: x_{cust} stores the customer's partner name, x_{id} stores the order identifier, that is used to correlate the ship notice(s) with the ship order, x_c stores

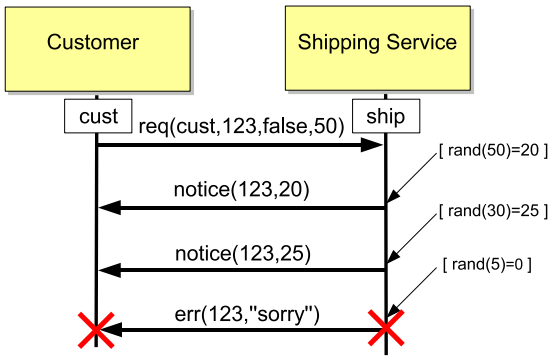

Figure 3.3: A computation in the shipping service scenario

a boolean indicating whether the order is to be shipped complete or not, and x_{items} stores the total number of items in the order. Shipping notices and error messages to customers are sent using the partner stored in x_{cust} and operations *notice* and *err*, respectively. A notice message is a tuple composed of the order identifier and the number of items in the shipping notice. When partial shipment is acceptable, $x_{shipped}$ is used to record the number of items already shipped. Replication and the internal operation *while* are used to model iteration.

Our example extends that in [166] by allowing the service to generate a fault in case the shipping company has ended the stock of items (this is modelled by function $rand(n)$ returning an integer less or equal to 0). The fault is handled by sending an error message to the customer and by compensating the inner scope *price*, that has already completed successfully. Function $rand(n)$ returns a random integer number not greater than n and represents an internal interaction with a back-end system (that, for simplicity, we do not further describe). Moreover, we do not show services $s_{priceCalc}$ and $s_{priceComp}$. Basically, the former calculates the shipping price according to the value assigned to x_{items} and sends the result to the accounts department. The latter is the corresponding compensation activity, that sends information about the non-shipped items to the accounts department and sends a refund to the customer according to the ratio (stored in x_{ratio}) between the shipped items (stored in $x_{shipped}$) and the required ones (stored in x_{items}).

3.3 A formal account of WS-BPEL

In this section, we informally present the close correspondence between WS-BPEL activities and COWS terms.

3.3.1 Basic activities

We start with the mapping of activities allowing a WS-BPEL process to send/receive a message. Among the different patterns of interaction provided by WSDL, only two of

<table>
<tr><td colspan="2">Partner link type</td></tr>
<tr><td colspan="2"><partnerLinkType name="plt">
  <role name="r"> <portType name="pt"/> </role>
</partnerLinkType></td></tr>
<tr><td>Client-side</td><td>Provider-side</td></tr>
<tr><td>WS-BPEL
<partnerLink name="plc"
 partnerLinkType="plt"
 partnerRole="r" />
<invoke partnerLink="plc"
 operation="op" inputVariable="y1" /></td><td>WS-BPEL
<partnerLink name="plp"
 partnerLinkType="plt"
 myRole="r" />
<receive partnerLink="plp"
 operation="op" variable="y2" /></td></tr>
<tr><td>COWS
$u_r \cdot op!\bar{y}_1$</td><td>COWS
$p_r \cdot op?\bar{y}_2$</td></tr>
</table>

Table 3.14: Mapping of *partner links*, *invoke* and *receive* activities (one-way)

them are directly supported by WS-BPEL: one-way and (synchronous) request-response. There is another interaction pattern that is largely used in WS-BPEL (see, e.g., the example 15.1 in [166]) but it is not directly provided by WSDL: asynchronous request-response. These three interaction patterns are illustrated in Tables 3.14, 3.15 and 3.16. It is worth noticing the use of the element <partnerLinkType> that, in case of the first two interaction patterns, contains a single role, while, in case of the last pattern, contains two roles. Each role has associated an element <portType> that declares a set of operations. Each interacting partner must implement an element <partnerLink> with an associated <partnerLinkType>. Elements <partnerLinkType> and <portType> do not have a counterpart in COWS (that does not have declarative constructs), instead the information they provide is exploited to map elements <partnerLink> into COWS partner names, that are then used in receive and invoke activities.

An activity <receive> allows a WS-BPEL process to wait for a matching message to arrive and can be associated with an activity <reply> to form a synchronous request-response operation. One-way is the simplest interaction pattern: the service providing the operation performs the receive activity, whereas the client service performs the invoke activity. Our mapping in Table 3.14 shows how one-way <receive> and <invoke> activities are directly supported in COWS. Here, we use u_r to indicate the fact that the client either knows the provider's partner name at design-time (when $u_r = p_r$) or discover it dynamically (when $u_r = x_r$). A synchronous request-response interaction, as shown in Table 3.15, is implemented in COWS through a pair of one-way interactions (the request and the callback). Thus, COWS forces the client to send the partner *cb* used by the provider for sending the reply back to the client. On the other hand, to be capable to handle such a request, the service providing the operation is ready to receive also such partner name (stored in x). Table 3.16 shows that an asynchronous request-response is implemented through a partner link connecting two one-way interactions. Here, the partner u_{r2} is used whether it is required to initialize a partner link's partner role. If the attribute

<table>
<tr><td colspan="2">Partner link type</td></tr>
<tr><td colspan="2"><partnerLinkType name="plt">
  <role name="r"> <portType name="pt"/> </role>
</partnerLinkType></td></tr>
<tr><td>Client-side</td><td>Provider-side</td></tr>
<tr><td>WS-BPEL
<partnerLink name="plc"
 partnerLinkType="plt"
 partnerRole="r" />
<invoke partnerLink="plc"
 operation="op" inputVariable="yi"
 outputVariable="yo" /></td><td>WS-BPEL
<partnerLink name="plp"
 partnerLinkType="plt"
 myRole="r" />
<receive partnerLink="plp"
 operation="op" variable="zi" />
...
<reply partnerLink="plp"
 operation="op" variable="zo" /></td></tr>
<tr><td>COWS
$[cb]\,(\,u_r \bullet op!(cb, \bar{y}_i) \mid cb \bullet op?\bar{y}_o\,)$</td><td>COWS
$[x]\,(\,p_r \bullet op?(x, \bar{z}_i)\ \ldots\ x \bullet op!\bar{z}_o\,)$</td></tr>
</table>

Table 3.15: Mapping of *partner links*, *invoke*, *receive* and *reply* activities (sync. request-response)

`initializePartnerRole` is set to `yes` then u_{r2} is x_{r2}, otherwise u_{r2} is p_{r2}. Notably, for the sake of simplicity, we do not consider standard variables, i.e. all variables can be assigned only once; however, they could be dealt with by exploiting the encoding introduced in Section 3.2.1.3 (page 50).

The mapping of the remaining basic activities is quite straightforward and is illustrated in Table 3.17. We have adopted the convention that each part of a message is stored in a COWS variable (in Section 3.2.1.3 we have shown that, at a certain level of abstraction, tuples can be used to represent XML messages). Thus, in the mapping of activity <assign>, variables $z_{part1\_of\_x}$ and $z_{part2\_of\_y}$ store message parts `part1_of_x` and `part2_of_y`, respectively. Mappings of the activities <throw> and <compensateScope> are shown in Table 3.13. Activity <exit> is rendered as a kill activity specifying a killer label having as scope a whole process instance (see last raw of Table 3.19), while <empty> is modelled by a COWS term that can only perform an internal step. Notably, the fact that activities <throw>, <compensateScope>, <exit> and <empty> are *short-lived activities* (i.e. sufficiently brief activities that may be allowed to complete) is rendered in COWS by protecting them with $\{\!|\_|\!\}$.

Activities <rethrow> and <compensate> are not considered here. In fact, the former can be rendered by an activity **throw**(ϕ) within a fault handler, by using an appropriate ϕ (recall that, according to the mapping in Table 3.13, scope constructs act as binders for fault names). Instead, implementing activity <compensate> requires some ingenuity mainly because installed compensation handlers have to be stored in queues to execute them in the reverse order of completion when compensation is invoked (for a detailed account of the implementation in COWS of activity <compensate> we refer to [12]). Finally, activities <validate> and <extensionActivity> have been left out because

<table>
<tr><td colspan="2">Partner link type</td></tr>
<tr><td colspan="2"><pre>
<partnerLinkType name="plt">
  <role name="client"> <portType name="pt1"/> </role>
  <role name="provider"> <portType name="pt2"/> </role>
</partnerLinkType>
</pre></td></tr>
<tr><td>Client-side</td><td>Provider-side</td></tr>
<tr><td>WS-BPEL
<pre>
<partnerLink name="plc"
 partnerLinkType="plt"
 myRole="client" />
 partnerRole="provider" />

<invoke partnerLink="plc"
 operation="op1"
 inputVariable="y1" />
...
<receive partnerLink="plc"
 operation="op2"
 variable="y2" />
</pre></td><td>WS-BPEL
<pre>
<partnerLink name="plp"
 partnerLinkType="plt"
 myRole="provider" />
 partnerRole="client"
 initializePartnerRole="yes/no" />
<receive partnerLink="plp"
 operation="op1"
 variable="z1" />
...
<invoke partnerLink="plp"
 operation="op2"
 inputVariable="z2" />
</pre></td></tr>
<tr><td>COWS
$u_{r1} \cdot op_1!(p_{r2}, \bar{y}_1) \ \ldots \ p_{r2} \cdot op_2?\bar{y}_2$</td><td>COWS
$p_{r1} \cdot op_1?(u_{r2}, \bar{z}_1) \ \ldots \ u_{r2} \cdot op_2!\bar{z}_2$</td></tr>
</table>

Table 3.16: Mapping of *partner links*, *invoke* and *receive* activities (async. request-response)

they are specifically related to XML mechanisms (namely, schema validation and language extensibility).

3.3.2 Structured activities

By exploiting the encoding introduced in the previous sections, the mapping of WS-BPEL structured activities into COWS terms is straightforward. Activities <sequence>, <if>, <while>, <repeatUntil> and <flow> are shown in Table 3.18, while activities <pick>, <scope> and <process> are shown in Table 3.19. These latter activities are supported with some limitations due to the fact that COWS does not provide timed activities/events (see Section 5.2.1 for an extension of COWS with timed constructs) and termination signals/handlers (since their handling seems not to differ from that of fault signals/handlers). We omit the mapping of the <forEach> activity because its sequential form is a special case of <while> activity, while its parallel form can be rendered by a sophisticated (but not particularly interesting) encoding (e.g., by exploiting replication to spawn in parallel a non-statically known number of copies of the argument of the <forEach>).

3.3.3 Specification of the WS-BPEL processes from Section 2.2.2

We report here the COWS specifications of the WS-BPEL processes graphically presented in the examples of Section 2.2.2.

WS-BPEL	COWS

`<assign>`
`  <copy>`
`  <from variable="x"`
`  part="part1_of_x" />`
`  <to variable="y"`
`  part="part2_of_y" />`
`  </copy>`
`</assign>`
	$[z_{part2\_of\_y} = z_{part1\_of\_x}]$
	
	where
	$\bar{x} = \langle z_{part1\_of\_x}, z_{part2\_of\_x}, z_{part3\_of\_x} \rangle$
	$\bar{y} = \langle z_{part1\_of\_y}, z_{part2\_of\_y} \rangle$
	
	mappings (3.3) and (3.5)

`<throw faultName="fname" />`	$\{\!\!\{ \mathbf{throw}(\phi_{fname}) \}\!\!\}$ mapping in Table 3.13
`<compensateScope target="sname" />`	$\{\!\!\{ \mathbf{compensate}(i_{sname}) \}\!\!\}$ mapping in Table 3.13
`<exit />`	$\{\!\!\{ \mathbf{kill}(k_{exit}) \}\!\!\}$
`<empty />`	$\{\!\!\{ [\mathtt{n}]\,(\mathtt{n}!\langle\rangle \mid \mathtt{n}?\langle\rangle) \}\!\!\}$

Table 3.17: Mapping of *assign*, *throw*, *compensateScope*, *exit* and *empty* activities

Message correlation. The example of Figure 2.3(a) of the two uncorrelated receive activities corresponds to the following COWS term:

$$\begin{aligned} *\,[x_{cb}, x_{logID}, x_{info}]\; & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info}\rangle. \\ & LogOnServ \bullet RequestLogInfo?\langle\rangle. \\ & x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info}\rangle \end{aligned}$$

By correlating consecutive messages by means of the correlation variable x_{logID}, as in Figure 2.3(b), we obtain the following modified service:

$$\begin{aligned} *\,[x_{cb}, x_{logID}, x_{info}]\; & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info}\rangle. \\ & LogOnServ \bullet RequestLogInfo?\langle x_{logID}\rangle. \\ & x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info}\rangle \end{aligned}$$

The special case when the two initial receives *LogOn* are identical (see Figure 2.3(c)) is expressed in COWS as follows:

$$\begin{aligned} *\,[x_{cb}, x_{logID}, x_{info1}, x_{info2}]\; & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle. \\ & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info2}\rangle. \\ & LogOnServ \bullet RequestLogInfo?\langle x_{logID}\rangle. \\ & x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info1}, x_{info2}\rangle \end{aligned}$$

To illustrate, consider the following composition that also includes the deployment of two client processes:

$$\begin{aligned} *\,[x_{cb}, x_{logID}, x_{info1}, x_{info2}]\; & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle. \\ & LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info2}\rangle. \\ & LogOnServ \bullet RequestLogInfo?\langle x_{logID}\rangle. \\ & x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info1}, x_{info2}\rangle \\ \mid\; (& LogOnServ \bullet LogOn!\langle c_1, id, data_1\rangle \\ & \mid LogOnServ \bullet RequestLogInfo!\langle id\rangle \\ & \mid [x_{info1}, x_{info2}]\, c_1 \bullet SendLogInfo?\langle id, x_{info1}, x_{info2}\rangle.\; s\,) \\ \mid\; & LogOnServ \bullet LogOn!\langle c_1, id, data_2\rangle \end{aligned}$$

WS-BPEL	COWS
<sequence> activity1 ... activityN </sequence>	$s_{activity1}$; ... ; $s_{activityN}$ mapping (3.7)
<if> <condition> cond </condition> activity1 <else> activity2 </else> </if>	**if** (ϵ_{cond}) **then** $\{s_{activity1}\}$ **else** $\{s_{activity2}\}$ mapping (3.4)
<while> <condition> cond </condition> activity </while>	**while** (ϵ_{cond}) $\{s_{activity}\}$ mapping (3.8)
<repeatUntil> activity <condition> cond </condition> </repeatUntil>	$s_{activity}$; **while** (ϵ_{cond}) $\{s_{activity}\}$ mappings (3.7) and (3.8)
<flow> activity1 ... activityN </flow>	$s_{activity1}$ | ... | $s_{activityN}$ mapping in Table 3.5

Table 3.18: Mapping of structured activities

According to the semantics of COWS, the client processes trigger only one instantiation of the service. Thus, the only possible evolution, after three computational steps, leads to

$$
\begin{aligned}
&*\,[x_{cb}, x_{logID}, x_{info1}, x_{info2}]\, LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle.\\
&\qquad\qquad LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info2}\rangle.\\
&\qquad\qquad LogOnServ \bullet RequestLogInfo?\langle x_{logID}\rangle.\\
&\qquad\qquad x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info1}, x_{info2}\rangle\\
&\mid\ c_1 \bullet SendLogInfo!\langle id, data_1, data_2\rangle\\
&\mid\ [x_{info1}, x_{info2}]\, c_1 \bullet SendLogInfo?\langle id, x_{info1}, x_{info2}\rangle.\ s
\end{aligned}
$$

Asynchronous message delivering. To illustrate the example in Figure 2.5, consider the following COWS term:

$$
\begin{aligned}
&*\,[x_{cb}, x_{logID}, x_{info}]\, LogOnServ \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info}\rangle.\\
&\qquad\qquad LogOnServ \bullet RequestLogInfo?\langle x_{logID}\rangle.\\
&\qquad\qquad x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info}\rangle\\
&\mid (\ LogOnServ \bullet RequestLogInfo!\langle id\rangle\\
&\quad \mid LogOnServ \bullet LogOn!\langle c, id, data\rangle \mid [x_{info}]\, c \bullet SendLogInfo?\langle id, x_{info}\rangle.\ s\ )
\end{aligned}
$$

WS-BPEL	COWS

`<pick>`
`  <onMessage partnerLink="pl1"`
`  operation="op1" variable="y1">`
`  activity1`
`  </onMessage>`
`  <onMessage partnerLink="pl2"`
`  operation="op2" variable="y2">`
`  activity2`
`  </onMessage>`
`</pick>`
	$p_1 \bullet op_1?\bar{y}_1.\ s_{activity1} \ + \ p_2 \bullet op_2?\bar{y}_2.\ s_{activity2}$

`<scope name="i">`
`  <faultHandlers>`
`  <catch faultName="fault1">`
`  activityF1`
`  </catch>`
`  <catch faultName="fault2">`
`  activityF2`
`  </catch>`
`  </faultHandlers>`
`  <compensationHandler>`
`  activityC`
`  </compensationHandler>`
`  activity`
`</scope>`
	$[s_{activity}$
	$: \ \mathbf{catch}(\phi_{fault1})\{s_{activityF1}\}$
	$: \ \mathbf{catch}(\phi_{fault2})\{s_{activityF2}\}$
	$: \ s_{activityC}]_i$
	
	mapping in Table 3.13

`<process name="pname">`
`  <faultHandlers>`
`  <catch faultName="fault1">`
`  activityF1`
`  </catch>`
`  <catch faultName="fault2">`
`  activityF2`
`  </catch>`
`  </faultHandlers>`
`  activity`
`</process>`
	$*\,[k_{exit}, Vars]$
	$[s_{activity}$
	$: \ \mathbf{catch}(\phi_{fault1})\{s_{activityF1}\}$
	$: \ \mathbf{catch}(\phi_{fault2})\{s_{activityF2}\} : \mathbf{0}\,]_{pname}$
	
	Vars denotes the set of variables of `pname`
	
	mapping in Table 3.13

Table 3.19: Mapping of structured activities (cont.)

After a service instance is created as a result of consumption of the message $\langle c, id, data\rangle$ produced by the second invoke activity, the first produced message $\langle id\rangle$ can be consumed by the newly created service instance.

Multiple start and conflicting receive activities. A term corresponding to the multiple start activities in Figure 2.6(a) is:

$$
\begin{array}{l}
*\,[x_{cb}, x_{logID}, x_{info1}, x_{info2}, \mathtt{n}] \\
\quad (\,(\,LogOnServ_1 \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle.\,\mathtt{n}!\langle\rangle \\
\qquad \mid LogOnServ_2 \bullet LogOn?\langle x_{logID}, x_{info2}\rangle.\,\mathtt{n}!\langle\rangle\,) \\
\qquad \mid \mathtt{n}?\langle\rangle.\,\mathtt{n}?\langle\rangle.\,LogOnServ_1 \bullet RequestLogInfo?\langle x_{logID}\rangle. \\
\qquad\qquad x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info1}, x_{info2}\rangle\,)
\end{array}
$$

while that in Figure 2.6(b) can be modelled as follows:

$$
\begin{array}{l}
*\,[x_{cb}, x_{logID}, x_{info1}, x_{info2}, \mathtt{n}] \\
\quad (\,(\,LogOnServ_1 \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle.\,LogOnServ_2 \bullet LogOn?\langle x_{logID}, x_{info2}\rangle.\,\mathtt{n}!\langle\rangle \\
\qquad + LogOnServ_2 \bullet LogOn?\langle x_{logID}, x_{info2}\rangle.\,LogOnServ_1 \bullet LogOn?\langle x_{cb}, x_{logID}, x_{info1}\rangle.\,\mathtt{n}!\langle\rangle\,) \\
\qquad \mid \mathtt{n}?\langle\rangle.\,LogOnServ_1 \bullet RequestLogInfo?\langle x_{logID}\rangle.\,x_{cb} \bullet SendLogInfo!\langle x_{logID}, x_{info1}, x_{info2}\rangle\,)
\end{array}
$$

Scheduling of parallel activities. The example of Figure 2.8(a) in COWS can be rendered by the following term:

$$[x_1 = v_1] \mid [x_2 = v_2] \mid [x_3 = v_3]$$

The semantics of COWS prescribes that the three assignments can be executed in an unpredictable order that may change in different executions.

Forced termination. The example of Figure 2.9(a) in COWS can be rendered by the following term:

$$\{\!|\mathbf{kill}(k_{exit})|\!\} \mid [x_1 = v_1]\,;\,[x_2 = v_2]$$

while that of Figure 2.9(b) can be rendered by the following term:

$$\{\!|\mathbf{throw}(\phi)|\!\} \mid [x_1 = v_1]\,;\,[x_2 = v_2]$$

Operator $\{\!|\_|\!\}$ is used to distinguish *short-lived activities* from the other basic activities.

Eager execution of activities causing termination. The COWS terms are the same as the example before. To illustrate that activities <throw> and <exit> have higher priority than the remaining ones, consider the following term:

$$\{\!|\mathbf{throw}(\phi)|\!\} \mid \{\!|s_1|\!\}\,;\,\{\!|s_2|\!\} \mid p \bullet o?\langle x\rangle.\,s$$

for some short-lived activities s_1 and s_2. If the term is embedded in a proper context modelling an enclosing scope construct, by executing activity $\mathbf{throw}(\phi)$, it can only reduce to:

$$halt(\{\!|s_1|\!\}\,;\,\{\!|s_2|\!\}) \mid halt(p \bullet o?\langle x\rangle.\,s) \;\equiv\; \{\!|s_1|\!\}$$

In fact, due to definition of encoding (3.7), we get that $halt(\{\!|s_1|\!\}\,;\,\{\!|s_2|\!\}) = \{\!|s_1|\!\}$, as required.

Handlers protection. The following COWS term corresponds to the example illustrated in Figure 2.11:

$$[x]\ [\,(\ [\ [\,[x = b_1]\ :\ [x = b_2]\,]_{inner}\ ;\ \mathbf{throw}(\phi_1)\ :\ \mathbf{catch}(\phi_1)\{s_{FH1}\}\ :\ \mathbf{0}\,]_{outer} \\ \quad \mid \mathbf{if}\ (x)\ \mathbf{then}\ \{\mathbf{throw}(\phi_2)\}\,) \\ \quad :\ \mathbf{catch}(\phi_2)\{s_{FH2}\}\ :\ \mathbf{0}\,]_{main}$$

where three different gray tones are used to distinguish the three scope constructs.

3.4 Specification of the case studies

We present now the more relevant parts of the specifications in COWS of the automotive and finance case studies introduced in Sections 2.4.1 and 2.4.2, respectively. The complete specifications, written in CMC[6] 'machine readable' syntax, are reported in Appendix A.

3.4.1 Automotive case study

The COWS term modelling the overall scenario of Section 2.4.1 is:

$$[p_{car}]\,(\,\mathit{SensorsMonitor} \mid \mathit{GpsSystem} \mid \mathit{Discovery} \mid \mathit{Reasoner} \mid \mathit{Orchestrator}\,) \\ \mid \mathit{Bank} \mid \mathit{OnRoadRepairServices}$$

All services of the in-vehicle platform share a private partner name p_{car}, that is used for intra-vehicle communication and is passed to external services (e.g. the bank service) for receiving data from them.

When an engine failure occurs, a signal (raised by *SensorsMonitor*) triggers the execution of the *Orchestrator* and activates the corresponding 'recovery' service. *Orchestrator*, the most important component of the in-vehicle platform, is

$$[x_{carData}]\,(\,p_{car} \bullet o_{engineFailure}?\langle x_{carData}\rangle.s_{engfail} + p_{car} \bullet o_{lowOilFailure}?\langle x_{carData}\rangle.s_{lowoil} + \ldots\,)$$

This term uses the choice operator $\_+\_$ to pick one of those alternative recovery behaviours whose execution can start immediately. Notice that, while executing a recovery behaviour, *Orchestrator* does not accept other recovery requests. We are also assuming that it is reinstalled at the end of the recovery task.

The recovery behaviour $s_{engfail}$ executed when an engine failure occurs is

$$[p_{end}, o_{end}, x_{loc}, x_{list}, o_{undo}] \\ (\,[k]\,(\,\mathit{CardCharge} \mid \mathit{FindServices}\,) \mid p_{end} \bullet o_{end}?\langle\rangle.\, p_{end} \bullet o_{end}?\langle\rangle.\, \mathit{ChooseAndOrder}\,)$$

[6] CMC [192] is a tool supporting specification and verification of COWS terms. In particular, CMC supports model checking of SocL [83, 84] formulae and deriving all computations originating from a COWS term in an automated way. A prototypical version of CMC, developed by Franco Mazzanti at ISTI-CNR of Pisa, can be experimented via a web interface available at the address `http://fmt.isti.cnr.it/cmc/`. We refer to Sections 4.2 and Appendix A for more details about CMC and SocL.

$p_{end} \bullet o_{end}$ is a scoped endpoint along which successful termination signals (i.e. communications that carry no data) are exchanged to orchestrate execution of the different components. *CardCharge* corresponds to the homonymous UML action of Figure 2.13, while *FindServices* corresponds to the sequential composition of the UML actions RequestLocation and FindServices. The two terms are defined as follows:

$$
\begin{aligned}
\mathit{CardCharge} \triangleq\ & p_{bank} \bullet o_{charge}!\langle p_{car}, ccNum, amount, p_{car}\rangle \\
& \mid \{\!| p_{car} \bullet o_{chargeFail}?\langle p_{car}\rangle.\ \mathbf{kill}(k) \\
& \quad + p_{car} \bullet o_{chargeOK}?\langle p_{car}\rangle. \\
& \quad\ (\ p_{end} \bullet o_{end}!\langle\rangle \\
& \quad\quad \mid p_{car} \bullet o_{undo}?\langle cc\rangle.\ p_{car} \bullet o_{undo}?\langle cc\rangle.\ p_{bank} \bullet o_{revoke}!\langle p_{car}\rangle\ )\ |\!\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{FindServices} \triangleq\ & p_{car} \bullet o_{reqLoc}!\langle\rangle \\
& \mid p_{car} \bullet o_{respLoc}?\langle x_{loc}\rangle. \\
& \quad (\ p_{car} \bullet o_{findServ}!\langle x_{loc}, servicesType\rangle \\
& \quad\quad \mid p_{car} \bullet o_{found}?\langle x_{list}\rangle.\ p_{end} \bullet o_{end}!\langle\rangle \\
& \quad\quad + p_{car} \bullet o_{notFound}?\langle\rangle. \\
& \quad\quad\quad (\ \{\!| p_{car} \bullet o_{undo}!\langle cc\rangle \mid p_{car} \bullet o_{undo}!\langle cc\rangle |\!\} \mid \mathbf{kill}(k)\ )\ )
\end{aligned}
$$

Therefore, the recovery service concurrently contacts service *Bank*, to charge the driver's credit card with a security amount, and services *GpsSystem* and *Discovery*, to get the car's location (stored in x_{loc}) and a list of on road services (stored in x_{list}). When both activities terminate (the fresh endpoint $p_{end} \bullet o_{end}$ is used to appropriately synchronise their successful terminations), the recovery service forwards the obtained list to service *Reasoner*, that will choose the most convenient services (see definition of *ChooseAndOrder*). Whenever services finding fails, *FindServices* terminates the whole recovery behaviour (by means of the kill activity $\mathbf{kill}(k)$) and sends two signals *cc* (abbreviation of 'card charge') along the endpoint $p_{car} \bullet o_{undo}$. Similarly, if charging the credit card fails, then *CardCharge* terminates the whole recovery behaviour. Otherwise, it installs a compensation handler that takes care of revoking the credit card charge. Activation of this compensation activity requires two signals *cc* along $p_{car} \bullet o_{undo}$ and, thus, takes place either whenever *FindService* fails or, as we will see soon, whenever both garage and car rental orders fail.

ChooseAndOrder tries to order the selected services by contacting a car rental and, concurrently, a garage and a tow truck. It is defined as follows:

$$
\begin{aligned}
& [x_{gps}]\,(\ p_{car} \bullet o_{choose}!\langle x_{list}\rangle \\
& \qquad \mid\ [x_{garage}, x_{towTruck}, x_{rentalCar}]\ p_{car} \bullet o_{chosen}?\langle x_{garage}, x_{towTruck}, x_{rentalCar}\rangle. \\
& \qquad\qquad (\ \mathit{OrderGarageAndTowTruck} \mid \mathit{RentCar}\ )\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{OrderGarageAndTowTruck} \triangleq\ & [x_{garageInfo}] \\
& (\ x_{garage} \bullet o_{orderGar}!\langle p_{car}, x_{carData}\rangle \\
& \ \mid\ p_{car} \bullet o_{garageFail}?\langle\rangle. \\
& \qquad (\ p_{car} \bullet o_{undo}!\langle cc\rangle \mid [p,o]\,(p \bullet o!\langle x_{loc}\rangle \mid p \bullet o?\langle x_{gps}\rangle)\ ) \\
& \quad + p_{car} \bullet o_{garageOk}?\langle x_{gps}, x_{garageInfo}\rangle. \\
& \qquad (\ \mathit{OrderTowTruck} \\
& \qquad \mid\ p_{car} \bullet o_{undo}?\langle gar\rangle. \\
& \qquad\quad (\ x_{garage} \bullet o_{cancel}!\langle p_{car}\rangle \\
& \qquad\quad \mid\ p_{car} \bullet o_{undo}!\langle cc\rangle \mid p_{car} \bullet o_{undo}!\langle rc\rangle\ )\ )\ ) \\
\\
\mathit{OrderTowTruck} \triangleq\ & [x_{towInfo}] \\
& (\ x_{towTruck} \bullet o_{orderTow}!\langle p_{car}, x_{loc}, x_{gps}\rangle \\
& \ \mid\ p_{car} \bullet o_{towTruckFail}?\langle\rangle.\ p_{car} \bullet o_{undo}!\langle gar\rangle \\
& \quad + p_{car} \bullet o_{towTruckOK}?\langle x_{towInfo}\rangle\ ) \\
\\
\mathit{RentCar} \triangleq\ & [x_{rcInfo}] \\
& (\ x_{rentalCar} \bullet o_{orderRC}!\langle p_{car}, x_{gps}\rangle \\
& \ \mid\ p_{car} \bullet o_{rentalCarFail}?\langle\rangle.\ p_{car} \bullet o_{undo}!\langle cc\rangle \\
& \quad + p_{car} \bullet o_{rentalCarOK}?\langle x_{rcInfo}\rangle. \\
& \qquad p_{car} \bullet o_{undo}?\langle rc\rangle.\ x_{rentalCar} \bullet o_{redirect}!\langle p_{car}, x_{loc}\rangle\ )
\end{aligned}
$$

If ordering a garage fails, the compensation of the credit card charge is invoked by sending a signal cc along the endpoint $p_{car} \bullet o_{undo}$, and the car's location (stored in x_{loc}) is assigned to variable x_{gps} (whose value will be passed to the rental car service). This assignment is rendered in COWS as a communication along the private endpoint $p \bullet o$. Otherwise, the tow truck ordering starts and the garage's location is assigned to variable x_{gps}. Moreover, a compensation handler is installed; it will be activated whenever tow truck ordering fails and, in that case, attempts to cancel the garage order (by invoking operation o_{cancel}) and to compensate the credit card charge and the rental car order (by sending signal cc and rc along the endpoint $p_{car} \bullet o_{undo}$). Renting a car proceeds concurrently and, in case of successful completion, the compensation handler for the redirection of the rented car is installed; otherwise, the compensation of the credit card charge is invoked.

For the sake of presentation, we relegate the specification of the remaining components of the in-vehicle platform, i.e. *SensorsMonitor*, *GpsSystem*, *Discovery* and *Reasoner*, to Appendix A.2.

The COWS specification of the bank service is composed of two persistent subservices: *BankInterface*, that is publicly invocable by customers, and *CreditRating*, that instead is an 'internal' service that can only interact with *BankInterface*. Specifically, *Bank* is the COWS term

$$[o_{check}, o_{checkOK}, o_{checkFail}]\,(\ *\,\mathit{BankInterface} \mid *\,\mathit{CreditRating}\ )$$

where *BankInterface* and *CreditRating* are defined as follows:

$$\begin{aligned}
\mathit{BankInterface} \triangleq\ & [x_{cust}, x_{cc}, x_{amount}, x_{id}] \\
& p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle. \\
& (\ p_{bank} \bullet o_{check}!\langle x_{id}, x_{cc}, x_{amount}\rangle \\
& \quad \mid p_{bank} \bullet o_{checkFail}?\langle x_{id}\rangle.\ x_{cust} \bullet o_{chargeFail}!\langle x_{id}\rangle \\
& \quad +\ p_{bank} \bullet o_{checkOK}?\langle x_{id}\rangle. \\
& \qquad [k']\,(\ x_{cust} \bullet o_{chargeOK}!\langle x_{id}\rangle \mid p_{bank} \bullet o_{revoke}?\langle x_{id}\rangle.\ \mathbf{kill}(k')\ )\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{CreditRating} \triangleq\ & [x_{id}, x_{cc}, x_a] \\
& p_{bank} \bullet o_{check}?\langle x_{id}, x_{cc}, x_a\rangle. \\
& [p, o]\,(\ p \bullet o!\langle\rangle \mid p \bullet o?\langle\rangle.\ p_{bank} \bullet o_{checkOK}!\langle x_{id}\rangle \\
& \qquad\qquad +\ p \bullet o?\langle\rangle.\ p_{bank} \bullet o_{checkFail}!\langle x_{id}\rangle\ )
\end{aligned}$$

Whenever prompted by a client request, *BankInterface* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each instance forwards the request to *CreditRating*, by invoking the 'internal' operation o_{check} through the invoke activity $p_{bank} \bullet o_{check}!\langle x_{id}, x_{cc}, x_{amount}\rangle$, then waits for a reply on one of the other two internal operations $o_{checkOK}$ and $o_{checkFail}$, by exploiting the receive-guarded choice operator, and finally sends the reply back to the client by means of a final invoke activity using the partner name of the client stored in the variable x_{cust}. In case of a positive answer, the possibility of revoking the request through invocation of operation o_{revoke} is enabled (in fact, should the other request fail, the customer charge operation should be cancelled in order to implement the wanted transactional behaviour). Revocation causes deletion of the reply to the client, if this has still to be performed. Notably, if an invocation along the endpoints $p_{bank} \bullet o_{checkOk}$, $p_{bank} \bullet o_{checkFail}$ or $p_{bank} \bullet o_{revoke}$ takes place after a certain number of service instances have been created, then it could be received by any of these instances. Hence, to synchronise with the proper instance, an appropriate customer datum stored in the variable x_{id} is exploited as a correlation value. Service *CreditRating* takes care of checking clients' requests and decides if they can be authorised or not. For the sake of simplicity, the choice between approving or not a request is left here completely non-deterministic.

OnRoadRepairServices is actually a composition of various on road services, i.e. it is

$$\mathit{Garage}_1 \mid \mathit{Garage}_2 \mid \mathit{TowTruck}_1 \mid \mathit{TowTruck}_2 \mid \mathit{RentalCar}_1 \mid \mathit{RentalCar}_2$$

Such concurrent on road services are all modelled in a similar way, e.g.

$$\begin{aligned}
\mathit{Garage}_i \triangleq\ & *\,[x_{cust}, x_{sensorsData}, o_{checkOK}, o_{checkFail}] \\
& p_{garage_i} \bullet o_{orderGar}?\langle x_{cust}, x_{sensorsData}\rangle. \\
& (\ p_{garage_i} \bullet o_{checkOK}!\langle\rangle \mid p_{garage_i} \bullet o_{checkFail}!\langle\rangle \\
& \quad \mid p_{garage_i} \bullet o_{checkFail}?\langle\rangle.\ x_{cust} \bullet \mathit{garageFail}!\langle\rangle \\
& \quad + p_{garage_i} \bullet o_{checkOK}?\langle\rangle. \\
& \qquad [k]\,(\ x_{cust} \bullet \mathit{garageOK}!\langle \mathit{garageGPS}, \mathit{garageInfo}\rangle \\
& \qquad\quad \mid p_{garage_i} \bullet o_{cancel}?\langle x_{cust}\rangle.\ \mathbf{kill}(k)\ )\ )
\end{aligned}$$

For simplicity, success or failure of garage orders are modelled by means of non-deterministic choice by exploiting internal operations $o_{checkOK}$ and $o_{checkFail}$.

3.4.2 Finance case study

The COWS term representing the overall scenario of Section 2.4.2 is

$$[key]\,(\,Customer \mid CreditInstitute\,) \mid Validation \mid Employee_1 \mid \ldots \mid Employee_n \mid Supervisor_1 \mid \ldots \mid Supervisor_m$$

The delimitation $[key]$ is used to declare that *key* is a shared element known to *Customer* and *CreditInstitute*, and only to them.

CreditInstitute is defined as follows

$$\begin{array}{l}[createInst, reqProcessing, reqUpdate, contractProcessing]\\ (\,[authentication, notAuthorized, authorized]\,(\,Portal \mid Authentication\,)\\ \quad \mid InformationUpload \mid InformationUpdate \mid RequestProcessing\\ \quad \mid ContractProcessing \mid EmployeeTaskList \mid SupervisorTaskList\,)\end{array}$$

The term is the parallel composition of the (considered) subservices of the credit institute: the behaviour of *Portal*, *InformationUpload*, *InformationUpdate* and *RequestProcessing* is graphically represented by the UML activity diagrams depicted in Figures 2.15, 2.16, 2.17 and 2.18; *Authentication* and *ContractProcessing* appear as external services in Figures 2.15 and 2.18, respectively.

The delimitation operator ensures that operations *authentication*, *notAuthorized* and *authorized* are used to communicate only by *Portal* and *Authentication*, while operations *createInst*, *reqProcessing*, *reqUpdate* and *contractProcessing* can also be used by the other subservices. This guarantees that external services cannot interfere with the credit portal during the login and instantiation phases.

Service *Portal* is publicly invocable and can interact with customers other than with the 'internal' services of the credit institute. *Portal* is defined as follows:

$$\begin{array}{l}*\,[x_{user}, x_{pwd}, x_{cust}]\\ \quad portal \bullet login?\langle x_{user}, x_{pwd}, x_{cust}\rangle.\\ \quad (\,portal \bullet authentication!\langle x_{user}, x_{pwd}\rangle\\ \qquad \mid\; portal \bullet notAuthorized?\langle x_{user}\rangle.\; x_{cust} \bullet failedLogin!\langle key\rangle\\ \qquad +\; portal \bullet authorized?\langle x_{user}\rangle.\\ \qquad\quad [sessionID]\,(\,x_{cust} \bullet logged!\langle key, sessionID\rangle\\ \qquad\qquad\qquad \mid\; portal \bullet creditRequest?\langle sessionID\rangle.\\ \qquad\qquad\qquad\qquad\quad portal \bullet createInst!\langle sessionID\rangle\\ \qquad\qquad\qquad +\; portal \bullet bankTransferRequest?\langle sessionID\rangle.\;\ldots\\ \qquad\qquad\qquad +\; \ldots \text{other services provided by the credit portal} \ldots\,)\,)\end{array}$$

The replication operator is exploited to model the fact that *Portal* can create multiple instances to serve several customer requests simultaneously. Each interaction with the portal starts with a receive activity of the form $portal \bullet login?\langle x_{user}, x_{pwd}, x_{cust}\rangle$ corresponding

to reception of a request emitted by a customer. The receive activity initializes the variables x_{user}, x_{pwd} and x_{cust}, declared local to *Portal* by the delimitation operator, with data provided by a customer. Whenever prompted by a customer request, *Portal* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each instance forwards the request to *Authentication*, by invoking the 'internal' operation *authentication* through the invoke activity $portal \cdot authentication!\langle x_{user}, x_{pwd}\rangle$, and waits for a reply on one of the other two internal operations *notAuthorized* and *authorized*, by exploiting the receive-guarded choice operator. In case of a positive answer, by means of the delimitation operator, a fresh session identifier *sessionID* is generated, and a reply is sent back to the customer by means of an invoke activity using the partner name of the customer stored in the variable x_{cust}. Moreover, by using the choice operator again, *Portal* allows the customer to choose among several services (however, only the credit request service is actually modelled). Whenever the customer selects this service, *InformationUpload* is instantiated through invocation of the private operation *createInst*. Notably, the identifier *sessionID* is passed both to *Customer* and to the created instance of *InformationUpload*, to allow them to safely communicate. In fact, in each interaction between *Customer* and the instances of the credit institute subservices, the identifier is used as a correlation datum, i.e. it appears within each message. Pattern-matching permits locating such datum in the messages and, therefore, delivering them to the instances identified by the same datum.

InformationUpload is defined as follows:

$$
\begin{array}{l}
*\,[x_{id}]\ portal \cdot createInst?\langle x_{id}\rangle. \\
\quad [k, fault, abort]\,(\\
\quad\quad [k_{abortFault}]\,(\\
\quad\quad\quad [x_{custData}, x_{secData}, x_{final\_balance}, x_{amount}, x_{cust}] \\
\quad\quad\quad portal \cdot getCreditRequest?\langle x_{id}, x_{custData}, x_{amount}, x_{cust}\rangle. \\
\quad\quad\quad\quad [\mathtt{end}] \\
\quad\quad\quad\quad (\ portal \cdot securities?\langle x_{id}, x_{secData}\rangle.\ \mathtt{end}!\langle\rangle \\
\quad\quad\quad\quad\quad \mid\ [\mathtt{repeat}] \\
\quad\quad\quad\quad\quad\quad (\ \mathtt{repeat}!\langle\rangle \\
\quad\quad\quad\quad\quad\quad\quad \mid\ *\,\mathtt{repeat}?\langle\rangle. \\
\quad\quad\quad\quad\quad\quad\quad\quad [x_{balance}]\ portal \cdot balance?\langle x_{id}, x_{balance}\rangle. \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\ validation \cdot validateBalance!\langle x_{id}, portal, x_{balance}\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid\ portal \cdot validateBalance?\langle x_{id}, no\rangle. \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (x_{cust} \cdot balanceNotValid!\langle x_{id}\rangle \mid \mathtt{repeat}!\langle\rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad + portal \cdot validateBalance?\langle x_{id}, yes\rangle.\ \mathtt{end}!\langle x_{balance}\rangle\ )\ ) \\
\quad\quad\quad\quad\quad \mid\ \mathtt{end}?\langle\rangle.\ \mathtt{end}?\langle x_{final\_balance}\rangle. \\
\quad\quad\quad\quad\quad\quad (\mathbf{kill}(k) \mid \{\!|portal \cdot reqProcessing!\langle x_{id}, x_{custData}, x_{secData}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad x_{final\_balance}, x_{amount}, x_{cust}\rangle|\!\}) \\
\quad\quad\quad\quad \mid\ portal \cdot cancel?\langle x_{id}\rangle.\ (\mathbf{kill}(k_{abortFault}) \mid \{\!|fault \cdot abort!\langle\rangle|\!\})\ ) \\
\quad\quad \mid\ fault \cdot abort?\langle\rangle.\ \mathbf{0}\ )
\end{array}
$$

Each instance of *InformationUpload* is created to serve a customer request identified by a specific session identifier. Once created, an instance is actually activated by *Customer* by invoking operation *getCreditRequest* and transmitting the credit request application data (i.e. name, address, desired credit amount, ...). Then, two activities are concurrently executed: (1) additional security data are received, and (2) balance information are received and forwarded to *Validation* for checking consistency and validation; if the verification was negative (i.e. the second argument of operation *validateBalance* is *no*), then *Customer* is informed and (2) is repeated. Notice that, endpoints `end` and `repeat` have been exploited to model sequential and repeat loop constructs. When (1) and (2) terminate successfully, *RequestProcessing* is instantiated, by invoking the (private) operation *reqProcessing* and initialising the created instance with all the request data (i.e. customer data, amount, security data and balance information). Notably, at any time after the login *Customer* can require the cancellation of the credit request processing by invoking operation *cancel* using the identifier. This causes the forced termination of all unprotected parallel activities, through the execution of the activity **kill**(k_{abort}), and the emission of an (internal) fault signal $fault \bullet abort!\langle\rangle$. To deal with such faults, each instance has a specific fault handler, that catches a fault and does nothing. If an instance completes successfully, then its fault handler is removed by executing the activity **kill**(k).

RequestProcessing is defined as follows:

$$
\begin{array}{l}
* \ [x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{amount}, x_{cust}] \\
\quad portal \bullet reqProcessing?\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{amount}, x_{cust}\rangle . \\
\quad [k, fault, abort, undo] \, (\\
\qquad [k_{abortFault}] \, (\\
\qquad\quad portal \bullet addToETL!\langle x_{id}, x_{secData}, x_{balance}, x_{amount}\rangle \\
\qquad\quad | \ portal \bullet taskAddedToETL?\langle x_{id}\rangle . \\
\qquad\qquad (\{\!| portal \bullet undo?\langle empTaskList\rangle . \ portal \bullet removeTaskETL!\langle x_{id}\rangle |\!\} \\
\qquad\qquad\quad | \ EmployeeEval \,) \\
\qquad\quad | \ portal \bullet cancel?\langle x_{id}\rangle . \, (\mathbf{kill}(k_{abortFault}) \mid \{\!| fault \bullet abort!\langle\rangle |\!\}) \,) \\
\qquad | \ fault \bullet abort?\langle\rangle . \\
\qquad\quad (portal \bullet undo!\langle empTaskList\rangle \mid portal \bullet undo!\langle supTaskList\rangle) \,)
\end{array}
$$

When an instance of *RequestProcessing* is created, all the data relevant for computing the rating are inserted in the *EmployeeTaskList*, through invocation of operation *addToETL*. When an acknowledgment from *EmployeeTaskList* is received, a compensation handler for undoing the insertion activity is installed, and the instance is blocked waiting for the employee evaluation (term *EmployeeEval*). The compensation handler is a protected term waiting for a compensation request, i.e. a signal *empTaskList* along $portal \bullet undo$. When this signal is received, the compensation handler becomes active and, to compensate the insertion activity, invokes operation *removeTaskETL* provided by *EmployeeTaskList*. Compensation is activated by the body of the fault handler, that sends the two compensation signals *empTaskList* and *supTaskList* (corresponding to action ≪Compensate All≫ of Figure 2.18).

The term *EmployeeEval* is

$$
\begin{array}{l}
[x_{rating}, x_{info}, x_{decision}] \\
portal \bullet empEvaluation?\langle x_{id}, x_{rating}, x_{info}, x_{decision}\rangle. \\
[cond, choice]\,(cond \bullet choice!\langle x_{decision}\rangle \\
\quad \mid\ cond \bullet choice?\langle no\rangle.\,(\mathbf{kill}(k) \mid \{\!|x_{cust} \bullet negativeResp!\langle x_{id}, x_{info}\rangle|\!\}) \\
\quad\ + cond \bullet choice?\langle update\rangle. \\
\quad\quad (\mathbf{kill}(k) \mid \{\!|portal \bullet reqUpdate!\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{amount}, x_{cust}, x_{info}\rangle|\!\}) \\
\quad\ + cond \bullet choice?\langle yes\rangle. \\
\quad\quad (\ portal \bullet addToSTL!\langle x_{id}, x_{secData}, x_{balance}, x_{amount}, x_{info}\rangle \\
\quad\quad\ \mid\ portal \bullet taskAddedToSTL?\langle x_{id}\rangle. \\
\quad\quad\quad (\{\!|portal \bullet undo?\langle supTaskList\rangle.\ portal \bullet removeTaskSTL!\langle x_{id}\rangle|\!\} \\
\quad\quad\quad\ \mid\ SupervisorEval\ )\ )
\end{array}
$$

It receives the employee evaluation and performs a choice on the basis of the value stored in the variable $x_{decision}$, that can be either *no* (i.e. the credit request is rejected), or *update* (i.e. the customer is asked to update the desired amount and/or the security data), or *yes* (i.e. the employee accepts the request). Conditional choice is modelled in a natural way by a choice among three receives along the private endpoint $cond \bullet choice$, and by relying on pattern-matching. In case of *no* and *update*, the instance is halted (by means of a kill activity), while in case of acceptance the request data are inserted in the supervisor task list, the corresponding compensation handler (i.e. the protected term) is installed, and the instance is blocked waiting for the supervisor evaluation (term *SupervisorEval*).

Finally, *SupervisorEval* is

$$
\begin{array}{l}
[x_{offer}, x_{motivation}, x_{supDecision}] \\
portal \bullet supEvaluation?\langle x_{id}, x_{offer}, x_{motivation}, x_{supDecision}\rangle. \\
[cond, choice]\,(cond \bullet choice!\langle x_{supDecision}\rangle \\
\quad \mid\ cond \bullet choice?\langle no\rangle.\,(\mathbf{kill}(k) \mid \{\!|x_{cust} \bullet negativeResp!\langle x_{id}, x_{motivation}\rangle|\!\}) \\
\quad\ + cond \bullet choice?\langle update\rangle. \\
\quad\quad (\mathbf{kill}(k) \mid \{\!|portal \bullet reqUpdate!\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, \\
\quad\quad\quad\quad x_{amount}, x_{cust}, x_{motivation}\rangle|\!\}) \\
\quad\ + cond \bullet choice?\langle yes\rangle. \\
\quad\quad (\ x_{cust} \bullet offer!\langle x_{id}, x_{offer}, x_{motivation}\rangle \\
\quad\quad\ \mid\ portal \bullet answer?\langle x_{id}, yes\rangle. \\
\quad\quad\quad (\mathbf{kill}(k) \mid \{\!|portal \bullet contractProcessing!\langle x_{id}, x_{custData}, x_{secData}, \\
\quad\quad\quad\quad x_{balance}, x_{amount}, x_{cust}, x_{rating}, x_{info}, x_{offer}, x_{motivation}\rangle|\!\}) \\
\quad\quad\ + portal \bullet answer?\langle x_{id}, no\rangle.\mathbf{kill}(k)\ )\ )
\end{array}
$$

The above term behaves similarly to *EmployeeEval* except for the case of positive evaluation, for which it sends an offer to the customer and, in case of acceptance, forwards all the information to *ContractProcessing*.

The remaining terms composing the scenario are reported in Appendix A.3. In particular, the task list services *EmployeeTaskList* and *SupervisorTaskList* could be modelled

in different ways, according to the underlying data structures and the properties that they enjoy. For simplicity sake, in the specification reported in the appendix, task lists do not preserve the arrival order of requests that are then withdrawn in a non-deterministic way.

3.5 Concluding remarks

We have introduced COWS, a formalism for specifying and combining services, while modelling their dynamic behaviour (i.e. it deals with service orchestration rather than choreography). COWS borrows many constructs from well-known process calculi, e.g. π-calculus, update calculus, StAC$_i$, and Lπ, but combines them in an original way, thus being different from all existing calculi. COWS permits modelling different and typical aspects of (web) services technologies, such as multiple start activities, receive conflicts, delivering of correlated messages, service instances and interactions among them. As a further evidence of COWS's suitability for modelling SOC applications, we have also presented the close correspondence between WS-BPEL activities and COWS terms.

Since its definition, however, some linguistic variants of COWS have been introduced to model timed activities [133] (presented in Section 5.2.1) and dynamic service discovery and negotiation [136] (presented in Section 5.2.2), thus obtaining a linguistic formalism capable of modelling all the phases of the life cycle of service-oriented applications. A number of methods and tools have also been devised to analyse COWS specifications, such as the stochastic extension and the BPMN-based notaion defined in [172, 173] to enable quantitative reasoning on service behaviours, the type system introduced in [132] to check confidentiality properties, the logic and model checker presented in [83, 84] to express and check functional properties of services, the bisimulation-based observational semantics defined in [174] to check interchangeability of services and conformance against service specifications, and the symbolic characterisation of the operational semantics of COWS presented in [175] to avoid infinite representations of COWS terms due to the value-passing nature of communication. In the next chapter, we present some of the tools mentioned above and illustrate the classes of properties that can be analysed by using them.

Let us comment on related work. The correlation mechanism was first exploited in [200], that, however, only considers interaction among different instances of a single business process. Instead, to connect the interaction protocols of clients and of the respective service instances, SCC [34] and CaSPiS [35] rely on the explicit modelling of sessions and their dynamic creation (that exploits the mechanism of private names of π-calculus). Interaction sessions are not explicitly modelled in COWS, instead they can be identified by tracing all those exchanged messages that are correlated each other through their same contents (as in [104]). We believe that the mechanism based on correlation sets (also used by WS-BPEL), that exploits business data and communication protocol headers to correlate different interactions, is more robust and fits the loosely coupled world of Web Services better than that based on explicit session references.

Many works put forward enrichments of some well-known process calculus with constructs inspired by those of WS-BPEL. The most of them deal with issues of web transactions such as interruptible processes, failure handlers and time. This is, for example, the case of [127, 128, 143, 144] that present timed and untimed extensions of the π-calculus, called $\texttt{web}\pi$ and $\texttt{web}\pi_\infty$, tailored to study a simplified version of the scope construct of WS-BPEL. Other proposals on the formalization of flow compensation are [46, 44] that give a more compact and closer description of the Sagas mechanism [98] for dealing with long running transactions, while some other works [127, 144] have concentrated on modelling web transactions and on studying their properties in programming languages based on the π-calculus.

Many efforts have been devoted to develop analytical tools for SOC foundational languages. Some of these works study mechanisms for comparing global descriptions (i.e. choreographies) and local descriptions (i.e. orchestrations) of a same system. Means to check conformance of these different views have been defined in [49, 50] and, by relying on session types, in [56]. COWS, instead, only considers service orchestration and focuses on modelling the dynamic behaviour of services without the limitations possibly introduced by a layer of choreography. Other approaches are based on the use of schema languages [63] and Petri nets [109]. In [125] a sort of distributed input-guarded choice of join patterns, called *smooth orchestrators*, gives a simple and effective representation of synchronization constructs. A type system specifying security policies for orchestration has been introduced in [14] for a very basic formalism based on the λ-calculus. Finally, a type system for checking compliance between (simplified) WS-BPEL terms and the associated WSDL documents has been defined in [130].

The work presenting the formalism closest to COWS is [130], where WS-CALCULUS is introduced to formalize the semantics of WS-BPEL. COWS represents a more foundational formalism than WS-CALCULUS in that it does not rely on explicit notions of location and state, it is more manageable (e.g. has a simpler operational semantics) and, at least, equally expressive (as the encoding of WS-CALCULUS in COWS shows, Section 5.1.3).

Chapter 4

Analysis techniques for COWS specifications

The previous chapter has introduced the formal language COWS we designed for specifying SOC applications. This chapter, instead, is devoted to illustrate some methods and tools we have devised to analyse COWS terms.

As a first analysis technique, we introduce a type system for checking confidentiality properties: types are used to express and enforce policies for regulating the exchange of data among services. Then, we present a logical verification methodology for checking functional properties of SOC systems specified using COWS. The properties are described by means of a branching-time temporal logic specifically designed to express in a convenient way peculiar aspects of services. Thereafter, we introduce a bisimulation-based observational semantics for COWS. Specifically, we define notions of (strong and weak) open barbed bisimilarities and prove their coincidence with more manageable characterisations in terms of labelled bisimilarities. Finally, we define a symbolic characterisation of the operational semantics of COWS to avoid infinite representations of COWS terms due to the value-passing nature of communication.

Structure of the chapter. The rest of the chapter is organized as follows. Section 4.1 introduces a typed variant of COWS for regulating data exchange and presents type safety and subject reduction. Section 4.2 presents the logic SocL and the verification methodology for checking SocL formulae over COWS specifications. Section 4.3 introduces a bisimulation theory for COWS. Section 4.4 describes a symbolic semantics for COWS. At the end of each section, a comparison with related work is reported.

4.1 A type system for checking confidentiality properties

In this section, we present a typed variant of COWS that permits expressing and forcing policies regulating the exchange of data among interacting services and ensuring that, in

that respect, services do not manifest unexpected behaviours. Programmers can indeed settle the partners usable to exchange any given datum (and, then, the services that can share it), thus avoiding the datum be accessed (by unwanted services) through unauthorized partners. The (static and dynamic) language semantics then guarantees that well-typed services always comply with the constraints expressed by the type associated to each single datum. This enables us to check confidentiality properties, e.g., that critical data such as credit card information are shared only with authorized partners.

4.1.1 Static and dynamic semantics of typed COWS

The type system we present in this section permits to express and enforce policies for regulating the exchange of data among services. To implement such policies, programmers can annotate data with sets of partner names characterizing the services authorized to use and exchange them; these sets are called *regions*. The language operational semantics uses these annotations to guarantee that computations proceed according to them. This property, called *soundness*, can be stated as follows

> A service s is *sound* if, for any datum v in s associated to region r and for all evolutions of s, it holds that v can be exchanged only by using partners in r.

To facilitate the task of decorating COWS terms with type annotations, we let the type system partially infer such annotations *à la* ML: service programmers explicitly write only the annotations necessary to specify the wanted policies for communicable data; then, a type inference system (statically) performs some coherence checks (e.g. the partner used by an invoke must belong to the regions of all data occurring in the argument of the activity) and derives the minimal region annotations for variable declarations that ensure consistency of services initial configuration. This allows us to define an *operational semantics with types* [102] which is simpler than a full-fledged *typed operational semantics*, because it only performs simple checks (i.e. subset inclusion) using region annotations to authorize or block transitions. Our main results prove that the type system and the operational semantics are sound. As a consequence, we have that services always comply with the constraints expressed by the type of each single datum.

We present now the syntax of COWS extended with regions, the type inference system and the operational semantics for typed COWS terms.

4.1.1.1 Syntax

We will consider here *raw* services, namely COWS terms written according to the syntax in Table 4.1. Such syntax differs from that introduced in the previous chapter only for the presence of data annotations (i.e. regions) in the invoke activities. Intuitively, raw services only contain those region annotations that implement the policies for data exchange settled by the service programmers.

s	::=	(services)	g	::=	(receive-guarded choice)
	$\mathbf{kill}(k)$	(kill)		$\mathbf{0}$	(nil)
|	$u \bullet u'!\overline{\{\epsilon\}_r}$	(invoke)	|	$p \bullet o?\bar{w}.s$	(request processing)
|	g	(receive-guarded choice)	|	$g + g$	(choice)
|	$s \mid s$	(parallel composition)			
|	$\{\!\{s\}\!\}$	(protection)			
|	$[e]\, s$	(delimitation)			
|	$* s$	(replication)			

Table 4.1: COWS syntax (*raw* services)

Regions can be either finite subsets of partners and variables or the distinct element $\top$ (denoting the universe of partners). The set of all regions, ranged over by r, is partially ordered by the subset inclusion relation $\subseteq$, and has $\top$ as top element. An expression ϵ tagged with region r will be written as $\{\epsilon\}_r$; an untagged ϵ will stand for $\{\epsilon\}_\top$. We will write $\epsilon(\bar{x})$ to make explicit all the variables $\bar{x}$ occurring in ϵ (we still write ϵ when this information is not needed), and $\bar{\epsilon}$ (resp. $\bar{r}$) to denote the tuple of the expressions (resp. regions) occurring in $\overline{\{\epsilon\}_r}$.

In the sequel, we shall denote by $\mathrm{fe}(t)$ (resp. $\mathrm{be}(t)$) the set of elements that occur free (resp. bound) in a term t, by $\mathrm{fv}(t)$ (resp. $\mathrm{bv}(t)$) the set of free (resp. bound) variables in t, and by $\mathrm{fk}(t)$ the set of free killer labels in t. For simplicity sake, we assume that bound variables in services are pairwise distinct (of course, this condition is not restrictive and can always be fulfilled by possibly using α-conversion).

4.1.1.2 A type inference system

The annotations put by the type inference are written as superscripts, to better distinguish them from those put by the programmers. Thus, the syntax of variable delimitation becomes $[\{x\}^r]\, s$, which means that the datum that dynamically will replace x will be used at most by the partners in r. *Typed* COWS services have the same syntax of raw services except for the presence of region annotations on variable declarations. Typed services are then generated by the syntax in Table 4.1 where, differently from Section 4.1.1.1, e ranges over killer labels, names and annotated variables as $\{x\}^r$. Notably, types may *depend* on partner variables, i.e. on parameters of receiving activities; during computation, they are therefore affected by application of substitutions that replace partner variables with partner names. We assume that the region of a partner name always contains, at least implicitly, such partner.

The type inference system is presented in Table 4.2. Typing judgements are written $\Gamma \vdash s \succ \Gamma' \vdash s'$, where the type environment Γ is a finite function from variables to regions such that $\mathrm{fv}(s) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{bv}(s) \cap \mathrm{dom}(\Gamma) = \emptyset$ (the same holds for Γ' and s'). Type environments are written as sets of pairs of the form $x : r$, where x is a partner variable and r is its assumed region annotation. The domain of an environment is defined

$$\Gamma \vdash \mathbf{0} \succ \Gamma \vdash \mathbf{0} \quad \textit{(t-nil)} \qquad \Gamma \vdash \mathbf{kill}(k) \succ \Gamma \vdash \mathbf{kill}(k) \quad \textit{(t-kill)}$$

$$\frac{\forall\, r' \in \{r_i\}_{i\in\{1,..,n\}} \quad u_1 \in r'}{\Gamma \vdash u_1 \bullet u_2!\langle\{\epsilon_1(\bar{y}_1)\}_{r_1}, \ldots, \{\epsilon_n(\bar{y}_n)\}_{r_n}\rangle \succ (\Gamma + \{x : r_1\}_{x\in\bar{y}_1} + \ldots + \{x : r_n\}_{x\in\bar{y}_n}) \vdash u_1 \bullet u_2!\langle\{\epsilon_1(\bar{y}_1)\}_{r_1}, \ldots, \{\epsilon_n(\bar{y}_n)\}_{r_n}\rangle} \quad \textit{(t-inv)}$$

$$\frac{\Gamma + \{x : \{p\}\}_{x\in \mathrm{fv}(\bar{w})} \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash p \bullet o?\bar{w}.s \succ \Gamma' \vdash p \bullet o?\bar{w}.s'} \quad \textit{(t-rec)}$$

$$\frac{\Gamma \vdash g_1 \succ \Gamma_1 \vdash g_1' \qquad \Gamma \vdash g_2 \succ \Gamma_2 \vdash g_2'}{\Gamma \vdash g_1 + g_2 \succ \Gamma_1 + \Gamma_2 \vdash g_1' + g_2'} \quad \textit{(t-sum)}$$

$$\frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash \{\!|s|\!\} \succ \Gamma' \vdash \{\!|s'|\!\}} \quad \textit{(t-prot)} \qquad \frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash *s \succ \Gamma' \vdash *s'} \quad \textit{(t-repl)}$$

$$\frac{\Gamma \vdash s \succ \Gamma' \vdash s' \qquad n \notin \mathrm{reg}(\Gamma')}{\Gamma \vdash [n]\, s \succ \Gamma' \vdash [n]\, s'} \quad \textit{(t-del}_{name}\textit{)} \qquad \frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash [k]\, s \succ \Gamma' \vdash [k]\, s'} \quad \textit{(t-del}_{lab}\textit{)}$$

$$\frac{\Gamma, \{x : \emptyset\} \vdash s \succ \Gamma', \{x : r\} \vdash s' \qquad x \notin \mathrm{reg}(\Gamma')}{\Gamma \vdash [x]\, s \succ \Gamma' \vdash [\{x\}^{r-\{x\}}]\, s'} \quad \textit{(t-del}_{var}\textit{)}$$

$$\frac{\Gamma \vdash s_1 \succ \Gamma_1 \vdash s_1' \qquad \Gamma \vdash s_2 \succ \Gamma_2 \vdash s_2'}{\Gamma \vdash s_1 \mid s_2 \succ \Gamma_1 + \Gamma_2 \vdash s_1' \mid s_2'} \quad \textit{(t-par)}$$

Table 4.2: Type inference system

as usual: $\mathrm{dom}(\emptyset) = \emptyset$ and $\mathrm{dom}(\Gamma, \{x : r\}) = \mathrm{dom}(\Gamma) \cup \{x\}$, where ',' denotes union between environments with disjoint domains. The *region* of Γ is the union of the regions in Γ, i.e. $\mathrm{reg}(\emptyset) = \emptyset$ and $\mathrm{reg}(\Gamma, \{x : r\}) = r \cup \mathrm{reg}(\Gamma)$. We will write $\Gamma + \Gamma'$ to denote the environment obtained by extending Γ with Γ'; $+$ is inductively defined by

$$\begin{aligned}
&\Gamma + \emptyset = \Gamma \\
&\Gamma + \{x : r\} = \begin{cases} \Gamma', \{x : r \cup r'\} & \text{if } \Gamma = \Gamma', \{x : r'\} \\ \Gamma, \{x : r\} & \text{otherwise} \end{cases} \\
&\Gamma + (\{x : r\}, \Gamma') = (\Gamma + \{x : r\}) + \Gamma'
\end{aligned}$$

Hence, the judgement $\emptyset \vdash s \succ \emptyset \vdash s'$ can be derived only if s is a closed raw service (because the initial environment is empty); if it is derivable, then s' is the typed service obtained by decorating s with the region annotations describing the use of each variable of s in its scope. Type inference determines such regions by considering the invoking and receiving partners where the variables occur.

We now comment on the most significant typing rules. Rule *(t-inv)* checks if the in-

voked partner u_1 belongs to the regions of the communicated data. If it succeeds, the type environment Γ is extended by associating a proper region to each variable used in the expressions argument of the invoke activity. Rule *(t-rec)* tries to type s in the type environment Γ extended by adding the receiving partner to the regions of the variables in $\bar{w}$. Rules *(t-sum)* and *(t-par)* yield the same typing; this is due to the sharing of variables. For instance, service $[x]\,(p \bullet o?\langle x\rangle \mid p' \bullet o'!\langle\{x\}_r\rangle)$ with $p' \in r$ is annotated as $[\{x\}^{r'}]\,(p \bullet o?\langle x\rangle \mid p' \bullet o'!\langle\{x\}_r\rangle)$ with $r' = (\{p\} \cup r - \{x\})$. In rule *(t-del$_{name}$)*, premise $n \notin \mathrm{reg}(\Gamma')$ prevents a new name n to escape from its binder $[n]$ in the inference. As an example, consider the closed raw service

$$[z]\,p \bullet o?\langle z\rangle \,.\, [p']\,p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle \tag{4.1}$$

Without the premise $n \notin \mathrm{reg}(\Gamma')$, the service resulting from the type inference would be $[\{z\}^{\{p'',p',p\}}]\,p \bullet o?\langle z\rangle \,.\, [p']\,p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle$. The problem with this service is that the name p' occurring in the annotation associated to z by the inference system escapes from the scope of its binder and, thus, represents a completely different name. Although, service (4.1) is not typable, by a simple semantics preserving manipulation one can get a typable service as, e.g., the following one $[p']\,[z]\,p \bullet o?\langle z\rangle \,.\, p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle$.

Similarly, in rule *(t-del$_{var}$)*, premise $x \notin \mathrm{reg}(\Gamma')$ prevents initially closed services to become open at the end of the inference. Otherwise, e.g., the type inference would transform the closed raw service

$$[x]\,p \bullet o?\langle x\rangle \,.\, [y]\,p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle \tag{4.2}$$

into the open service $[\{x\}^{\{p,p'',y\}}]\,p \bullet o?\langle x\rangle \,.\, [\{y\}^{\{p'\}}]\,p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle$. Also in this case, we can easily modify the untypable service (4.2) to get a typable one with a similar semantics like, e.g., the service $[y]\,[x]\,p \bullet o?\langle x\rangle \,.\, p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle$.

Furthermore, in *(t-del$_{var}$)*, x is annotated with $r - \{x\}$, rather than with r, otherwise initially closed services could become open. E.g., the closed raw service $[x]\,p \bullet o?\langle x\rangle \,.\, p' \bullet o'!\langle\{x\}_{\{p',x\}}\rangle$ would be transformed into the open service $[\{x\}^{\{p,p',x\}}]\,p \bullet o?\langle x\rangle \,.\, p' \bullet o'!\langle\{x\}_{\{p',x\}}\rangle$ (indeed, x occurs in the annotation associated to its declaration). Notice that, although the region associated to x by the inference does never record that a service possibly transmits x with regions containing x, rule *(t-del$_{var}$)* is sound because we assumed that the region of a partner name, at least implicitly, contains the partner name.

Definition 4.1.1 *A service s is* well-typed *if $\emptyset \vdash s' \succ \emptyset \vdash s$ for some raw service s'.*

4.1.1.3 Operational semantics

The structural congruence $\equiv$ remains the same of the original Tables 3.2 and 3.10 except for laws in Table 4.3. All the presented (extended) laws are straightforward and describe the interplay between delimited names/variables and region annotations.

The modified rules of the labelled transition relation $\xrightarrow{\alpha}$ are shown in Table 4.4 (the remaining ones are those of Table 3.12) which exploit the same predicates and auxiliary

$$
\begin{array}{rcll}
[e_1]\,[e_2]\,s & \equiv & [e_2]\,[e_1]\,s & \text{if } e_1 \neq \{x\}^{r_1} \text{ and } e_2 \neq \{y\}^{r_2} \\
[n]\,[\{x\}^{r}]\,s & \equiv & [\{x\}^{r}]\,[n]\,s & \text{if } n \notin r \\
[\{x\}^{r_1}]\,[\{y\}^{r_2}]\,s & \equiv & [\{y\}^{r_2}]\,[\{x\}^{r_1}]\,s & \text{if } y \notin r_1 \text{ and } x \notin r_2
\end{array}
$$

Table 4.3: Structural congruence (extended laws) for typed COWS

$$
\frac{[\![\bar{\epsilon}]\!] = \bar{v} \qquad \mathrm{fv}(\bar{r}) = \emptyset}{\mathtt{n}!\overline{\{\epsilon\}_r} \xrightarrow{\mathtt{n} \lhd \overline{\{v\}_r}} \mathbf{0}}\ \textit{(r-inv)}
\qquad
\frac{s \xrightarrow{\alpha} s' \qquad x \notin \mathrm{e}(\alpha)}{[\{x\}^{r}]\,s \xrightarrow{\alpha} [\{x\}^{r}]\,s'}\ \textit{(r-del}_{var}\textit{)}
$$

$$
\frac{s \xrightarrow{\mathtt{n}\,\sigma \uplus \{x \mapsto \{v\}_r\}\,\ell\,\bar{v}} s' \qquad r' \cdot \sigma \subseteq r}{[\{x\}^{r'}]\,s \xrightarrow{\mathtt{n}\,\sigma\,\ell\,\bar{v}} s' \cdot \{x \mapsto \{v\}_r\}}\ \textit{(r-del}_{com}\textit{)}
$$

$$
\frac{s \xrightarrow{\alpha} s' \quad e \neq \{x\}^{r} \quad e \notin \mathrm{e}(\alpha) \quad \alpha \neq k, \dagger \quad \mathrm{noKill}(s, e)}{[e]\,s \xrightarrow{\alpha} [e]\,s'}\ \textit{(r-del)}
$$

$$
\frac{s_1 \xrightarrow{\mathtt{n} \rhd \bar{w}} s_1' \quad s_2 \xrightarrow{\mathtt{n} \lhd \overline{\{v\}_r}} s_2' \quad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \quad \mathrm{noConf}(s_1 \mid s_2, \mathtt{n}, \bar{v}, |\sigma|)}{s_1 \mid s_2 \xrightarrow{\mathtt{n}\,\{\bar{w} \mapsto \overline{\{v\}_r}\}\,|\sigma|\,\bar{v}} s_1' \mid s_2'}\ \textit{(r-com)}
$$

Table 4.4: Typed COWS operational semantics (modified rules)

functions of the original semantics of COWS. Label α is now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd \overline{\{v\}_r} \quad | \quad \mathtt{n} \rhd \bar{w} \quad | \quad \mathtt{n}\,\sigma\,\ell\,\bar{v} \quad | \quad k \quad | \quad \dagger$$

The meaning of the new labels is as follows: $\mathtt{n} \lhd \overline{\{v\}_r}$ denotes execution of an invoke activity over the endpoint $\mathtt{n}$ with argument the annotated data $\overline{\{v\}_r}$, while $\mathtt{n}\,\sigma\,\ell\,\bar{v}$ has the same meaning as usual but for the substitution σ that now maps variables to annotated values. We will use $\{\bar{w} \mapsto \overline{\{v\}_r}\}$, with $w = \langle w_1, \ldots, w_n \rangle$ and $\overline{\{v\}_r} = \langle \{v_1\}_{r_1}, \ldots, \{v_n\}_{r_n} \rangle$, to denote the substitution obtained by removing the pairs of the form $v \mapsto \{v\}_r$ from $\{w_1 \mapsto \{v_1\}_{r_1}, \ldots, w_n \mapsto \{v_n\}_{r_n}\}$. The definition of $\mathrm{e}(\alpha)$ remains the same (Section 3.2.3, page 66) except for $\alpha = \mathtt{n}\,\sigma\,\ell\,\bar{v}$ for which we let $\mathrm{e}(\mathtt{n}\,\sigma\,\ell\,\bar{v}) = \mathrm{e}(\sigma)$, where $\mathrm{e}(\{x \mapsto \{v\}_r\}) = \{x, \mathrm{e}(v)\} \cup r$. Finally, a *computation* from a closed service s_0 is a sequence of connected transitions of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \ldots$$

where, for each i, α_i is either $\mathtt{n}\,\emptyset\,\ell\,\bar{v}$ or $\dagger$, and s_i is called *reduct* of s_0.

We comment on salient points. A service invocation can proceed only if the expressions in the argument can be evaluated and their regions do not contain variables (rule

(r-inv)). Communication can take place when two parallel services perform matching receive and invoke activities (rule *(r-com)*). Communication generates a substitution that is recorded in the transition label (for subsequent application). A substitution $\{x \mapsto \{v\}_r\}$ for a variable x is applied to a term (rule *(r-del$_{com}$)*) when the delimitation for x is encountered, i.e. the whole scope s of x is determined, provided that the region annotations of the variable declaration and of the substituent datum v do comply i.e. $r' \cdot \sigma \subseteq r$. This condition also means that as a value is received it gets annotated with a smaller region. The substitution for x is then applied to s and, as usual, x disappears from the term and cannot be reassigned a value.

4.1.2 Main results

Our main results are standard and state that well-typedness is preserved along computations (*subject reduction*) and that well-typed services do respect region annotations (*type safety*). Together, these results imply the *soundness* of our theory, i.e. no violation of data regions will ever occur during the evolution of well-typed services. For the sake of readability, we only outline here the techniques used in the proofs and refer the interested reader to Appendix B.1 for a full account.

For the proof of subject reduction, we need some standard lemmata concerning substitution and weakening. The substitution lemma handles the substitution of partner variables by partner names. Application of a substitution σ to a type environment Γ, written $\Gamma \cdot \sigma$, is defined only when $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\Gamma) = \emptyset$ and, for each $x \mapsto \{v\}_r \in \sigma$, has the effect of replacing every occurrence of x in the regions of Γ with v, i.e.

$$\emptyset \cdot \{x \mapsto \{v\}_r\} = \emptyset \quad \text{and} \quad (\Gamma, \{y : r'\}) \cdot \{x \mapsto \{v\}_r\} = \Gamma \cdot \{x \mapsto \{v\}_r\}, \{y : (r' \cdot \{x \mapsto \{v\}_r\})\}$$

Lemma 4.1.1 (Substitution Lemma) *If* $\Gamma, \{x : r\} \vdash s \succ \Gamma', \{x : r'\} \vdash s'$ *and* $\sigma = \{x \mapsto \{v\}_{r''}\}$, *then* $\Gamma \cdot \sigma \vdash s \cdot \sigma \succ \Gamma' \cdot \sigma \vdash s' \cdot \sigma$.

Proof (sketch). By induction on the length of the inference used to derive the typing judgement, with a case analysis on the last rule used in the derivation. □

Lemma 4.1.2 (Weakening Lemma) *Let* $\Gamma' \vdash s' \succ \Gamma \vdash s$ *and* $x \notin \mathrm{be}(s)$, *then* $\Gamma' + \{x : r\} \vdash s' \succ \Gamma + \{x : r\} \vdash s$.

Proof. By a straightforward induction on the length of the type derivation, with a case analysis on the last used rule, and by exploiting the fact that extending Γ by adding $\{x : r\}$ does not affect the premise of rule *(t-inv)*. □

We also need a few auxiliary results. The first one states that function *halt*(_) preserves well-typedness and can be easily proved by induction on the definition of *halt*(_).

Lemma 4.1.3 *If s is well-typed then halt*(*s*) *is well-typed.*

The other two auxiliary results establish well-typedness preservation by the structural congruence and by the labelled transition relation, respectively. We will use the following preorder $\sqsubseteq$ on type environments: we write $\Gamma \sqsubseteq \Gamma'$ if there exists a type environment Γ'' such that $\Gamma + \Gamma'' = \Gamma'$.

Lemma 4.1.4 *If* $\Gamma' \vdash s'_1 \succ \Gamma \vdash s_1$ *and* $s_1 \equiv s_2$ *then there exists a raw service* s'_2 *such that* $\Gamma' \vdash s'_2 \succ \Gamma \vdash s_2$.

Proof. By a straightforward induction on the derivation of $s_1 \equiv s_2$. □

Theorem 4.1.1 *If* $\Gamma'_1 \vdash s'_1 \succ \Gamma_1 \vdash s_1$ *and* $s_1 \xrightarrow{\alpha} s_2$ *then there exist a raw service* s'_2 *and two type environments* Γ_2 *and* Γ'_2 *such that* $\Gamma_2 \sqsubseteq \Gamma_1$, $\Gamma'_1 \sqsubseteq \Gamma'_2$ *and* $\Gamma'_2 \vdash s'_2 \succ \Gamma_2 \vdash s_2$.

Proof (sketch). By induction on the length of the inference of $s_1 \xrightarrow{\alpha} s_2$, with a case analysis on the last used rule. □

We can now easily prove that well-typedness is preserved along computations.

Corollary 4.1.1 (Subject Reduction) *If* s *is well-typed and* $s \xrightarrow{\alpha} s'$ *with* $\alpha \in \{\dagger, \mathtt{n}\,\emptyset\,\ell\,\bar{v}\}$, *then* s' *is well-typed.*

To characterize the errors that our type system can capture we use predicate $\Uparrow : s \Uparrow$ holds true when s can immediately generate a runtime error. This happens when in an active context there is an invoke activity on a partner not included in the region annotation of some of the expressions argument of the activity. Formally, $\Uparrow$ is defined as the least predicate closed under the following rules

$$\frac{\exists\, r' \in \bar{r}\,.\;\; p \notin r'}{p \bullet o!\overline{\{\epsilon\}_r} \Uparrow} \qquad \frac{s \Uparrow}{\mathbb{A}[\![s]\!] \Uparrow} \qquad \frac{s \equiv s' \quad s \Uparrow}{s' \Uparrow}$$

where $\mathbb{A}$ is an *active context*, namely a service $\mathbb{A}$ with a 'hole' $[\![\cdot]\!]$, i.e. a term generated by the following grammar:

$$\mathbb{A} \;::=\; [\![\cdot]\!] \;\mid\; \mathbb{A} + g \;\mid\; g + \mathbb{A} \;\mid\; \mathbb{A} \mid s \;\mid\; s \mid \mathbb{A} \;\mid\; \{\!|\mathbb{A}|\!\} \;\mid\; [e]\,\mathbb{A} \;\mid\; * \mathbb{A}$$

such that, once the hole is filled with a service s, the resulting term $\mathbb{A}[\![s]\!]$ is a COWS service that is capable of immediately performing an activity of s.

We remark that the runtime errors that our type discipline can capture are related to the policies for the exchange of data. We skip such runtime errors as 'unproper use of variables' (e.g. in $x \bullet o!\bar{v}$ the variable x is not replaced by a partner name) that can be easily dealt with standard type systems. We can now prove that well-typed services do respect region annotations.

Theorem 4.1.2 (Type Safety) *If s is a well-typed service then $s \Uparrow$ does not hold.*

Proof (sketch). By induction on the derivation of $s \Uparrow$, with a case analysis on the last used rule, we prove that if $s \Uparrow$ then s is not well-typed, from which the thesis follows. □

We can finally conclude by stating that the type system and the operational semantics are *sound*.

Corollary 4.1.2 (Type Soundness) *Let s be a well-typed service. Then $s' \Uparrow$ does not hold for every reduct s' of s.*

Proof. Corollary 4.1.1 can be repeatedly applied to prove that s' is well-typed, then Theorem 4.1.2 permits to conclude. □

4.1.3 Application scenarios

In this section, we illustrate some applications of our framework. The first example is a simplified but realistic electronic marketplace scenario inspired by W3C [178]. To show usefulness of our approach, we focus on the central part of the protocol where sensitive data are exchanged, i.e. we omit the initial bartering and the concluding interactions, and expand the part relative to the payment process. The second example is inspired to an automotive scenario focussing in security issues studied in the SENSORIA project [185], while the last two examples are applications to the case studies presented in Sections 2.4.1 and 2.4.2.

An electronic marketplace. Suppose a service *buyer* invokes a service *seller* to purchase some goods. Once *seller* has received an order request, it sends back the partner name of the service *credit_agency* to be used for the payment. *buyer* can then check the information on *credit_agency* and, possibly, confirm the payment by sending its credit card data to *seller*. In this case, *seller* forwards the received data to *credit_agency* and passes the order to the service *shipper*. In the end, the whole system is

$$EMP \triangleq buyer \mid credit\_agency \mid [p_{sh}]\ (seller \mid shipper)$$

When fixing the policies for data exchange, services can (safely) assume that, at the outset, partner names p_s, p_{ca} and p_b are publicly available for invoking *seller*, *credit_agency* and *buyer*, respectively. Instead, the partner name p_{sh} for invoking *shipper* is private and only shared with *seller*. Of course, due to the syntactical restrictions, the 'locality' condition for partner names is preserved by the semantics. Thus, the initials assumptions remain true forever.

The *buyer* service is defined as

$$\begin{aligned} buyer \triangleq\ & [id]\,(p_s \bullet o_{ord}!\langle \{id\}_{\{p_s,p_b\}}, p_b, order\rangle \\ & \mid [x_{ca}]\, p_b \bullet o_{ca\_info}?\langle id, x_{ca}\rangle. \\ & \qquad [p,o]\,(\, p \bullet o!\langle\rangle \mid p \bullet o?\langle\rangle.p_s \bullet o_{pay}!\langle \{id\}_{\{p_s,p_b\}}, \{cc\_data\}_{\{p_s,x_{ca}\}}\rangle \\ & \qquad\qquad + \ p \bullet o?\langle\rangle.p_s \bullet o_{canc}!\langle \{id\}_{\{p_s,p_b\}}\rangle\,)\,) \end{aligned}$$

The endpoint $p_s \bullet o_{ord}$ is used for invoking the seller service and transmitting the order together with the *buyer*'s partner name p_b. The (restricted) name *id* represents the order identifier and is used for correlating all those service interactions that logically form a same session relative to the processing of *order*. For example, the specification of *buyer* could be slightly modified to allow the service to simultaneously make multiple orders: of course, although all such parallel threads must use the same partner p_s to interact with *seller*, they can exploit different order identifiers as a means to correlate messages belonging to different interaction sessions. The type attached to *id* only allows *buyer* and *seller* to exchange and use it, since they are the only services that can receive along p_s and p_b. Instead, p_b comes without an attached policy, since it is publicly known (it is transmitted to indicate the service making the invocation for the call-back operation). For simplicity, also *order* has no attached policy; thus, it could be later on communicated to any other service. Variable x_{ca} is used to store the partner name of the credit agency service to be used to possibly finalize the purchase and also to implement the policy for *buyer*'s credit card data. After the information on the credit agency service are verified, *buyer* sends a message to *seller* either to confirm or to cancel the order. This is simply modelled as an internal non-deterministic choice, by exploiting the private endpoint $p \bullet o$.

The *seller* service is defined as

$$\begin{aligned} seller \triangleq\ & *\,[x_b, x_{id}, x_{ord}, k]\, p_s \bullet o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle. \\ & \qquad (\, x_b \bullet o_{ca\_info}!\langle \{x_{id}\}_{\{x_b\}}, p_{ca}\rangle \\ & \qquad\quad \mid [x_{cc}]\, p_s \bullet o_{pay}?\langle x_{id}, x_{cc}\rangle.(\, p_{ca} \bullet o_{cr\_req}!\langle x_{ord}, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\ & \qquad\qquad\qquad\qquad\qquad \mid p_{sh} \bullet o_{sh\_req}!\langle x_{ord}\rangle\,) \\ & \qquad\quad \mid p_s \bullet o_{canc}?\langle x_{id}\rangle.\mathbf{kill}(k)\,) \end{aligned}$$

Once *seller* receives an order along $p_s \bullet o_{ord}$, it creates one specific instance that sends back to *buyer* (via x_b) the partner name p_{ca} of the credit agency service where the payment will be made. Whenever the seller instance receives the credit card data correlated to x_{id}, it forwards them to *credit_agency* and passes the order to the (internal) shipper service. Instead, if *buyer* demands cancellation of the order, the corresponding instance of *seller* is immediately terminated. Label k is used to delimit the effect of the kill activity only to the relevant instance.

The remaining two services are defined as

$$\begin{aligned} credit\_agency &\triangleq *\,[x,y]\, p_{ca} \bullet o_{cr\_req}?\langle x, y\rangle.\text{“}\texttt{execute\_the\_payment}\text{”} \\ shipper &\triangleq *\,[z]\, p_{sh} \bullet o_{sh\_req}?\langle z\rangle.\text{“}\texttt{process\_the\_order}\text{”} \end{aligned}$$

Let now consider the type inference phase. Service *seller* gets annotated as follows:

$$\begin{aligned} \mathit{seller}' \triangleq\ & *\,[\{x_b\}^{\{p_s\}}]\,[\{x_{id}\}^{\{p_s,x_b\}}, \{x_{ord}\}^{\top}, k]\; p_s \bullet o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle. \\ & (\; x_b \bullet o_{ca\_info}!\langle \{x_{id}\}_{\{x_b\}}, p_{ca}\rangle \\ & \quad \mid\ [\{x_{cc}\}^{\{p_s,p_{ca}\}}]\; p_s \bullet o_{pay}?\langle x_{id}, x_{cc}\rangle.(\; p_{ca} \bullet o_{cr\_req}!\langle x_{ord}, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mid\ p_{sh} \bullet o_{sh\_req}!\langle x_{ord}\rangle\;) \\ & \quad \mid\ p_s \bullet o_{canc}?\langle x_{id}\rangle.\mathbf{kill}(k)\;) \end{aligned}$$

The type inference has to check consistency of region annotations of the arguments occurring within invoke activities and that of deriving the annotations for variable declarations. As regards consistency, there are only two explicitly typed expressions used as arguments of invoke activities, i.e. x_{id} and x_{cc}, and their types $\{x_b\}$ and $\{p_{ca}\}$ satisfy the consistency constraint (see rule *(t-inv)*). The remaining expressions occurring as arguments of invoke activities, i.e. the only x_{ord}, have implicitly assigned type $\top$ (indeed, recall that we assumed that an untagged ϵ stands for $\{\epsilon\}_{\top}$) and are thus trivially consistent. As regards type derivation, when a variable is put in the environment (rule *(t-del$_{var}$)*), it is assigned type $\emptyset$. Later on, when a variable is used as an argument of an invoke or receive, its type can possibly be enriched (rules *(t-inv)* and *(t-rec)*). Thus, at the end of the inference, declaration of variable x_b, that is only used in $p_s \bullet o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle$, will have assigned region $\{p_s\}$ (application of rule *(t-rec)*). Instead, declaration of x_{ord} has assigned type $\top$ (rule *(t-inv)* is used) while that of x_{cc} has assigned type $\{p_s, p_{ca}\}$ and, similarly, declaration of x_{id} gets annotated with $\{p_s, x_b\}$ (in both cases rules *(t-inv)* and *(t-rec)* are used). Notably, in seller', delimitation $[\{x_b\}^{p_s}]$ does not commute any longer with delimitations $[\{x_{id}\}^{\{p_s,x_b\}}, \{x_{ord}\}^{\top}, k]$ (otherwise the service would become opened).

The variable declarations of the other services are annotated in a trivial way: x_{ca} with $\{p_b\}$, x and y with $\{p_{ca}\}$, and z with $\{p_{sh}\}$ (we assume that *credit_agency* and *shipper* do not re-transmit the received data). Thus, if we call buyer', $\mathit{credit\_agency}'$ and $\mathit{shipper}'$ the other typed services, then the system resulting from the type inference is

$$\mathit{buyer}' \mid \mathit{credit\_agency}' \mid [p_{sh}]\;(\mathit{seller}' \mid \mathit{shipper}')$$

After some computational steps, the system can become

$$\begin{aligned} & [id]\,(p_s \bullet o_{pay}!\langle \{id\}_{\{p_s,p_b\}}, \{cc\_data\}_{\{p_s,p_{ca}\}}\rangle \;\mid\; [p_{sh}]\,(\,\mathit{seller}' \;\mid \\ & \qquad [k, \{x_{cc}\}^{\{p_s,p_{ca}\}}]\,(\; p_s \bullet o_{pay}?\langle id, x_{cc}\rangle.(\; p_{ca} \bullet o_{cr\_req}!\langle order, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mid\ p_{sh} \bullet o_{sh\_req}!\langle order\rangle\;) \\ & \qquad\qquad\qquad \mid\ p_s \bullet o_{canc}?\langle id\rangle.\mathbf{kill}(k)\;) \\ & \qquad \mid\ *\,[\{x\}^{\{p_{ca}\}}, \{y\}^{\{p_{ca}\}}]\; p_{ca} \bullet o_{cr\_req}?\langle x, y\rangle.\text{“}\texttt{execute\_the\_payment}\text{”} \\ & \qquad \mid\ *\,[\{z\}^{\{p_{sh}\}}]\; p_{sh} \bullet o_{sh\_req}?\langle z\rangle.\text{“}\texttt{process\_the\_order}\text{”})\,) \end{aligned}$$

Thus, after buyer' sends the credit card data, we get

$$\begin{aligned} & [id, p_{sh}]\,(\mathit{seller}' \\ & \qquad \mid\ [k]\,(\; p_{ca} \bullet o_{cr\_req}!\langle order, \{cc\_data\}_{\{p_{ca}\}}\rangle \mid p_{sh} \bullet o_{sh\_req}!\langle order\rangle \\ & \qquad\qquad \mid\ p_s \bullet o_{canc}?\langle id\rangle.\mathbf{kill}(k)\;) \\ & \qquad \mid\ *\,[\{x\}^{\{p_{ca}\}}, \{y\}^{\{p_{ca}\}}]\; p_{ca} \bullet o_{cr\_req}?\langle x, y\rangle.\text{“}\texttt{execute\_the\_payment}\text{”} \\ & \qquad \mid\ *\,[\{z\}^{\{p_{sh}\}}]\; p_{sh} \bullet o_{sh\_req}?\langle z\rangle.\text{“}\texttt{process\_the\_order}\text{”}\,) \end{aligned}$$

At this point, *seller′* can safely communicate credit card data of *buyer′* to *credit_agency′* and, then, forward the order to *shipper′*.

Suppose now that *seller′* also contains such a malicious invoke as $p_{sh} \cdot o!\langle \ldots, \{x_{cc}\}_r, \ldots \rangle$. In order to successfully pass the type inference phase, it should be that $p_{sh} \in r$ (otherwise rule *(t-inv)* could not be applied). Therefore, in the resulting typed service we would have the variable declaration $[\{x_{cc}\}^{r'}]$, with $r \subseteq r'$. Now, communication with *buyer′* would be blocked by the runtime checks because the datum is tagged as $\{cc\_data\}_{\{p_s, p_{ca}\}}$, and $p_{sh} \in r \subseteq r'$ implies that $r' \not\subseteq \{p_s, p_{ca}\}$.

Automotive security. A car manufacturer offers an integrated infotainment system capable of forwarding the GPS coordinates of a car to relevant recipient services. We consider the case of a system designed to allow drivers to assist each other on deserted roads until professional service can arrive at the scene. The system is based on an automobile association monitoring the location of all trustworthy members' cars and contacting the drivers nearest to an accident to ask them to provide first aid. The driver in trouble is assured that information about his location cannot become available to unauthorized users. We suppose that customers of the automobile association service may not be trustworthy members. Thus, a driver can play both helper and customer roles, or only the latter one.

We consider the following (simplified) scenario

$$\mathrm{C} \mid \mathrm{T}_1 \mid \mathrm{T}_2 \mid \mathrm{T}_3 \mid \mathrm{A}$$

where C is the service installed in the customer's car, T_1, T_2, and T_3 are services installed in the trustworthy members' cars, and A is the service provided by the automobile association. For simplicity, we do not consider the customer services installed in the member's cars.

The *customer* service is defined as

$$\mathrm{C} \triangleq [p_{sys}]\,(\, p_{sys} \cdot o_{accident}!\langle gps \rangle \mid [y_{\mathrm{T}}]\,[y_{gps}]\, p_{sys} \cdot o_{accident}?\langle y_{gps} \rangle .\, \mathrm{C}_{\mathrm{findHelp}}\,)$$

$$\begin{aligned}\mathrm{C}_{\mathrm{findHelp}} \triangleq (\; & p_{\mathrm{A}} \cdot o_{carAccident}!\langle \{p_{\mathrm{C}}\}_{\{p_{\mathrm{A}}\}}, \{y_{gps}\}_{\{p_{\mathrm{A}}\}} \rangle \\ & \mid [y_{id}]\,(\, p_{\mathrm{C}} \cdot o_{helpNotFound}?\langle\rangle \\ & \qquad + \\ & \qquad p_{\mathrm{C}} \cdot o_{helpFound}?\langle y_{id}, y_{\mathrm{T}} \rangle .\, p_{\mathrm{A}} \cdot o_{position}!\langle y_{id}, \{y_{gps}\}_{\{p_{\mathrm{A}}, y_{\mathrm{T}}\}} \rangle\,)\,)\end{aligned}$$

When the diagnostic system of the customer's car detects that an accident has happened, it activates the recovery service by sending the current GPS coordinates of the car along the private endpoint $p_{sys} \cdot o_{accident}$. Then, it invokes the automobile association service A, by using the endpoint $p_{\mathrm{A}} \cdot o_{carAccident}$, and transmits its partner name (i.e. p_{C}, for the reply) and its coordinates. The type attached to those data only allows A to exchange and use them, since A is the only service that can receive along p_{A}. Afterwards, C waits a reply from A. If no trustworthy helper has been found, then the driver has to wait until professional service arrives at the scene. In case A has found a trustworthy helper T_i, C

transmits its coordinates again, this time with region enlarged with the partner name of T_i. Deliberately we do not model C as a persistent service; instead, we assume that it will be reinstalled at the end of the repairing activity.

A *trustworthy member* service is defined as

$$\begin{aligned}\mathrm{T}_i \triangleq\ & [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid *\ \mathtt{m}?\langle\rangle.[x_{id}]\,p_{\mathrm{T}i}\bullet o_{check}?\langle x_{id}\rangle.\\ & [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid \mathtt{n}?\langle\rangle.(\,p_{\mathrm{A}}\bullet o_{refuse}!\langle x_{id}\rangle \mid \mathtt{m}!\langle\rangle\,)\\ & \qquad + \\ & \qquad \mathtt{n}?\langle\rangle.(\,p_{\mathrm{A}}\bullet o_{ok}!\langle x_{id}\rangle\\ & \qquad\quad \mid [x_{gps}]\,p_{\mathrm{T}i}\bullet o_{fwdPos}?\langle x_{gps}\rangle.\\ & \qquad\quad\ \text{“}\mathtt{go\ to}\ x_{gps}\ \mathtt{and\ provide\ first\ aid}\text{”}.\mathtt{m}!\langle\rangle\,)\,)\end{aligned}$$

This is a persistent service that, however, can serve requests only sequentially (i.e. one at time). When the automobile association contacts one of its members, firstly asks him (by invoking the operation o_{check}) if he is willing to provide first aid to a near driver. Service T_i can accept or refuse by replying on the operations o_{ok} or o_{refuse}, respectively. For simplicity, we model the above decision by means of an internal non-deterministic choice exploiting the private endpoint $\mathtt{n}$. In case of negative reply, T_i is immediately reactivated (by means of $\mathtt{m}!\langle\rangle$). Otherwise, i.e. in case of positive reply, the service waits for the position of the driver in trouble from A, goes to the accident scene, provides first aid and, finally, reactivates itself.

The *automobile association* service is defined as

$$\begin{aligned}\mathrm{A} \triangleq\ & [o_{find}, o_{found}, o_{notFound}]\\ & (\,*\,[z_{\mathrm{C}}, z_{gps}]\,p_{\mathrm{A}}\bullet o_{carAccident}?\langle z_{\mathrm{C}}, z_{gps}\rangle.\,[id]\,\mathrm{A}_{\mathrm{inst}}\\ & \ \mid *\,[z_{id}, z_{pos}]\,p_{\mathrm{A}}\bullet o_{find}?\langle z_{id}, z_{pos}\rangle.\\ & \qquad\qquad \text{“}\mathtt{find\ in\ DB\ and\ reply\ on}\ o_{found}\ \mathtt{or}\ o_{notFound}\text{”}\,)\end{aligned}$$

$$\begin{aligned}\mathrm{A}_{\mathrm{inst}} \triangleq\ & [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.(\,p_{\mathrm{A}}\bullet o_{find}!\langle id, z_{gps}\rangle\\ & \qquad \mid [z'_{T}]\,(\,p_{\mathrm{A}}\bullet o_{notFound}?\langle id\rangle.\,z_{\mathrm{C}}\bullet o_{helpNotFound}!\langle\rangle\\ & \qquad\quad + \\ & \qquad\quad p_{\mathrm{A}}\bullet o_{found}?\langle id, z'_{\mathrm{T}}\rangle.(\,z'_{\mathrm{T}}\bullet o_{check}!\langle id\rangle\\ & \qquad\qquad \mid p_{\mathrm{A}}\bullet o_{refuse}?\langle id\rangle.\,\mathtt{n}!\langle\rangle\\ & \qquad\qquad\quad + \\ & \qquad\qquad\quad p_{\mathrm{A}}\bullet o_{ok}?\langle id\rangle.\,[z_{\mathrm{T}}, o]\,(\,p_{\mathrm{A}}\bullet o!\langle z'_{\mathrm{T}}\rangle\\ & \qquad\qquad\qquad \mid p_{\mathrm{A}}\bullet o?\langle z_{\mathrm{T}}\rangle.\\ & \qquad\qquad\qquad\ (\,z_{\mathrm{C}}\bullet o_{helpFound}!\langle id, \{z_{\mathrm{T}}\}_{\{z_{\mathrm{C}}\}}\rangle\\ & \qquad\qquad\qquad\quad \mid [z'_{gps}]\,p_{\mathrm{A}}\bullet o_{position}?\langle id, z'_{gps}\rangle.\\ & \qquad\qquad\qquad\qquad z_{\mathrm{T}}\bullet o_{fwdPos}!\langle\{z'_{gps}\}_{\{z_{\mathrm{T}}\}}\rangle\,)\,)\,)\,)\,)\,)\end{aligned}$$

This service is composed of two persistent subservices, both capable of receiving along p_{A}. Service A is publicly invocable and can interact with customers and members services, other than with the ‘internal’ service $\mathrm{A}_{\mathrm{inst}}$. This latter service, instead, can only be invoked by A (indeed, all the operations used by the service, i.e. $o_{find}, o_{found}, o_{notFound}$,

are restricted and this prevents them to be invoked from the outside) and has the task of looking up in a database the member nearest to given a location and replying accordingly.

Differently from C and T_i, A and $\mathrm{A}_{\mathrm{inst}}$ can serve requests concurrently and, thus, exploit correlation mechanisms to route each received message to its right instance. Indeed, when a customer request is received along $p_{\mathrm{A}} \bullet o_{carAccident}$, a new specific service instance is created that is uniquely identified by a fresh correlation identifier *id*; this identifier is generated by the instance itself and is communicated to every other involved service. The instance then tries to find a helper member that is near to the accident scene and is willing to provide first aid. This is done by repeatedly looking up the database ($p_{\mathrm{A}} \bullet o_{find}!\langle id, z_{gps}\rangle$) and checking willingness of the returned trustworthy member ($z'_{\mathrm{T}} \bullet o_{check}!\langle id\rangle$), until either one available member is found (i.e. the member replies on operation o_{ok}) or the looking up in the database fails (i.e. a reply on $o_{notFound}$ is returned). Technically, the cycle is implemented by means of the replication operator and the private endpoint n. If no available helper is found, the instance notifies this to the customer ($z_{\mathrm{C}} \bullet o_{helpNotFound}!\langle\rangle$) and terminates; otherwise, it sends the partner name of the found helper to the customer ($z_{\mathrm{C}} \bullet o_{helpFound}!\langle id, \{z_{\mathrm{T}}\}_{\{z_{\mathrm{C}}\}}\rangle$), waits for the customer location (whose region now also include the partner name of the helper) and forwards it to the helper ($z_{\mathrm{T}} \bullet o_{fwdPos}!\langle\{z'_{gps}\}_{\{z_{\mathrm{T}}\}}\rangle$).

Notably, the only relevant types are that attached to the partner name of the found helper, which allows only C to receive this partner, and that attached to the customer location, which allows only the found helper to receive such location.

Let now consider the type inference phase. Service C gets annotated as follows:

$$\begin{aligned}\mathrm{C}' \triangleq\ & [p_{sys}]\,(\,p_{sys} \bullet o_{accident}!\langle gps\rangle \\ & \quad \mid [\{y_{\mathrm{T}}\}^{\{p_{\mathrm{C}}\}}]\,[\{y_{gps}\}^{\{p_{sys},p_{\mathrm{A}},y_{\mathrm{T}}\}}]\,p_{sys} \bullet o_{accident}?\langle y_{gps}\rangle.\,\mathrm{C}'_{\mathrm{findHelp}}\,)\end{aligned}$$

$$\begin{aligned}\mathrm{C}'_{\mathrm{findHelp}} \triangleq\ & (\,p_{\mathrm{A}} \bullet o_{carAccident}!\langle\{p_{\mathrm{C}}\}_{\{p_{\mathrm{A}}\}}, \{y_{gps}\}_{\{p_{\mathrm{A}}\}}\rangle \\ & \quad \mid [\{y_{id}\}^{\top}]\,(\,p_{\mathrm{C}} \bullet o_{helpNotFound}?\langle\rangle \\ & \qquad + \\ & \qquad p_{\mathrm{C}} \bullet o_{helpFound}?\langle y_{id}, y_{\mathrm{T}}\rangle.\ p_{\mathrm{A}} \bullet o_{position}!\langle y_{id}, \{y_{gps}\}_{\{p_{\mathrm{A}},y_{\mathrm{T}}\}}\rangle\,)\,)\end{aligned}$$

As regards consistency of region annotations, the only explicitly typed expressions used as arguments of invoke activities are p_{C} and y_{gps} (this latter is used twice); in any case, their types ($\{p_{\mathrm{A}}\}$, $\{p_{\mathrm{A}}\}$ and $\{p_{\mathrm{A}}, y_{\mathrm{T}}\}$, resp.) satisfy the consistency constraint (see rule *(t-inv)*). The remaining expressions occurring as arguments of invoke activities, i.e. gps and y_{id}, have implicitly assigned type $\top$ and, thus, the consistency constraint is trivially satisfied. As regards derivation of types for variable declarations, at the end of the inference, declaration of variable y_{T}, that is only used in $p_{\mathrm{C}} \bullet o_{helpFound}?\langle y_{id}, y_{\mathrm{T}}\rangle$ and in the region of y_{gps}, will have assigned region $\{p_{\mathrm{C}}\}$ (application of rule *(t-rec)*). Instead, y_{gps} will have assigned type $\{p_{sys}, p_{\mathrm{A}}, y_{\mathrm{T}}\}$ (rules *(t-inv)* and *(t-rec)*), while y_{id} will have assigned type $\top$ (rule *(t-inv)*). Notably, in C′, delimitation $[\{y_{\mathrm{T}}\}^{\{p_{\mathrm{C}}\}}]$ does not commute any longer with delimitation $[\{y_{gps}\}^{\{p_{sys},p_{\mathrm{A}},y_{\mathrm{T}}\}}]$ (otherwise the service would become opened).

Services T_i get annotated as follows:

$$
\begin{aligned}
\mathrm{T}'_i \triangleq\ & [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid *\ \mathtt{m}?\langle\rangle.[\{x_{id}\}^{\top}]\ p_{\mathrm{T}i} \bullet o_{check}?\langle x_{id}\rangle. \\
& \qquad [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid \mathtt{n}?\langle\rangle.(\,p_{\mathrm{A}} \bullet o_{refuse}!\langle x_{id}\rangle \mid \mathtt{m}!\langle\rangle\,) \\
& \qquad\qquad + \\
& \qquad\qquad \mathtt{n}?\langle\rangle.(\,p_{\mathrm{A}} \bullet o_{ok}!\langle x_{id}\rangle \\
& \qquad\qquad\qquad \mid [\{x_{gps}\}^{\{p_{\mathrm{T}i}\}}]\ p_{\mathrm{T}i} \bullet o_{fwdPos}?\langle x_{gps}\rangle. \\
& \qquad\qquad\qquad\quad \text{“}\mathtt{go\ to}\ x_{gps}\ \mathtt{and\ provide\ first\ aid}\text{”}.\mathtt{m}!\langle\rangle\,)\,)
\end{aligned}
$$

In this case, derivation of types for variables is trivial: x_{id} has type $\top$ and x_{gps} has type $\{p_{\mathrm{T}i}\}$ (we assume that T_i does not re-transmit the GPS data store in x_{gps}).

Service A and $\mathrm{A}_{\mathrm{inst}}$ get annotated as follows:

$$
\begin{aligned}
\mathrm{A}' \triangleq\ & [o_{find}, o_{found}, o_{notFound}] \\
& (*\,[\{z_{\mathrm{C}}\}^{\{p_{\mathrm{A}}\}}, \{z_{gps}\}^{\{p_{\mathrm{A}}\}}]\ p_{\mathrm{A}} \bullet o_{carAccident}?\langle z_{\mathrm{C}}, z_{gps}\rangle.\ [id]\ \mathrm{A}'_{\mathrm{inst}} \\
& \ \mid *\,[\{z_{id}\}^{\{p_{\mathrm{A}}\}}, \{z_{pos}\}^{\{p_{\mathrm{A}}\}}]\ p_{\mathrm{A}} \bullet o_{find}?\langle z_{id}, z_{pos}\rangle. \\
& \qquad\qquad\qquad \text{“}\mathtt{find\ in\ DB\ and\ reply\ on}\ o_{found}\ \mathtt{or}\ o_{notFound}\text{”})
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{A}'_{\mathrm{inst}} \triangleq\ & [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.\,(\,p_{\mathrm{A}} \bullet o_{find}!\langle id, z_{gps}\rangle \\
& \qquad\qquad \mid [\{z'_T\}^{\{p_{\mathrm{A}}\}}]\,(\,p_{\mathrm{A}} \bullet o_{notFound}?\langle id\rangle.\, z_{\mathrm{C}} \bullet o_{helpNotFound}!\langle\rangle \\
& \qquad\qquad\qquad + \\
& \qquad\qquad\qquad p_{\mathrm{A}} \bullet o_{found}?\langle id, z'_{\mathrm{T}}\rangle.\,(z'_{\mathrm{T}} \bullet o_{check}!\langle id\rangle \\
& \qquad\qquad\qquad\quad \mid p_{\mathrm{A}} \bullet o_{refuse}?\langle id\rangle.\,\mathtt{n}!\langle\rangle \\
& \qquad\qquad\qquad\qquad + \\
& \qquad\qquad\qquad\qquad p_{\mathrm{A}} \bullet o_{ok}?\langle id\rangle.\,[\{z_{\mathrm{T}}\}^{\{p_{\mathrm{A}}, z_{\mathrm{C}}\}}, o]\,(\,p_{\mathrm{A}} \bullet o!\langle z'_{\mathrm{T}}\rangle \\
& \qquad\qquad\qquad\qquad\quad \mid p_{\mathrm{A}} \bullet o?\langle z_{\mathrm{T}}\rangle.(\,z_{\mathrm{C}} \bullet o_{helpFound}!\langle id, \{z_{\mathrm{T}}\}_{\{z_{\mathrm{C}}\}}\rangle \\
& \qquad\qquad\qquad\qquad\qquad \mid [\{z'_{gps}\}^{\{p_{\mathrm{A}}, z_{\mathrm{T}}\}}]\ p_{\mathrm{A}} \bullet o_{position}?\langle id, z'_{gps}\rangle. \\
& \qquad\qquad\qquad\qquad\qquad\quad z_{\mathrm{T}} \bullet o_{fwdPos}!\langle\{z'_{gps}\}_{\{z_{\mathrm{T}}\}}\rangle\,)\,)\,)\,)\,)\,)
\end{aligned}
$$

Variables z_{C}, z_{gps}, z_{id}, z_{pos}, z'_{T} are trivially annotated with type $\{p_{\mathrm{A}}\}$, because A does not re-transmit the corresponding data, while variables z_{T} and z'_{gps} are annotated with $\{p_{\mathrm{A}}, z_{\mathrm{C}}\}$ and $\{p_{\mathrm{A}}, z_{\mathrm{T}}\}$, respectively.

Therefore, the system resulting from the type inference is

$$\mathrm{C}' \mid \mathrm{T}'_1 \mid \mathrm{T}'_2 \mid \mathrm{T}'_3 \mid \mathrm{A}'$$

Figure 4.1 shows a possible evolution of the scenario, where T'_3 is too far from the accident scene, T'_2 is not available, and T'_1 accepts to provide first aid to C'.

For example, after the computation steps 1–9 described in Figure 4.1, the system be-

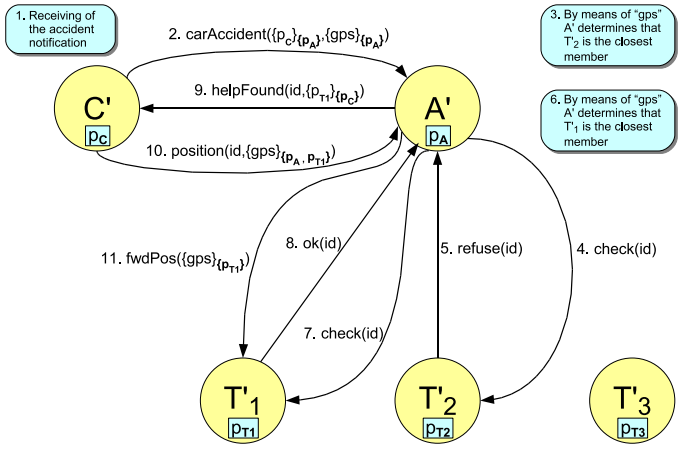

Figure 4.1: Automotive security scenario

comes

$$
\begin{array}{l}
p_{\mathrm{A}} \bullet o_{position}!\langle id, \{gps\}_{\{p_{\mathrm{A}}, p_{\mathrm{T1}}\}}\rangle \\
\mid \\
[o_{find}, o_{found}, o_{notFound}] \\
(*\,[\{z_{\mathrm{C}}\}^{\{p_{\mathrm{A}}\}}, \{z_{gps}\}^{\{p_{\mathrm{A}}\}}]\, p_{\mathrm{A}} \bullet o_{carAccident}?\langle z_{\mathrm{C}}, z_{gps}\rangle.\ \cdots \\
\quad \mid *\,[\{z_{id}\}^{\{p_{\mathrm{A}}\}}, \{z_{pos}\}^{\{p_{\mathrm{A}}\}}]\, p_{\mathrm{A}} \bullet o_{find}?\langle z_{id}, z_{pos}\rangle.\ \cdots \\
\quad \mid [\{z'_{gps}\}^{\{p_{\mathrm{A}}, p_{\mathrm{T1}}\}}]\, p_{\mathrm{A}} \bullet o_{position}?\langle id, z'_{gps}\rangle.\, p_{\mathrm{T1}} \bullet o_{fwdPos}!\langle \{z'_{gps}\}_{\{p_{\mathrm{T1}}\}}\rangle\,) \\
\mid \\
[\mathtt{m}]\,(*\,\mathtt{m}?\langle\rangle.\ \cdots \\
\qquad \mid [\{x_{gps}\}^{\{p_{\mathrm{T1}}\}}]\, p_{\mathrm{T1}} \bullet o_{fwdPos}?\langle x_{gps}\rangle.\, \text{“}\mathtt{go\ to}\ x_{gps}\ \mathtt{and\ provide\ first\ aid}\text{”}.\mathtt{m}!\langle\rangle\,) \\
\mid \\
\mathrm{T}'_2 \mid \mathrm{T}'_3
\end{array}
$$

At this point, the customer can safely communicate its location to the automobile association (namely, rule *(r-del$_{sub}$)* of the operational semantics can be applied because the region annotations of gps and z'_{gps} are both $\{p_{\mathrm{A}}, p_{\mathrm{T1}}\}$ and, hence, do comply), and the system becomes

$$
\begin{array}{l}
[o_{find}, o_{found}, o_{notFound}] \\
(*\,[\{z_{\mathrm{C}}\}^{\{p_{\mathrm{A}}\}}, \{z_{gps}\}^{\{p_{\mathrm{A}}\}}]\, p_{\mathrm{A}} \bullet o_{carAccident}?\langle z_{\mathrm{C}}, z_{gps}\rangle.\ \cdots \\
\quad \mid *\,[\{z_{id}\}^{\{p_{\mathrm{A}}\}}, \{z_{pos}\}^{\{p_{\mathrm{A}}\}}]\, p_{\mathrm{A}} \bullet o_{find}?\langle z_{id}, z_{pos}\rangle.\ \cdots \\
\quad \mid p_{\mathrm{T1}} \bullet o_{fwdPos}!\langle \{gps\}_{\{p_{\mathrm{T1}}\}}\rangle\,) \\
\mid \\
[\mathtt{m}]\,(*\,\mathtt{m}?\langle\rangle.\ \cdots \\
\qquad \mid [\{x_{gps}\}^{\{p_{\mathrm{T1}}\}}]\, p_{\mathrm{T1}} \bullet o_{fwdPos}?\langle x_{gps}\rangle.\, \text{“}\mathtt{go\ to}\ x_{gps}\ \mathtt{and\ provide\ first\ aid}\text{”}.\mathtt{m}!\langle\rangle\,) \\
\mid \\
\mathrm{T}'_2 \mid \mathrm{T}'_3
\end{array}
$$

Now, the automobile association can safely forward the customer location, because also the region annotations of gps and x_{gps} do comply.

Applying the approach to the automotive case study. We illustrate here some relevant properties for the automotive case study informally presented in Section 2.4.1 and specified in COWS in Section 3.4.1.

Firstly, a driver in trouble must be assured that information about his credit card and his location cannot become available to unauthorized users. Thus, for example, the credit card identifier *ccNum*, communicated by activity *CardCharge* to service *Bank*, gets annotated with the policy $\{p_{bank}\}$, that allows *Bank* to receive the datum but prevents it from transmitting the datum to other services. Other non-critical data, e.g. *amount*, can be transmitted without an attached policy. The typed version of *CardCharge* is defined as follows

$$
\begin{array}{l}
p_{bank} \bullet o_{charge}!\langle p_{car}, \{ccNum\}_{\{p_{bank}\}}, amount, p_{car}\rangle \\
\mid \{\!| p_{car} \bullet o_{chargeFail}?\langle p_{car}\rangle.\ \mathbf{kill}(k) \\
\quad + p_{car} \bullet o_{chargeOK}?\langle p_{car}\rangle. \\
\quad\ (\ p_{end} \bullet o_{end}!\langle\rangle \\
\quad\quad \mid p_{car} \bullet o_{undo}?\langle cc\rangle.\ p_{car} \bullet o_{undo}?\langle cc\rangle.\ p_{bank} \bullet o_{revoke}!\langle p_{car}\rangle\ )\ |\!\}
\end{array}
$$

Once the type inference phase ends, *BankInterface* gets annotated as follows

$$
\begin{array}{l}
[\{x_{cust}\}^{\{p_{bank}\}}, \{x_{cc}\}^{\{p_{bank}\}}, \{x_{amount}\}^{\{p_{bank}\}}, \{x_{id}\}^{\{p_{bank}, x_{cust}\}}] \\
p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle. \\
\quad (\ p_{bank} \bullet o_{check}!\langle x_{id}, x_{cc}, x_{amount}\rangle \\
\quad\ \mid p_{bank} \bullet o_{checkFail}?\langle x_{id}\rangle.\ x_{cust} \bullet o_{chargeFail}!\langle x_{id}\rangle \\
\quad\quad + \ p_{bank} \bullet o_{checkOK}?\langle x_{id}\rangle. \\
\quad\quad\quad [k']\,(\ x_{cust} \bullet o_{chargeOK}!\langle x_{id}\rangle \mid p_{bank} \bullet o_{revoke}?\langle x_{id}\rangle.\ \mathbf{kill}(k')\ )\ )
\end{array}
$$

Indeed, the annotations inferred for variables x_{cust}, x_{cc}, x_{amount} and x_{id} are derived from the use of these variables made by *BankInterface*. Thus, the first three are assigned region $\{p_{bank}\}$ because they are only used in the receive along $p_{bank} \bullet o_{charge}$ and in the internal invocation $p_{bank} \bullet o_{check}$. Of course, the partner name of the endpoint must belong to the region of the variables. Instead, variable x_{id} is assigned region $\{p_{bank}, x_{cust}\}$ because it is also used in the reply to the customer, i.e. activity $x_{cust} \bullet o_{chargeFail}!\langle x_{id}\rangle$. For similar reasons, *CreditRating*, the other subservice of *Bank*, gets annotated as follows

$$
\begin{array}{l}
[\{x_{id}\}^{\{p_{bank}\}}, \{x_{cc}\}^{\{p_{bank}\}}, \{x_a\}^{\{p_{bank}\}}] \\
p_{bank} \bullet o_{check}?\langle x_{id}, x_{cc}, x_a\rangle. \\
[p, o]\,(\ p \bullet o!\langle\rangle \mid p \bullet o?\langle\rangle.\ p_{bank} \bullet o_{checkOK}!\langle x_{id}\rangle \\
\qquad\qquad\qquad + p \bullet o?\langle\rangle.\ p_{bank} \bullet o_{checkFail}!\langle x_{id}\rangle\ )
\end{array}
$$

Thus, the fact that in both subservices variable x_{cc}, which will store the credit card identifier, is assigned region $\{p_{bank}\}$ guarantees that *Bank* will not transimit such information to other (unathorized) services, as required by the policy $\{p_{bank}\}$ attached to *ccNum*.

Suppose instead that service *Bank* (accidentally or maliciously) attempts to reveal the credit card number through some 'internal' operation such as $p_{int} \bullet o!\langle\{x_{cc}\}_r\rangle$, for some region r. For *Bank* to successfully complete the type inference phase, we should have

$p_{int} \in r$. Then, as result of the inference, we would get the annotated variable declaration $[\{x_{cc}\}^{r'}]$, for some region r' with $r \subseteq r'$. Now, the interaction between the typed terms *CardCharge* and *Bank* would be blocked by the runtime checks because the datum sent by *CardCharge* would be annotated as $\{ccNum\}_{\{p_{bank}\}}$ while the region r' of the receiving variable x_{cc} is such that $p_{int} \in r \subseteq r' \not\subseteq \{p_{bank}\}$.

When delivering a datum, we can specify different policies according to the invoked service. For example, when sending the car's current location stored in x_{loc} to services *OrderTowTruck* and *RentCar*, we annotate it with the regions $\{x_{towTruck}\}$ and $\{x_{rentalCar}\}$, respectively. This means that the corresponding service invocations get annotated as follows:

$$x_{towTruck} \bullet o_{orderTow}!\langle p_{car}, \{x_{loc}\}_{\{x_{towTruck}\}}, x_{gps}\rangle$$
$$x_{rentalCar} \bullet o_{redirect}!\langle p_{car}, \{x_{loc}\}_{\{x_{rentalCar}\}}\rangle$$

Notably, the used policies are not fixed at design time, but *depend* on the partner variables $x_{towTruck}$ and $x_{rentalCar}$, and, thus, will be determined by the services that these variables will be bound to as computation proceeds. For example, consider a towing truck service annotated as follows:

$$\begin{aligned} TowTruck \triangleq * [\{x_{cust}\}^{r_1}, \{x_{carGPS}\}^{r_2}, \{x_{garageGPS}\}^{r_3}, o_{checkOK}, o_{checkFail}] \\ p_{towTruck} \bullet o_{orderTow}?\langle x_{cust}, x_{carGPS}, x_{garageGPS}\rangle. \ \ldots \end{aligned}$$

Now, the car's current location can be communicated to the towing truck if, and only if, the region of the variable x_{carGPS} that, after communication, will store the datum and the region of x_{loc} do comply, i.e. $r_2 \subseteq \{p_{towTruck}\}$.

As a final example, the on road services could want to guarantee that critical data sent to the in-vehicle services, such as cost and quality of the service supplied, are not disclosed to competitors. For example, suppose that the towing truck services, like *TowTruck* before, must send the estimated travel time (ETT) to clients. To prevent this datum from being sent to competitor services, ETT is communicated with an attached policy that only authorizes the client partner to access it, as in the following activity

$$x_{cust} \bullet o_{towTruckOK}!\langle\{ETT\}_{\{x_{cust}\}}\rangle$$

Applying the approach to the finance case study. As in the previous examples, we can identify some confidentiality properties also for the finance case study informally presented in Section 2.4.2 and specified in COWS in Section 3.4.2.

From the customer point of view, the service programmer can specify policies stating that the customer's personal information, the security values and the required amount will not be transmitted by the portal to other services, while the customer's balance can be only communicated to the validation service identified by the partner name *validation* (assumed to be known a priori by the customer). This means that the service invocations

performed by the customer get annotated as follows:

$$portal \bullet getCreditRequest!\langle x_{id}, \{customerData\}_{\{portal\}}, \{amount\}_{\{portal\}}, customer\rangle$$
$$portal \bullet securities!\langle x_{id}, \{securityValues\}_{\{portal\}}\rangle$$
$$portal \bullet balance!\langle x_{id}, \{balance\}_{\{portal,validation\}}\rangle$$

As expected, the typed version of the portal services, first of all *InformationUpload*, respect the above policies.

Instead, from the portal point of view, the service programmer can require the customer to not pass to other services the session identifier, to avoid these services act in customer's stead, and the final offer, which has been specifically computed for the customer demands. Therefore, the corresponding invocations performed by the portal get annotated as follows:

$$x_{cust} \bullet logged!\langle key, \{sessionID\}_{\{x_{cust}\}}\rangle$$
$$x_{cust} \bullet offer!\langle x_{id}, \{x_{offer}\}_{\{x_{cust}\}}, x_{motivation}\rangle$$

4.1.4 Concluding remarks

We have introduced a first analytical tool for checking that COWS specifications enjoy some desirable properties concerning the partners, and hence the services, that can safely access any given datum and, in that respect, do not manifest unexpected behaviors. Our type system is quite simple: types are just sets and operations on types are union, intersection, subset inclusion, etc. The language operational semantics only involves types in efficiently implementable checks, i.e. subset inclusions. While implementation of our framework (and, hence, of a type inference algorithm) is currently in progress, we are also working on the definition of a completely static variant where all dynamic checks have been moved to the static phase. This would require a static analysis that gathers information about all the values that each variable can assume at runtime and uses these information to verify the compliance with the specified policies. At the price of a more complex static phase, this approach, on the one hand, would alleviate the runtime checks but, on the other hand, could discard terms that at runtime would behave safely since statically they cannot guarantee to comply with their policies. A relational static analysis [160], which is not a type system, for a fragment of COWS has been introduced in [16]. It can be used for ensuring that, in a COWS term, service invocations do not interfere in malign ways.

Our types are essentially inspired by the 'region types' for Confined-λ of [120] and for global computing calculi of [75]. There are however some noticeable differences. In fact, COWS permits describing systems exchanging heterogeneous data along endpoints, which calls for a more dynamic typing mechanism than that used with communication channels. Moreover, COWS permits annotating only the relevant data while Confined-λ

requires typing any constant, function and channel. The group types, originally proposed for the Ambients calculus [62] and then recast to the π-calculus [61], have purposes similar to our region annotations, albeit they are only used for constraining the exchanges of ambient and channel names. Confinement has been also explored in the context of Java, and related calculi, for confining classes and objects within specific packages [201, 208].

More expressive type disciplines based, e.g., on session types and behavioural types are emerging as powerful tools for taking into account behavioural and non-functional properties of computing systems. In the case of services, they could permit to express and enforce many relevant policies for, e.g., regulating resources usage, constraining the sequences of messages accepted by services, ensuring service interoperability and compositionality, guaranteeing absence of deadlock in service composition, checking that interaction obeys a given protocol. Some of the studies developed for the π-calculus (see e.g. [112, 56, 207, 121, 117, 122, 114]) are promising starting points, but they need non-trivial adaptations to deal with all COWS peculiar features. For example, one of the major problems we envisage concerns the treatment of killing and protection activities and priorities, that are not commonly used in process calculi.

4.2 A logical verification methodology

Modal and temporal logics have long been used to represent properties of concurrent and distributed systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc. (see e.g. [116]). These logics have proved suitable to reason about complex software systems because they only provide abstract specifications of these systems and can thus be used for describing system properties rather than system behaviours. Moreover, the application of temporal logics to the analysis of systems is often supported by software tools (see e.g. [67, 68, 110]). By following this line of research, we introduce a logical verification methodology for checking functional properties of services. The properties are described by means of SocL, a logic specifically designed to express in a convenient way distinctive aspects of services. The verification of SocL formulae over their interpretation domain, abstract representations of service behaviours, is assisted by the on-the-fly model checker CMC[1]. Service behaviours can be initially specified using COWS. Then, by relying on the semantics of the calculus and appropriate transformation rules, CMC can extract the corresponding abstract representations to be used for model checking purposes.

[1] CMC [192] is a tool supporting specification and verification of COWS terms. Besides model checking of SocL formulae, CMC also permits deriving all computations originating from a COWS term in an automated way. A prototypical version of CMC, developed by Franco Mazzanti at ISTI-CNR of Pisa, can be experimented via a web interface available at the address `http://fmt.isti.cnr.it/cmc/`. We refer to Sections 4.2.4 and Appendix A for more details about CMC.

4.2.1 An overview of the verification methodology

Our verification methodology does not put any specific demand on what a service is, rather, for the sake of generality, takes an abstract point of view. Thus, we think of services as software entities which may have an internal state and can interact with each other by, e.g., sending/accepting requests, delivering corresponding responses and, on-demand, cancelling requests. Thus, for example, we say that a service is

1. *available*: if it is always capable to accept a request;
2. *parallel*: if, after accepting a request, before giving a response it can accept further requests;
3. *sequential*: if, after accepting a request, it cannot accept further requests before giving a response;
4. *one-shot*: if, after accepting a request, it cannot accept any further requests;
5. *off-line*: if it provides an unsuccessful response to each received request;
6. *cancelable*: if, before a response has been provided, it permits to cancel the corresponding request;
7. *revocable*: if, after a successful response has been provided, it permits to cancel a request;
8. *responsive*: if it guarantees at least a response to each received request;
9. *single-response*: if, after accepting a request, it provides no more than one response;
10. *multiple-response*: if, after accepting a request, it provides more than one response;
11. *no-response*: if it does never provide a response to any accepted request;
12. *reliable*: if it guarantees a successful response to each received request.

Albeit not exhaustive, the above list contains many desirable properties (see, e.g., [196, 6, 29]) of the externally observable behaviour of services.

The previous properties are stated in terms of the visible *actions* that services may perform. Any of these actions has a *type*, e.g. accept a request, provide a response, etc., and is part of an *interaction* started when a client (possibly another service) firstly invokes one of the operations exposed by the service. At first sight, then, the service properties could be formulated by properly tailoring an action-based temporal logic among those already proposed in the literature of concurrency theory (see e.g. [107, 81, 187]). However, these logics are not expressive enough to, e.g., associate a response action to the request acceptance action that originated the interaction. The possible presence of more request actions, sharing the same type and interaction name, may prevent this association. Indeed,

multiple instances of an interaction can be simultaneously active since service operations can be independently invoked by several clients. Hence, by taking inspiration from SOC emerging standards like WS-BPEL and WS-CDL, to enable the previously mentioned association we use *correlation data* as a third attribute of actions that services can do.

By relying on the actions described above, the branching time, temporal logic SocL, introduced in [83, 84], is capable of expressing in a convenient way peculiar aspects of services and of formalizing the 'abstract' properties previously stated. Being SocL an action- and state-based logic, its formulae predicate properties of systems in terms of states and state changes and of the actions that are performed when moving from one state to another. Indeed, the interpretation domain of SocL formulae are *Doubly Labelled Transition Systems* (L^2TSs, [80]), namely extensions of *Labelled Transition Systems* (LTSs) with a labelling function from states to sets of atomic propositions. The combination of the action paradigm, classically used to describe systems via LTS, with propositions that are true over states, as usually exploited when using Kripke structures as semantic model, facilitates the task of formalizing properties of concurrent systems, where it is often necessary to specify both state information and evolution in time by actions. To assist the verification of SocL formulae over L^2TSs we are developing CMC [192], an on-the-fly model checker, whose use we also describe in this thesis.

Although the proposed methodology handles L^2TSs, we put forward to use COWS as a language for concretely specifying service behaviours. In practice, once the service or SOC system to be analysed has been specified in COWS, to check if the corresponding term enjoys some abstract properties expressed as SocL formulae, the following steps must be performed. Firstly, the LTS representing the semantics of the COWS term is transformed into an L^2TS by labelling each state with the set of activities the COWS term is able to immediately perform from that state. Then, by applying a set of application-dependent abstraction rules, the concrete L^2TS is transformed into a more abstract one. Finally, the SocL formulae are checked over this abstract L^2TS. The overall verification process is supported by CMC.

4.2.2 The logic SocL

SocL is an action- and state-based, branching time, temporal logic, that is a development of the logic UCTL [190]. The two logics mainly differ for the fact that SocL formulae are parameterized by data values and, hence, are more suitable for representing distinctive aspects of services. In this section, we first define SocL and then show how it can be used to formalize the service properties we have mentioned in Section 4.2.1.

4.2.2.1 Syntax and semantics

We start introducing the set of actions which the logic is based upon, then we define the auxiliary logic of actions. As we said before, the actions of the logic should correspond to the actions performed by service providers and service consumers. They are

characterised by three attributes: type, interaction name, and correlation data. Moreover, to enable capturing correlation data used to link together actions executed as part of the same interaction, they may also contain variables, that we call *correlation variables*. In the sequel, we will usually write *val* to denote a generic value and *var* to denote a generic correlation variable. For a given correlation variable *var*, its binding occurrence will be denoted by $\underline{var}$; all remaining occurrences, that are called *free*, will be denoted by *var*.

Definition 4.2.1 (Actions) SocL *actions have the form* $t(i, c)$*, where* t *is the type of the action,* i *is the name of the interaction which the action is part of, and* c *is a tuple of correlation values and variables identifying the interaction (*i *and* c *can be omitted whenever do not play any role). We will say that an action is* closed *if it does not contain variables. We will use Act to denote the set of all actions,* $\underline{a}$ *as a generic element of Act (notation* $\underline{\cdot}$ *emphasises the fact that the action may contain variable binders), and* a *as a generic action without variable binders. We will use* Act^c *to denote the subset of Act that only contains closed actions (i.e. actions without variables) and* A *as a generic subset of* Act^c *(as usual,* $\emptyset$ *denotes the empty set).*

Example 4.2.1 Action *request*(*charge*, 1234, 1) could stand for a *request* action for starting an (instance of the) interaction *charge* which will be identified through the correlation tuple $\langle 1234, 1\rangle$. If some correlation value is unknown at design time, a (binder for a) correlation variable *id* can be used instead, as in the action *request*(*charge*, 1234, $\underline{id}$). This way, during the formula verification process, *id* will capture the corresponding value that can be then used to correlate subsequent actions performed as part of the same interaction. Thus, for example, the *response* action corresponding to the request above could be written as *response*(*charge*, 1234, *id*).

Definition 4.2.2 (Action formulae) *The language* $\mathcal{AF}(Act)$ *of the action formulae on Act is defined as follows:*

$$\gamma \ ::= \ \underline{a} \ \mid \ \chi \qquad\qquad \chi \ ::= \ tt \ \mid \ a \ \mid \ \tau \ \mid \ \neg\chi \ \mid \ \chi \wedge \chi$$

Thus, an action formula γ can be either an action $\underline{a}$, which may contain variable binders, or an action formula χ, which is a boolean compositions of unobservable internal actions τ and actions a without variable binders. As we shall clarify later, the distinction between action formulae γ and χ is motivated by two reasons: (1) some logical operators can accept as argument only action formulae without variable binders, and (2) actions containing variable binders cannot be composed. As usual, we will use *ff* to abbreviate $\neg tt$ and $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$.

Satisfaction of an action formula is determined with respect to a set of closed actions, that represent the observable actions actually executed by the service under analysis. Therefore, since action formulae may contain variables, to define their semantics we introduce the notion of *substitution* and a partial function match that checks matching between an action and a closed action and, if it is defined, returns a substitution.

Definition 4.2.3 (Substitutions) Substitutions*, ranged over by ρ, are functions mapping correlation variables to values and are written as collections of pairs of the form var/val. The empty substitution is denoted by $\emptyset$. Application of substitution ρ to a formula ϕ, written $\phi\,\rho$, has the effect of replacing every free occurrence of var in ϕ with val, for each $var/val \in \rho$.*

Definition 4.2.4 (Matching function) *The partial function* match *from $Act \times Act^c$ to substitutions is defined by structural induction by means of auxiliary functions defined over syntactic subcategories of Act through the following rules:*

$$\begin{array}{l}
\mathsf{match}(t(i,c), t(i,c')) = \mathsf{match}_c(c,c') \\
\mathsf{match}_c((e_1,c_1),(e_2,c_2)) = \mathsf{match}_e(e_1,e_2) \cup \mathsf{match}_c(c_1,c_2) \\
\mathsf{match}_c(\langle\rangle,\langle\rangle) = \emptyset \\
\mathsf{match}_e(\underline{var}, val) = \{var/val\} \\
\mathsf{match}_e(val, val) = \emptyset
\end{array}$$

where (e,c) stands for a tuple with first element e, and $\langle\rangle$ stands for the empty tuple.

Example 4.2.2 Let us consider again the actions introduced in Example 4.2.1. Then, we have $\mathsf{match}(request(charge, 1234, \underline{id}), request(charge, 1234, 1)) = \{id/1\}$ and also $\mathsf{match}(response(charge, 1234, 1), response(charge, 1234, 1)) = \emptyset$. Instead, since the actions have different types, $\mathsf{match}(request(charge, 1234, \underline{id}), response(charge, 1234, 1))$ is not defined.

Definition 4.2.5 (Action formulae semantics) *The satisfaction relation $\models$ for action formulae is defined over a set* A *of closed actions and a substitution ρ.*

- $\mathrm{A} \models \underline{a} \rhd \rho$ *iff* $\exists!\, a' \in \mathrm{A}$ *such that* $\mathsf{match}(\underline{a}, a') = \rho$*;*
- $\mathrm{A} \models \chi \rhd \emptyset$ *iff* $\mathrm{A} \models \chi$*, where the relation* $\mathrm{A} \models \chi$ *is defined as follows:*
  - $\mathrm{A} \models tt$ *holds always;*
  - $\mathrm{A} \models a$ *iff* $a \in \mathrm{A}$
  - $\mathrm{A} \models \tau$ *iff* $\mathrm{A} = \emptyset$*;*
  - $\mathrm{A} \models \neg\chi$ *iff not* $\mathrm{A} \models \chi$*;*
  - $\mathrm{A} \models \chi \wedge \chi'$ *iff* $\mathrm{A} \models \chi$ *and* $\mathrm{A} \models \chi'$.

Notation $\mathrm{A} \models \gamma \rhd \rho$ means: the formula γ is satisfied over the set of closed actions A under substitution ρ. Notably, the semantics of action formulae requires that an action $\underline{a}$ or a matches exactly one closed action in A (see Section 4.2.6.1 for comments and for an extension of the logic where this requirement is relaxed). Moreover, since match is undefined when its first argument is a (free) variable, the semantics of actions containing free occurrences of correlation variables is undefined as well.

Before defining the syntax of the logic, we introduce atomic propositions. They correspond to the properties that can be true over the states of services.

Definition 4.2.6 (Atomic propositions) *SocL atomic propositions have the form $p(i, c)$, where p is the name, i is an interaction name, and c is a tuple of correlation values and (free) variables identifying i (as before, i and c can be omitted whenever do not play any role). Notably, atomic propositions cannot contain variable binders.*
We will use AP to denote the set of all atomic propositions and π as generic element of AP.

Example 4.2.3 Proposition $accepting\_request(charge)$ could indicate that a state can accept requests for interaction *charge*, while proposition $accepting\_cancel(charge, 1234, 1)$ could indicate that a state permits to cancel those requests for interaction *charge* identified by the correlation tuple $\langle 1234, 1\rangle$.

Definition 4.2.7 (SocL syntax) *The syntax of SocL formulae is defined as follows:*

$$\begin{array}{llllllllll} \text{(state formulae)} & \phi & ::= & true & | & \pi & | & \neg\phi & | & \phi \wedge \phi' \quad | \quad E\Psi \quad | \quad A\Psi \\ \text{(path formulae)} & \Psi & ::= & X_\gamma \phi & | & \phi \,{}_\chi U_\gamma\, \phi' & | & \phi \,{}_\chi W_\gamma\, \phi' & & \end{array}$$

E and A are existential and universal (resp.) *path quantifiers*. X, U and W are the *next*, *(strong) until* and *weak until* operators drawn from those firstly introduced in [81] and subsequently elaborated in [145].

Intuitively, the formula $X_\gamma\phi$ says that in the next state of the path, reached by an action satisfying γ, the formula ϕ holds.

Example 4.2.4 Formula $EX_{request(charge,1234,\underline{id})}\, AX_{response(charge,1234,id)}\, true$ can be satisfied by a service capable of accepting in the current state a request for the interaction *charge* and evolving to a state where all actions that it can perform are responses correlated to the accepted request. As another example, the formula $\neg\, EX_{tt}\, true$ means that the service is deadlocked.

The formula $\phi_\chi U_\gamma \phi'$ says that ϕ' holds at some future state of the path reached by a last action satisfying γ, while ϕ holds from the current state until that state is reached and all the actions executed in the meanwhile along the path satisfy χ or τ (i.e. are unobservable).

Example 4.2.5 Formula $E(true\ {}_{response(check,1234)}\, U_{request(stop,1234)}\ (\neg\, EX_{tt}\, true))$ is satisfied by a service that, after a (possible empty) sequence of actions $response(check, 1234)$, reaches a deadlock state by performing the action $request(stop, 1234)$.

The formula $\phi \,{}_\chi W_\gamma\, \phi'$ holds on a path either if the corresponding strong until operator holds or if for all the states of the path the formula ϕ holds and all the actions of the path satisfy χ or τ.

Example 4.2.6 Consider formula $E(true_{response(check,1234)}\, W_{request(stop,1234)}\, (\neg\, EX_{tt}\, true))$, a variant of the formula introduced in Example 4.2.5 where operator W replaces U. This formula can be also satisfied by a service that performs an infinite sequence of $response(check, 1234)$.

Notice that the weak until operator (also called *unless*) is not derivable from the until operator since disjunction or conjunction of path formulae is not expressible in the syntax of SocL, similarly to any other pure branching-time temporal logic.

The interpretation domain of SocL formulae are *Doubly Labelled Transition Systems* [80] over the set of actions *Act* and the set of atomic propositions *AP*, as introduced by the following definition.

Definition 4.2.8 (Doubly Labelled Transition System, L^{2}TS) *An* L^{2}TS *over the set of actions Act and the set of atomic propositions AP is a tuple* $\langle Q, q_0, Act, R, AP, L\rangle$*, where:*

- Q *is a set of states;*
- $q_0 \in Q$ *is the initial state;*
- $R \subseteq Q \times 2^{Act^c} \times Q$ *is the transition relation;*
- $L : Q \longrightarrow 2^{AP}$ *is the labelling function.*

Recall that Act^c is the subset of *Act* that only contains closed actions. Notably, transitions are labelled by sets of actions whilst in the standard definition of L^{2}TS they are labelled by a single action (see Section 4.2.6.1 for further comments). Those transitions labelled by the empty set correspond to execution of 'unobservable' internal actions. As a matter of notation, instead of $(q, \mathrm{A}, q') \in R$ we will sometimes write $q \xrightarrow{\mathrm{A}} q'$.

To define the semantics of SocL, we need the notion of *path* in an L^{2}TS.

Definition 4.2.9 (Path) *Let* $\langle Q, q_0, Act, R, AP, L\rangle$ *be an L^{2}TS and* $q \in Q$*.*

- σ *is a* path *from q if* $\sigma = q$ *(the* empty path *from q) or* σ *is a (possibly infinite) sequence* $(q_0, \mathrm{A}_1, q_1)(q_1, \mathrm{A}_2, q_2)\cdots$ *with* $q_0 = q$ *and* $(q_{i-1}, \mathrm{A}_i, q_i) \in R$ *for all* $i > 0$*.*
- *We write path*(q) *for the set of all paths from q.*
- *If* $\sigma = (q_0, \mathrm{A}_1, q_1)(q_1, \mathrm{A}_2, q_2)\cdots$ *then the* i^{th} *state in* σ*, i.e.* q_{i-1}*, is denoted by* $\sigma(i-1)$ *and the* i^{th} *set of action in* σ*, i.e.* A_i*, is denoted by* $\sigma\{i\}$*.*
- *The concatenation of paths* σ_1 *and* σ_2*, denoted by* $\sigma_1\sigma_2$*, is a partial operation, defined only if* σ_1 *is finite and its final state coincides with the first state of* σ_2*.*

We can now define the semantics of SocL formulae. In fact, the semantics is only defined for *closed* formulae, namely those formulae where any free occurrence of a correlation variable is syntactically preceded by its binding occurrence.

Definition 4.2.10 (SocL semantics) *Let* $\langle Q, q_0, Act, R, AP, L\rangle$ *be an L^{2}TS,* $q \in Q$*, and* $\sigma \in path(q')$ *for some* $q' \in Q$*. The satisfaction relation of closed* SocL *formulae is defined as follows:*

- $q \models \mathit{true}$ *holds always;*
- $q \models \pi$ *iff* $\pi \in L(q)$*;*
- $q \models \neg\phi$ *iff not* $q \models \phi$*;*
- $q \models \phi \wedge \phi'$ *iff* $q \models \phi$ *and* $q \models \phi'$*;*
- $q \models E\Psi$ *iff* $\exists\, \sigma \in \mathit{path}(q) :\ \sigma \models \Psi$*;*
- $q \models A\Psi$ *iff* $\forall\, \sigma \in \mathit{path}(q) :\ \sigma \models \Psi$*;*
- $\sigma \models X_\gamma \phi$ *iff* $\exists \rho : \sigma\{1\} \models \gamma \rhd \rho$*, and* $\sigma(1) \models \phi\rho$*;*
- $\sigma \models \phi\ {}_\chi U_\gamma \phi'$ *iff* $\sigma(0) \models \phi$*, and there exists* $j > 0$ *such that*
  $\exists \rho : \sigma\{j\} \models \gamma \rhd \rho$*,* $\sigma(j) \models \phi'\rho$*, and for all* $0 < i < j$*:*
  $\sigma(i) \models \phi$ *and* $\sigma\{i\} \models \chi$ *or* $\sigma\{i\} = \emptyset$*;*
- $\sigma \models \phi\ {}_\chi W_\gamma \phi'$ *iff* $\sigma(0) \models \phi$ *and*
  either there exists $j > 0$ *such that* $\exists \rho : \sigma\{j\} \models \gamma \rhd \rho$*,* $\sigma(j) \models \phi'\rho$*,*
  and for all $0 < i < j$*:* $\sigma(i) \models \phi$ *and* $\sigma\{i\} \models \chi$ *or* $\sigma\{i\} = \emptyset$
  or for all $j > 0$*:* $\sigma(j) \models \phi$ *and* $\sigma\{j\} \models \chi$ *or* $\sigma\{j\} = \emptyset$*.*

A distinctive feature of SocL is that the satisfaction relation of the next and until operators may define a substitution which is propagated to subformulae. Notably, in the left hand side of the until operators we use χ (i.e., closed actions) instead of γ, to prevent writing such formulae as $\phi\ {}_{request(i,\underline{var})} U_\gamma \phi'$ whose semantics would require $request(i, \underline{var})$ to be performed zero or more times before γ, which could produce undefined or multiple defined bindings on *var*. The distinction between γ and χ, moreover, permits to prevent writing action formulae that contain binding occurrences of correlation variables as arguments of boolean operators. Indeed, if we would try to evaluate, for example, the formula $AX_{request(charge,1234,\underline{id}) \vee request(check,1234)}\, EX_{response(charge,1234,id)}\, \mathit{true}$ over a state q of an L^{2}TS such that $q \xrightarrow{\{request(check,1234)\}} q'$, we would have to check the satisfaction of the subformula $EX_{response(charge,1234,id)}\, \mathit{true}$ over q', but this cannot be verified since this last formula is not closed (as required by the satisfaction relation). All these constraints on the syntax of formulae guarantee that, when checking the satisfaction of a SocL formula, evaluation of the involved action formulae always returns a unique substitution.

Other useful logic operators can be derived as usual. In particular, the ones that we use in the sequel are:

- *false* stands for $\neg\, \mathit{true}$.
- $< \gamma > \phi$ stands for $EX_\gamma\, \phi$; this is the *diamond* operator introduced in [107] and, intuitively, states that it is *possible* to perform an action satisfying γ and thereby satisfy formula ϕ.

- $[\gamma]\,\phi$ stands for $\neg < \gamma > \neg\,\phi$; this is the *box* operator introduced in [107] and states that no matter how a process performs an action satisfying γ, the state it reaches in doing so will *necessarily* have property ϕ.
- $EF\phi$ stands for $E(true\ _{tt}U\phi)$ and means that there is some path that leads to a state at which ϕ holds; that is, ϕ *potentially* holds.
- $EF_\gamma\, true$ stands for $E(true\,_{tt}U_\gamma\, true)$ and means that there is some path that leads to a state reached by a last action satisfying γ; that is, an action satisfying γ will *eventually* be performed;
- $AF_\gamma\, true$ stands for $A(true\,_{tt}U_\gamma\, true)$ and means that an action satisfying γ will be performed in the future along every path; that is, an action satisfying γ is *inevitable*.
- $AG\,\phi$ stands for $\neg\, EF\,\neg\,\phi$ and states that ϕ holds at every state on every path; that is, ϕ holds *globally*.
- Variants of until operators, which do not specify the last action leading to the state at which the formula on the right hand side holds, can be defined as follows:
  - $E(\phi\ _\chi U\ \phi')$ stands for $\phi' \vee E(\phi_\chi U_{\chi\vee\tau}\,\phi')$;
  - $A(\phi\ _\chi U\ \phi')$ stands for $\phi' \vee A(\phi_\chi U_{\chi\vee\tau}\,\phi')$;
  - $E(\phi\ _\chi W\ \phi')$ stands for $\phi' \vee E(\phi_\chi W_{\chi\vee\tau}\,\phi')$;
  - $A(\phi\ _\chi W\ \phi')$ stands for $\phi' \vee A(\phi_\chi W_{\chi\vee\tau}\,\phi')$.

4.2.2.2 A few patterns of service properties

We now show how the service properties presented in Section 4.2.1 can be expressed as formulae in SocL. To do this, we instantiate the set of actions *Act* and the set of atomic propositions *AP* which the logic is based upon as follows.

- *Act* contains (at least) the following five types of actions: *request*, *responseOk*, *responseFail*, *cancel* and *undo*. The intended meaning of the actions is: $request(i,c)$ indicates that the action performed by the service starts the interaction i which is identified by the correlation tuple c; similarly, $responseOk(i,c)$, $responseFail(i,c)$, and $cancel(i,c)$ correspond to actions that provide a successful response, an unsuccessful response, a cancellation, respectively, of the interaction i identified by c; $undo(i,c)$ corresponds to override the effects of a previous request.
- *AP* contains (at least) the atomic propositions *accepting_request*, *accepting_cancel* and *accepting_undo*, whose meaning is obvious.

For the sake of readability, in the formalization of the properties we consider correlation tuples composed of only one element (their generalization to tuples of any length is obvious).

1. - - *Available* service - -

   $AG\,(accepting\_request(i))$.

   This formula means that in every state the service may accept a request. A weaker interpretation of service availability, meaning that the service accepts a request infinitely often, is given by the formula $AG\,AF\,(accepting\_request(i))$.

2. - - *Parallel* service - -

   $AG\,[request(i, \underline{var})]$
   $E(true_{\neg\,(responseOk(i,var)\vee responseFail(i,var))}\,U\ accepting\_request(i))$.

3. - - *Sequential* service - -

   $AG\,[request(i, \underline{var})]$
   $A(\neg\ accepting\_request(i)\ _{tt}U_{responseOk(i,var)\vee responseFail(i,var)} true)$.

4. - - *One-shot* service - -

   $AG\,[request(i)]\,AG\,\neg\,accepting\_request(i)$.

5. - - *Off-line* service - -

   $AG\,[request(i, \underline{var})]\,AF_{responseFail(i,var)}\,true$.

6. - - *Cancelable* service - -

   $AG\,[request(i, \underline{var})]$
   $A(accepting\_cancel(i, var)\ _{tt}W_{responseOk(i,var)\vee responseFail(i,var)} true)$.

   This formula means that the service is ready to accept a cancellation required by the client (fairness towards the client). A different formulation is given by the formula
   $AG\,[responseOk(i, \underline{var})]\,\neg EF < cancel(i, var) > true$
   meaning that the service cannot accept a cancellation after responding to a request (fairness towards the service).

7. - - *Revocable* service - -

   $EF_{responseOk(i,\underline{var})}\,EF(accepting\_undo(i, var))$

   Again, we can have two interpretations. While the previous formula expresses a sort of weak revocability, the following one corresponds to a stronger interpretation
   $AG\,[responseOk(i, \underline{var})]\,A(accepting\_undo(i, var)\ _{tt}W_{undo(i,var)} true)$.

8. - -*Responsive* service - -

   $AG\,[request(i, \underline{var})]\,AF_{responseOk(i,var)\vee responseFail(i,var)}\,true$.

9. - - *Single-response* service - -

   $AG\,[request(i, \underline{var})]$
   $\neg EF_{responseOk(i,var)\vee responseFail(i,var)}\,EF_{responseOk(i,var)\vee responseFail(i,var)}\,true$.

10. - - *Multiple-response* service - -

$$AG\,[request(i,\underline{var})]\; AF_{responseOk(i,var)\vee responseFail(i,var)}\; AF_{responseOk(i,var)\vee responseFail(i,var)}\; true.$$

11. - - *No-response* service - -

$$AG\,[request(i,\underline{var})]\; \neg EF_{responseOk(i,var)\vee responseFail(i,var)}\; true.$$

12. - - *Reliable* service - -

$$AG\,[request(i,\underline{var})]\; AF_{responseOk(i,var)}\; true.$$

Notably, the response belongs to the same interaction i of the accepted request and they are correlated by the variable var.

The SocL formulation of the above properties is instructive in that it witnesses that the natural language descriptions of the properties can sometimes be interpreted in different ways: therefore, formalization within the logic enforces a choice among different interpretations. Notably, the formulation is given in terms of abstract actions and states thus, rather than specific properties, the properties we have considered so far represent sort of generic patterns or classes of properties. In other words, from time to time, type/name, interaction and correlation tuple of actions and propositions have to be projected on the actual actions performed by the specific service to be analysed. They, however, can be easily instantiated, as shown in Section 4.2.5, and such instantiation can be in principle automated. This is helpful e.g. to hide the temporal logic details to a developer only interested to know if the service he has designed is, say, responsive. Anyway, as shown in Section 4.2.4, the developer still keeps full control on the mapping from the (concrete) actions and states of the service to the abstract actions and states in terms of which the properties are formulated.

4.2.3 L^{2}TS semantics for COWS terms

The operational semantics of COWS associates an LTS to a COWS term. We have seen instead that SocL is interpreted over L^{2}TSs. We need therefore to transform the LTS associated to a COWS term into an L^{2}TS by defining a proper labelling for the states of the LTS. This is done by enriching the LTS with a function labelling each state with the set of activities that any active subterm of the COWS term corresponding to that state would be able to perform immediately. Of course, the transformation preserves the structure of the original LTS. Both in the LTS and in the obtained L^{2}TS, being computational steps, transitions are labelled by actions of the form $p \bullet o\,\bar{v}$ or $\dagger$. In fact, for the sake of simplicity, labels of the form $p \bullet o\,\emptyset\,\ell\,\bar{v}$ generated by the operational semantics are written $p \bullet o\,\bar{v}$; indeed, the two omitted components can be safely removed at the end of transitions' inference phase since they are only used to support implementation of global scope of variables and precedence among concurrent actions. In the next figures, we shall use arrows of the form $\dashrightarrow$ to denote multi-step computations, and arrows of the form $\cdots\!\!\rightarrow$ to denote further unspecified computations.

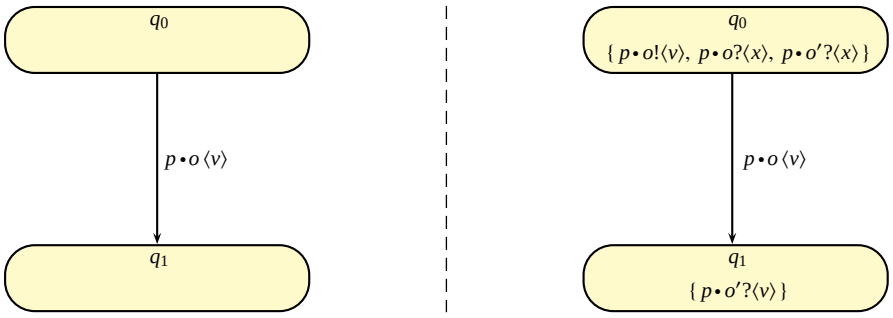

Figure 4.2: From LTS to L^{2}TS

We explain this transformation through a simple example. Consider the following COWS term:

$$p \bullet o!\langle v \rangle \mid [x]\,(\, p \bullet o?\langle x \rangle \mid p \bullet o'?\langle x \rangle.\,[y]\, p \bullet o''?\langle x, y \rangle\,)$$

The corresponding LTS and L^{2}TS are as in Figure 4.2. The LTS on the left hand side says that the term can actually perform only the computation corresponding to the communication along the endpoint $p \bullet o$. However, besides the activities $p \bullet o!\langle v \rangle$ and $p \bullet o?\langle x \rangle$, the COWS term can potentially perform also the receive activity $p \bullet o'?\langle x \rangle$. Thus, to record this information, the state q_0 of the L^{2}TS on the right hand side is labelled by a set containing all the three potential activities. Similarly, the state q_1 is labelled by the only potential receive activity $p \bullet o'?\langle v \rangle$. Notably, the receive $p \bullet o''?\langle x, y \rangle$, that cannot be immediately performed by the term, is missing both in the LTS and in the L^{2}TS.

Consider now the bank service presented in Section 3.4.1 and modelled in COWS by the term

$$[o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating}\,)$$

where *BankInterface* is a service publicly invocable by customers, while *CreditRating* is an 'internal' service that can only interact with *BankInterface*. To show the behaviour of the bank service we consider here the terms Client_1 and Client_2 that model a pair of mutually dependent requests for charging a customer's credit card with some amount. In fact, we want to model a sort of 'transactional' behaviour: for a charge operation to succeed, both requests by Client_1 and Client_2 must succeed; otherwise, no effect will be produced. Thus, the COWS term representing the considered scenario is

$$[o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating}\,) \mid [k]\,(\, \mathit{Client}_1 \mid \mathit{Client}_2\,)$$

where the customer processes are defined as follows:

$$\begin{aligned} \mathit{Client}_1 \triangleq\ & p_{bank} \bullet o_{charge}!\langle p_C, 1234, 100, id_1 \rangle \\ & \mid\ p_C \bullet o_{chargeOK}?\langle id_1 \rangle.\, s_1 + p_C \bullet o_{chargeFail}?\langle id_1 \rangle\,.(\, \{\!|\, p_{bank} \bullet o_{revoke}!\langle id_2 \rangle \,|\!\}\ \mid \mathbf{kill}(k)\,) \end{aligned}$$

$$Client_2 \triangleq p_{bank} \bullet o_{charge}!\langle p_C, 1234, 200, id_2\rangle \\ \mid\ p_C \bullet o_{chargeOK}?\langle id_2\rangle.\ s_2 + p_C \bullet o_{chargeFail}?\langle id_2\rangle\ .(\ \{\!|p_{bank} \bullet o_{revoke}!\langle id_1\rangle|\!\}\ \mid \mathbf{kill}(k)\ )$$

The LTS of the above bank scenario is shown in Figure 4.3, while the concrete L^2TS obtained by applying the transformation is shown in Figure 4.4. Transitions of both systems are labelled by 'concrete' information generated by the operational rules of the calculus. Thus, since we are interested in verifying abstract properties of services, such as those shown in Section 4.2.2.2, we need to abstract away from unnecessary details. This is done by using a set of suitable abstraction rules that permit to replace concrete actions on the transitions with 'abstract' actions of SocL, i.e. $request(i, c)$, $responseOk(i, c)$, $responseFail(i, c)$, $cancel(i, c)$ and $undo(i, c)$. Similar rules permit to replace the concrete activities labelling the states with predicates of SocL, e.g. $accepting\_request(i)$, $accepting\_cancel(i, c)$, and $accepting\_undo(i, c)$. Of course, in doing these further transformations, different concrete actions can be mapped into the same SocL action. Moreover, the transformations may involve only those concrete actions/activities that are considered worthwhile to be observed to carry on the analysis of interest. Indeed, those that are not replaced by their abstract counterparts may not be observed.

The abstraction procedure must however preserve those names and values occurring within concrete actions/activities of COWS specifications that are important to express properties of service behaviour. To capture such names and values, transformation rules can make use of 'metavariables', written as names starting with the character "$"; otherwise, they can use the wildcard " $*$ ". To avoid cumbersome notations, we refrain from introducing new symbols and write $\dot{v}$ to indicate that v can be either a value, or a metavariable, or the wildcard (this notation also applies to tuples, actions and predicates with a similar meaning). Anyway, take into account that the wildcard can only occur in the left hand side of the abstraction rules.

Formally, abstraction rules follow the templates:

$$\begin{array}{rlcll} Action & \dot{p} \bullet \dot{o}, \dot{\bar{v}} & \rightarrow & \dot{a} & (1) \\ State & \dot{p} \bullet \dot{o}?\dot{\bar{w}} & \rightarrow & \dot{\pi} & (2) \\ State & \dot{p} \bullet \dot{o}!\dot{\bar{v}} & \rightarrow & \dot{\pi} & (3) \end{array}$$

where $\dot{a}$ is a closed action and $\dot{\pi}$ is a closed atomic proposition of the logic SocL (except for, possibly, the occurrence of some of the metavariables introduced in the left hand side of the rule). Rules following the template (1) apply to concrete actions of transitions, while the remaining ones apply to concrete activities labelling states.

To define the effect of the application of abstraction rules to an L^2TS, we exploit an auxiliary function $match_t(\_, \_)$, that checks the matching between tuples of the form $\langle \dot{p}, \dot{o}, \dot{\bar{v}}\rangle$ drawn from the left hand sides of abstraction rules and tuples of the form $\langle p, o, \bar{w}\rangle$ drawn from concrete actions/activities. This function is defined by the following rules:

$$match_t(v, v) = \emptyset \qquad match_t(*, v) = \emptyset \qquad match_t(\$n, v) = \{\$n/v\}$$

$$match_t(*, x) = \emptyset \qquad \frac{match_t(\dot{v}_1, w_1) = \rho_1 \quad match_t(\dot{\bar{v}}_2, \bar{w}_2) = \rho_2}{match_t((\dot{v}_1, \dot{\bar{v}}_2), (w_1, \bar{w}_2)) = \rho_1 \uplus \rho_2}$$

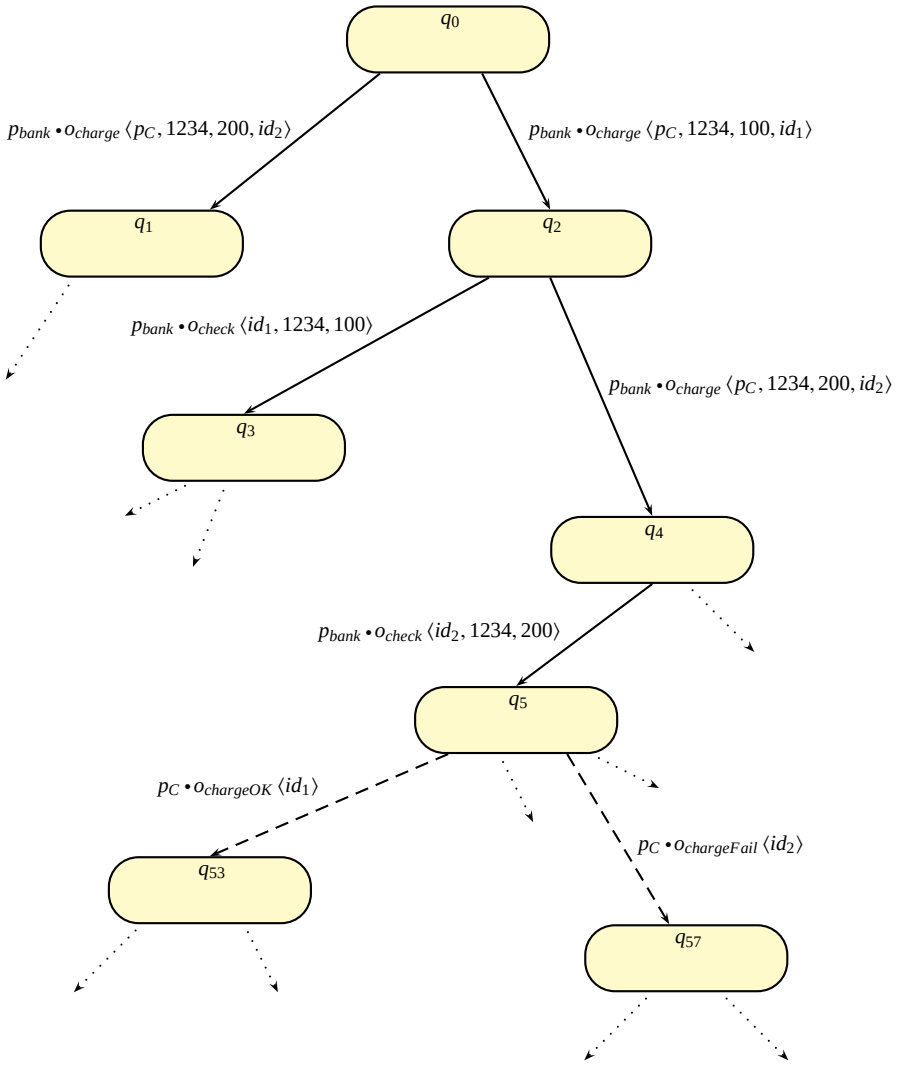

Figure 4.3: Excerpt of the LTS for the bank scenario

When using the left hand side of a rule to build the tuple $\langle \dot{p}, \dot{o}, \bar{\dot{v}} \rangle$, if any of $\dot{p}$ or $\dot{o}$ is missing, it is replaced by the wildcard, while if $\bar{\dot{v}}$ is missing, it is replaced by one or more tuples of wildcards of appropriate length (as drawn from the COWS specification according to the tuples of values that can be exchanged along the endpoints matching $\dot{p} \bullet \dot{o}$). In practice, each abstraction rule applies to the largest possible set of concrete actions/activities according to function $match_t(\_, \_)$. Omitting any of the elements in the left hand side of the rule corresponds then to enlarging its application domain.

For example, the abstract L^{2}TS of the bank scenario shown in Figure 4.5 is obtained by applying to the concrete L^{2}TS of Figure 4.4 the following abstraction rules:

$$Action \quad p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle \quad \rightarrow \quad request(charge, \$id)$$

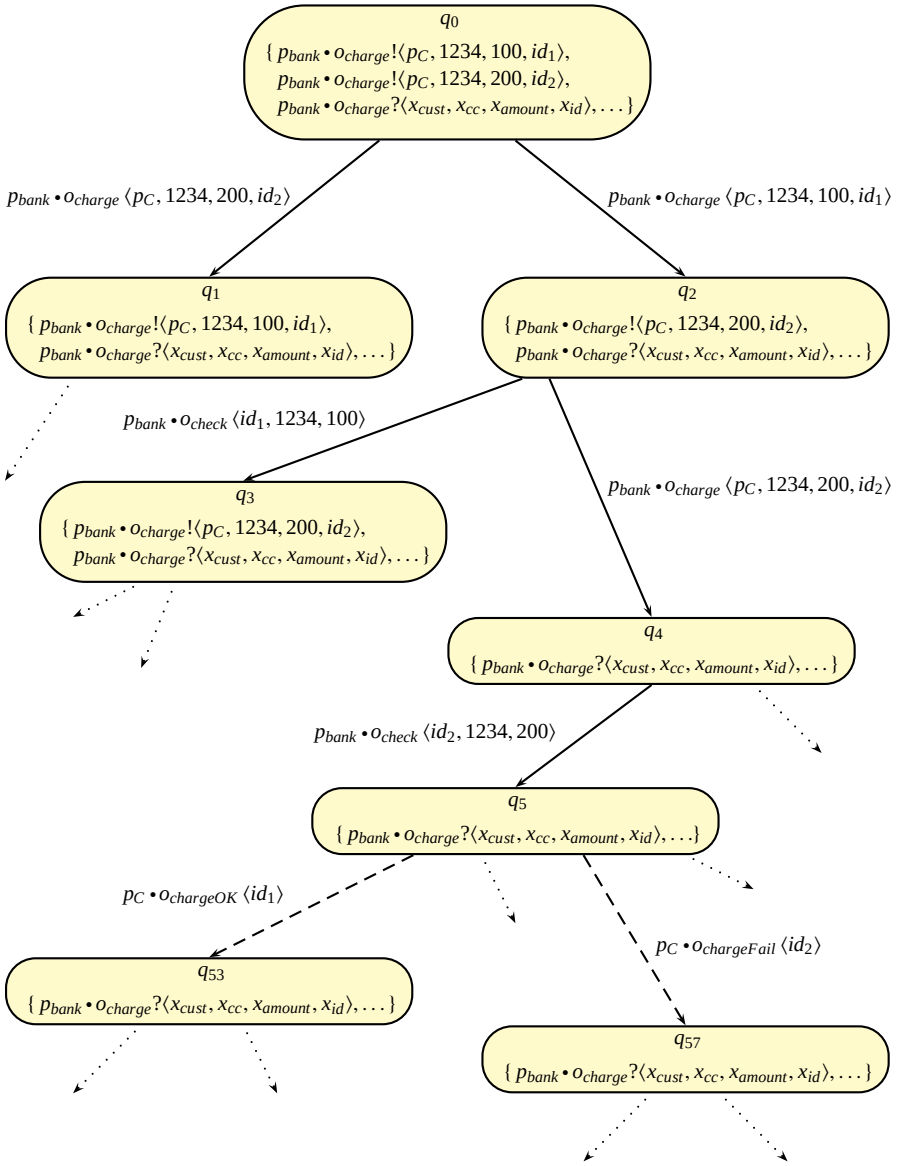

Figure 4.4: Excerpt of the L^2TS for the bank scenario with concrete labels

$$\begin{array}{rlcl} \mathit{Action} & * \bullet o_{chargeOK}, \langle \$id \rangle & \rightarrow & \mathit{responseOk}(\mathit{charge}, \$id) \\ \mathit{Action} & * \bullet o_{chargeFail}, \langle \$id \rangle & \rightarrow & \mathit{responseFail}(\mathit{charge}, \$id) \\ \mathit{State} & p_{bank} \bullet o_{charge}? & \rightarrow & \mathit{accepting\_request}(\mathit{charge}) \end{array}$$

Thus, as a consequence of the application of the first rule, the concrete action $p_{bank} \bullet o_{charge}, \langle p_C, 1234, 200, id2\rangle$ that matches the left hand side of the rule producing the substitution $\{\$id/id2\}$, is replaced by the SocL abstract action $\mathit{request}(\mathit{charge}, id2)$ that is obtained by applying the produced substitution to the right hand side of the rule.

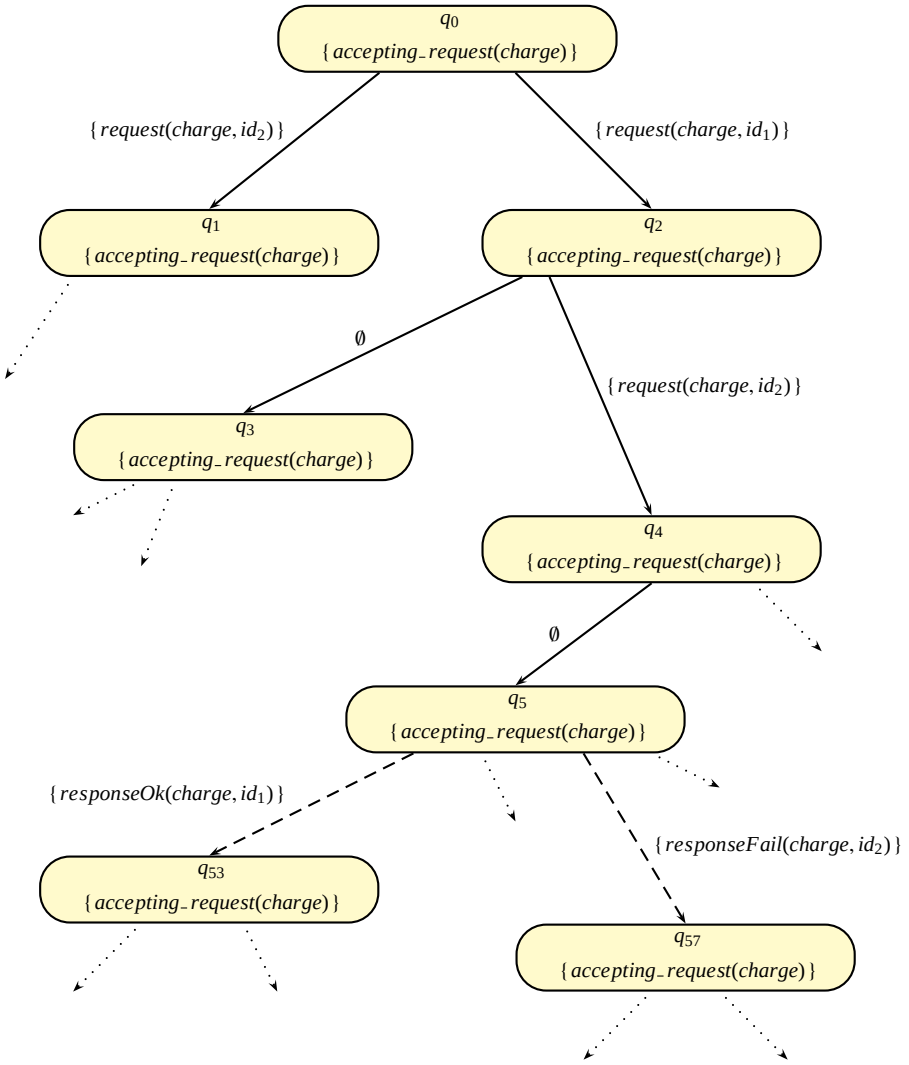

Figure 4.5: Excerpt of the L^2TS for the bank scenario with abstract labels

Similarly, as a consequence of the application of the last rule, the concrete action $p_{bank} \bullet o_{charge}?, \langle x_{cust}, x_{cc}, x_{amount}, x_{id} \rangle$ labelling state q_0 and matching the left hand side of the rule is replaced by the SocL atomic proposition $accepting\_request(charge)$. Notably, concrete actions corresponding to (internal) communications between the bank subservices are not transformed and, thus, become unobservable (the corresponding action in the abstract L^2TS is $\emptyset$).

Of course, the sets of transformation rules are not defined once and for all, but are application-dependent and, thus, must be defined from time to time. Indeed, they embeds information, like the intended semantics of each action and the predicates on the states, that are not coded into the COWS specification. Due to the low level of the COWS actions,

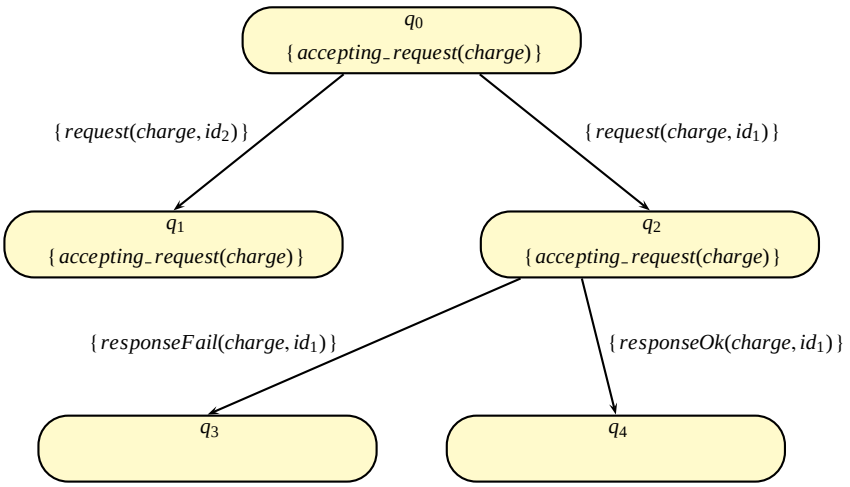

Figure 4.6: An example of L^2TS

a preliminary analysis of the behaviour may be needed to tell the abstract type of the actions, as well for defining the state predicates: this requires a certain knowledge of the actual meaning of the designed services, although heuristics can be defined to support this analysis.

4.2.4 Model checking COWS specifications

To assist the verification process of SocL formulae over L^2TS, we are contributing to the development of CMC, a model checker for SocL that can be used to verify properties of services specified in COWS. A prototypical version of CMC can be experimented via a web interface available at the address `http://fmt.isti.cnr.it/cmc/`.

CMC is implemented by exploiting an efficient on-the-fly algorithm whose complexity is comparable to that of the best on-the-fly model checking algorithms. Therefore, the complexity is linear with respect to the size of the state space and the number of operators of the formula [188, 23, 86]. Indeed, depending on the formula to be checked, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result. Moreover, in case of parametric formulae, only a subset of their possible instantiations will be generated as requested by the on-the-fly evaluation.

The basic idea behind CMC is that, given a state of an L^2TS, the validity of a SocL formula on that state can be established by checking the satisfiability of the state predicates, by analyzing the transitions allowed in that state, and by establishing the validity of some subformulae in some/all of the next reachable states.

To show the peculiarity of our methodology with respect to parametric formulae evaluation, we illustrate the process of establishing the satisfiability of the SocL formula

$$\phi = EX_{request(charge,\underline{id})}\, AX_{responseOK(charge,id)}\, true$$

```
Evaluate (F : Formula, Start : State, Current : State) is
  if we have already done this computation and its result is available,
  i.e. (⟨F, Start, Current⟩ → Result) has already been computed then
      return the already known result;
  else if we were already computing the value of exactly this computation,
  i.e. (⟨F,Start,Current⟩ → inprogress) has already been computed then
      return True or False depending on max or min fixed point semantics;
  else
      keep track that we started to compute the value of this computation,
      i.e. set (⟨F, Start, Current⟩ → inprogress);
      foreach subformula F′ and next state S′ to be computed do loop
        if F′ ≠ F (i.e. this is a syntactically nested subformula) then
            call Evaluate(F′, S′, S′);
        else if F′ = F ( i.e. this is just a recursive evaluation of F) then
            call Evaluate(F, Start, S′);
        end
        if the result of the call suffices to establish the final result then
            exit from the loop;
        end
      end loop
      At this point we have in any case a final result. We keep track of the
      result of this computation (e.g. set (⟨F, Start, Current⟩ → Result)).
      return the final result;
  end
end Evaluate;
```

Table 4.5: Simplified schema of the evaluation process

on the abstract L^{2}TS of Figure 4.6; namely, we must establish that $q_0 \models \phi$. Since $q_0 \xrightarrow{\{request(charge,id_1)\}} q_2$, and $\{request(charge, id_1)\} \models request(charge, \underline{id}) \rhd \{id/id_1\}$, we have that the label of the transition from q_0 to q_2 satisfies the first action formula by producing the substitution $\{id/id_1\}$. By using this substitution, we construct the new subformula

$$\phi' = (AX_{responseOK(charge,id)}\, true)\{id/id_1\} = AX_{responseOK(charge,id_1)}$$

thus we are left to check if $q_2 \models \phi'$, i.e. if the path formula $X_{responseOK(charge,id_1)}\, true$ is satisfied by all outgoing transitions from q_2. In our case we have two outgoing transitions: $q_2 \xrightarrow{\{responseOK(charge,id_1)\}} q_4$ satisfies the path formula under a trivial matching between the action formula and the action on the transition, while $q_2 \xrightarrow{\{responseFail(charge,id_1)\}} q_3$ does not satisfy the action formula under any matching. Therefore, we conclude that ϕ', and hence ϕ, is not satisfied.

The simplified schema shown in Table 4.5 gives an idea of the algorithmic structure of the evaluation process: `F` denotes the SocL formula (or subformula) to be evaluated, `Start` denotes the state in which the (recursive) evaluation of `F` was started and `Current`

```
Evaluate (AX_γ φ, StartState, CurrentState) is
  if we have already done this computation and its result is available,
    i.e. ⟨AX_γ φ, CurrentState, CurrentState⟩ → Result
    has already been computed then
      return the already known Result
  end
  if there are no outgoing transitions from CurrentState then
      set ⟨AX_γ φ, CurrentState, CurrentState⟩ → False
      return False
  end
  Result := True
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, γ) then
        TargetState := Transition.TargetState
        ρ := TransitionBindings(Transition.Label, γ)
        φ' := ApplySubstitution(φ, ρ)
        Result := Evaluate(φ', TargetState, TargetState)
        if Result = False then
            exit
        end
    else
        Result := False
        exit
    end
  end loop
  set ⟨AX_γ φ, CurrentState, CurrentState⟩ → Result
  return result
end Evaluate;
```

Table 4.6: More detailed schema of the evaluation process for *AX* operator

denotes the current state in which the evaluation of F is being continued. This schema has been extended with appropriate data-collection activities in order to be able to produce, in the end, also a clear and detailed explanation of the returned results (i.e. a *counterexample*), and with appropriate formula instantiation activities in order to deal with parametric formulae. Table 4.6 shows a more detailed view of the same schema which refers to the specific case of evaluation of the 'universal quantified next' operator *AX*, while Table 4.7 shows a more detailed view of the same schema which refers to the specific case of evaluation of the 'existential quantified until' operator $E(\phi\ {}_{\chi}U_{\gamma}\ \phi')$.

In case of infinite state space, this algorithm may fail to produce a result even when a result could actually be deduced in a finite number of steps. This is a consequence of its 'depth-first' recursive structure. The solution taken to solve this problem consists of adopting a bounded model-checking approach [25], i.e. the evaluation is started assuming a certain value as limit of the maximum depth of the evaluation. In this case, if a formula is given as result of the evaluation within the requested depth, then the result holds for the whole system; otherwise the maximum depth is increased and evaluation is subsequently

```
Evaluate (E φ1 χU_γ φ2, StartState, CurrentState) is
  if we have already done this computation and its definitive result is available,
  i.e. ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → Result
  has already been computed then
      return the already known Result
  end
  if we have already started this computation which is still in progress,
  i.e. ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → InProgress
  then
      return False  accordin to its min fixpoint semantics
  end
  set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → InProgress
  if  there are no outgoing transitions from CurrentState then
      set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → False
      return False
  end
  Result := Evaluate(φ1, CurrentState, CurrentState)
  if Result = False then
      set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → False
      return False
  end
  check for possible structural induction
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, γ) then
        TargetState := Transition.TargetState
        ρ := TransitionBindings(Transition.Label, γ)
        φ2′ := ApplySubstitution(φ2, ρ)
        Result := Evaluate(φ2′, TargetState, TargetState)
        if Result = True then
            set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → True
            return True
        end
    end
  end loop
  check for possible continuation of recursion
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, χ) then
        TargetState := Transition.TargetState
        Result := Evaluate(E φ1 χU_γ φ2, StartState, TargetState)
        if Result = True then
            set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → True
            return True
        end
    end
  end loop
  set ⟨E φ1 χU_γ φ2, StartState, CurrentState⟩ → False
  return False
end Evaluate;
```

Table 4.7: More detailed schema of the evaluation process for $E(\phi\ {}_{\chi}U_{\gamma}\ \phi')$ operator

retried (preserving all useful subresults that were already found). This approach, initially introduced to overcome the problem of infinite state machines, turns out to be quite useful also for another reason. By setting a small initial maximum depth and a small automatic increment of this limit at each re-evaluation failure, once we finally find a result then we have a reasonable (almost minimal) explanation for it, and this is very useful also in the case of finite states machines.

4.2.5 Analysis of the case studies

Now, we demonstrate usability of our methodology by specifying and analysing the automotive and finance case studies specified in COWS in Sections 3.4.1 and 3.4.2.

4.2.5.1 Automotive case study

We start the analysis of the automotive scenario by verifying if its main service, i.e. the engine failure recovery service executed by *Orchestrator*, enjoys the abstract properties expressed as SocL formulae in Section 4.2.2.2. To this aim, we focus our observations on the recovery service by applying the following abstraction rules:

$$
\begin{array}{rlcl}
Action & \$car \bullet o_{engineFailure} & \rightarrow & request(road\_assistance, \$car) \\
Action & \$car \bullet o_{towTruckOK} & \rightarrow & responseOk(road\_assistance, \$car) \\
Action & \$car \bullet o_{rentalCarOK} & \rightarrow & responseOk(road\_assistance, \$car) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow & responseFail(road\_assistance, \$car) \\
Action & \$car \bullet o_{notFound} & \rightarrow & responseFail(road\_assistance, \$car) \\
Action & \$car \bullet o_{garageFail} & \rightarrow & responseFail(road\_assistance, \$car) \\
Action & \$car \bullet o_{rentalCarFail} & \rightarrow & responseFail(road\_assistance, \$car) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow & responseFail(road\_assistance, \$car) \\
State & \$car \bullet o_{engineFailure}? & \rightarrow & accepting\_request(road\_assistance)
\end{array}
$$

According to this abstraction, the service accepts a request for the interaction $road\_assistance$ when it receives an engine failure signal, and replies with a positive response when an order (of garage/tow truck or of rental car) succeeds. Indeed, in this case, the driver may continue its journey. If during the recovery phase some operation fails (e.g. the bank does not accept the request of charging the driver's credit card), the service replies with a negative response. More specifically, in case of complete success, two actions $responseOk(road\_assistance, \$car)$ are performed and no action $responseFail(road\_assistance, \$car)$ is observed, while, in case of complete failure, some actions $responseFail(road\_assistance, \$car)$ are performed and no action $responseOk(road\_assistance, \$car)$ is observed. By means of the last abstraction rule, each state that can accept requests for interaction $road\_assistance$ is labelled by the atomic proposition $accepting\_request(road\_assistance)$.

Now, by using CMC, the SocL formulae of Section 4.2.2.2 can be checked over the obtained abstract L^2TS; the instantiation of those generic formulae over the recovery service has been obtained by simply replacing any occurrence of the generic interaction name

i with *road_assistance*. In addition, we will check some variants of such formulae, to understand in more detail the properties of the service under analysis.

1. - - *Available* service - -

   $AG\,(accepting\_request(road\_assistance))$

   This abstract property clearly does not hold for the considered service because, being the service of the one-shot kind, it will not be permanently available.

   However, we can check whether the service is at least available until it is invoked.

   $A(accepting\_request(road\_assistance)\,U_{request(road\_assistance)}\ true)$

   This second property, as intuitively expected, holds for our specification.

2. - - *Parallel* service - -

   $AG\,[request(road\_assistance, \underline{var})]$
   $E(true\ _{\neg\,(responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var))}$
   $U\ accepting\_request(road\_assistance))$

   Also in this case, being our service activated only once, it does not make much sense to check whether the service becomes available again after its first activation (and in particular even before providing a response to the initial request). Hence, this property correctly does not hold for our specification.

3. - - *Sequential* service - -

   $AG\,[request(road\_assistance, \underline{var})]$
   $A(\neg\, accepting\_request(road\_assistance)\,{}_{tt}$
   $U_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true)$

   In this case, although the service is of the one-shot kind, it satisfies the sequentiality property because it is true that it will not be available at least until a response is provided.

4. - - *One-shot* service - -

   $AG\,[request(road\_assistance)]\,AG\,\neg\, accepting\_request(road\_assistance)$

   After accepting a first request the service becomes permanently unavailable, hence this property holds. Actually, in our case also a stronger version of the above property, stating in addition that a first request will always be accepted, holds.

   $AF_{request(road\_assistance)}\,AG\,\neg\, accepting\_request(road\_assistance)$

5. - - *Off-line* service - -

   $AG\,[request(road\_assistance, \underline{var})]\,AF_{responseFail(road\_assistance,var)}\ true$

   The recovery service is not off-line because there exist some execution paths that lead to a complete success, i.e. no negative responses are provided. Hence, the above property does not hold.

6. - - *Cancelable* service - -

$$AG\,[request(road\_assistance, \underline{var})]$$
$$A(accepting\_cancel(road\_assistance, var)_{\,tt}$$
$$W_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true)$$

This property is trivially false, since the recovery service does not permit cancelling requests, i.e. the atomic proposition $accepting\_cancel(road\_assistance, var)$ does not hold in any state.

7. - - *Revocable* service - -

$$EF_{responseOk(road\_assistance,\underline{var})}\ EF(accepting\_undo(road\_assistance, var))$$

The recovery service does not accept undo requests, i.e. the atomic proposition $accepting\_undo(road\_assistance, var)$ does not hold in any state. Hence, this property is trivially false.

8. - -*Responsive* service - -

$$AG\,[request(road\_assistance, \underline{var})]$$
$$AF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true$$

The service is responsive because it always provides a response (at least one) for each accepted request.

9. - - *Single-response* service - -

$$AG\,[request(road\_assistance, \underline{var})]$$
$$\neg\, EF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}$$
$$EF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true$$

This property, stating that for each request a single response is provided, is false because the service provides generally more than one response (for example, two positive responses in case of complete success).

10. - - *Multiple-response* service - -

$$AG\,[request(road\_assistance, \underline{var})]$$
$$AF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}$$
$$AF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true$$

This property, which states that each request has at least two separate responses, does not hold for the service, because in some cases, e.g. when the credit card charge is denied by the bank or the discovery phase fails, only one response is provided.

11. - - *No-response* service - -

$$AG\,[request(road\_assistance, \underline{var})]$$
$$\neg\, EF_{responseOk(road\_assistance,var)\ \vee\ responseFail(road\_assistance,var)}\ true$$

In at least one case (actually, in all cases) the service provides a response, hence

this property clearly does not hold.

12. - - *Reliable* service - -

$$AG\ [request(road\_assistance, \underline{var})]\ AF_{responseOk(road\_assistance,var)}\ true$$

The service is not reliable because it may produce no positive responses to a request (e.g. when the credit card charge is denied by the bank).

We can also investigate in more detail the request-response relation for the recovery service. To do this, firstly we define a different abstraction of the scenario by applying the following rules:

$$\begin{array}{llll}
Action & \$car \bullet o_{engineFailure} & \rightarrow & request(road_assistance, \$car) \\
Action & \$car \bullet o_{towTruckOK} & \rightarrow & responseOk(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarOK} & \rightarrow & responseOk(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow & responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow & responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{notFound} & \rightarrow & responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{notFound} & \rightarrow & responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{garageFail} & \rightarrow & responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarFail} & \rightarrow & responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow & responseFail(road_assistance, \$car, truckGarage)
\end{array}$$

The obtained L^{2}TS differs from that previously introduced for the presence of a further argument (that can be either *truckGarage* or *rentalCar*, and indicates which order succeeds or fails) in the correlation tuple of the response actions. Now, we can verify for example if, once requested, the service always provides at least one response about the status of the garage/tow truck ordering and at least one response about the status of the car renting.

$$AG\ [request(road\_assistance, \underline{var})]$$
$$AF_{responseOk(road\_assistance,var,truckGarage)\ \vee\ responseFail(road\_assistance,var,truckGarage)}\ true$$

$$AG\ [request(road\_assistance, \underline{var})]$$
$$AF_{responseOk(road\_assistance,var,rentalCar)\ \vee\ responseFail(road\_assistance,var,rentalCar)}\ true$$

Similarly, we can verify that a positive response is never followed by a negative one (and vice versa) for the same order:

$$AG[responseOk(road\_assistance, \underline{var}, \underline{order})]$$
$$\neg\ EF_{responseFail(road\_assistance,var,order)}\ true$$

$$AG[responseFail(road\_assistance, \underline{var}, \underline{order})]$$
$$\neg\ EF_{responseOk(road\_assistance,var,order)}\ true$$

All these last four properties are indeed satisfied by the second abstraction of the service.

Analysis of other services of the automotive scenario. By changing again the abstraction rules applied to the concrete L^2TS modelling the automotive scenario, we can verify the above abstract properties (and possibly some specific variants of them) also over other services appearing in the scenario. For example, we consider here *GpsSystem*, *Bank* and *RentCar*, and apply the following rules:

Action	$\$car \bullet o_{reqLoc}$	$\rightarrow$	$request(gps, \$car)$
Action	$\$car \bullet o_{respLoc}$	$\rightarrow$	$responseOk(gps, \$car)$
State	$\$car \bullet o_{reqLoc}?$	$\rightarrow$	$accepting\_request(gps)$
Action	$p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle$	$\rightarrow$	$request(charge, \$id)$
Action	$* \bullet o_{chargeOK}, \langle \$id \rangle$	$\rightarrow$	$responseOk(charge, \$id)$
Action	$* \bullet o_{chargeFail}, \langle \$id \rangle$	$\rightarrow$	$responseFail(charge, \$id)$
Action	$p_{bank} \bullet o_{revoke}, \langle \$id \rangle$	$\rightarrow$	$undo(charge, \$id)$
State	$p_{bank} \bullet o_{charge}?$	$\rightarrow$	$accepting\_request(charge)$
State	$p_{bank} \bullet o_{revoke}?\langle \$id \rangle$	$\rightarrow$	$accepting\_undo(charge, \$id)$
Action	$p_{rentalCar1} \bullet o_{orderRC}, \langle \$car, * \rangle$	$\rightarrow$	$request(rental\_car1, \$car)$
Action	$\$car \bullet o_{rentalCarOK}$	$\rightarrow$	$responseOk(rental\_car1, \$car)$
Action	$\$car \bullet o_{rentalCarFail}$	$\rightarrow$	$responseFail(rental\_car1, \$car)$
State	$p_{rentalCar1} \bullet o_{orderRC}?$	$\rightarrow$	$accepting\_request(rental\_car1)$

All the following properties hold for the obtained abstraction of the case study.

- The service *GpsSystem* is always *available*.

  $AG\ (accepting\_request(gps))$

- The service *GpsSystem* always replies with successful responses, i.e. it is *reliable*.

  $AG\ [request(gps, \underline{var})]\ AF_{responseOk(gps,var)}\ true$

- The service *Bank* is always *available*.

  $AG\ (accepting\_request(charge))$

- The service *Bank*, after it has accepted a request, always provides a single (either positive or negative) response.

  $AG\ [request(charge, \underline{id})]$
  $AF_{responseOk(charge,id)\ \vee\ responseFail(charge,id)}$
  $\neg\, EF_{responseOk(charge,id)\ \vee\ responseFail(charge,id)}\ true$

- After a successful response to a credit card charge request, the bank accepts undo requests for the successfully completed transaction, i.e. *Bank* is a strong revocable service.

  $AG\ [responseOk(charge, \underline{id})]\ A(accepting\_undo(charge, id)\ {}_{tt}W_{undo(charge,id)}\ true)$

- The service $RentalCar_1$ is always *available*.

  $AG\ (accepting\_request(rental\_car1))$

- The service $RentalCar_1$, once a request is accepted, provides a single (either positive or negative) response.

  $$AG\ [request(rental\_car1, \underline{customer})]$$
  $$AF_{responseOk(rental\_car1,customer)\ \vee\ responseFail(rental\_car1,customer)}$$
  $$\neg\, EF_{responseOk(rental\_car1,customer)\ \vee\ responseFail(rental\_car1,customer)}\ true$$

Orchestration and compensation properties. The properties we have introduced and checked so far imply a sort of *black-box* view of the single services. In fact, their statements are general and given in terms of the externally observable behaviour of services. Of course, whenever details on the internal architecture of a given service, in terms of its subcomponents, are known, i.e. when the service is a sort of *white-box*, further functional properties can be stated in terms of the behaviours of these subcomponents. In general, these properties can express desirable orchestration or compensation behaviours. In the following, we show some examples of formalization of this kind of properties in SocL in the context of our automotive scenario.

The considered abstraction of the case study is obtained by applying the following rules:

$$\begin{array}{lll}
Action & \$car \bullet o_{engineFailure} & \rightarrow request(road_assistance, \$car) \\
Action & \$car \bullet o_{towTruckOK} & \rightarrow responseOk(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarOK} & \rightarrow responseOk(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{notFound} & \rightarrow responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{notFound} & \rightarrow responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{garageFail} & \rightarrow responseFail(road_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarFail} & \rightarrow responseFail(road_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow responseFail(road_assistance, \$car, truckGarage)
\end{array}$$

$$\begin{array}{lll}
Action & p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle & \rightarrow request(charge, \$id) \\
Action & * \bullet o_{chargeOK}, \langle \$id \rangle & \rightarrow responseOk(charge, \$id) \\
Action & * \bullet o_{chargeFail}, \langle \$id \rangle & \rightarrow responseFail(charge, \$id) \\
Action & p_{bank} \bullet o_{revoke}, \langle \$id \rangle & \rightarrow undo(charge, \$id)
\end{array}$$

$$\begin{array}{lll}
Action & \$car \bullet o_{garageOK} & \rightarrow responseOk(garage, \$car) \\
Action & * \bullet o_{cancel}, \langle \$car \rangle & \rightarrow undo(garage, \$car) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow responseFail(towtruck, \$car)
\end{array}$$

The above set of rules is obtained by putting together some of the rules previously introduced with some new rules for capturing interactions with garage, tow truck and rental car services.

Now, we can check the following properties for the automotive scenario.

- After a successful credit card charge, the rental car will be booked, or the garage and tow truck will be ordered, or the credit chard charge will be revoked.

$$AG\,[responseOk(charge, \underline{id})]$$
$$AF_{responseOk(road\_assistance,id,rentalCar)\ \vee\ responseOk(road\_assistance,id,truckGarage)\ \vee\ undo(charge,id)}\ true$$

- It cannot happen that, after the driver's credit card has been charged and some service ordered, the credit card charge is revoked.

$$\neg\, EF_{responseOk(charge,\underline{id})}$$
$$EF_{responseOk(road\_assistance,id,rentalCar)\ \vee\ responseOk(road\_assistance,id,truckGarage)}$$
$$EF_{undo(charge,id)}\ true$$

- It cannot happen that, after the credit card has been charged and then revoked, some order succeeds.

$$\neg\, EF_{responseOk(charge,\underline{id})}$$
$$EF_{undo(charge,id)}$$
$$EF_{responseOk(road\_assistance,id,rentalCar)\ \vee\ responseOk(road\_assistance,id,truckGarage)}\ true$$

- After the garage has been booked, if the tow truck service is not available then the garage is revoked.

$$AG\,[responseOk(garage, \underline{var})]$$
$$AG\,(\,[responseFail(towtruck, var)]\ AF_{undo(garage,var)}\ true\,)$$

4.2.5.2 Finance case study

There are several requirements and properties concerning liveness, correctness, and security that an implementation of the finance case study is expected to fulfill. Among these, in the following we will focus on:

- *Availability:* The credit portal is always capable to accept a credit request.
- *Responsiveness:* Whenever the customer uploads a credit request he always gets an answer, unless he cancels his own request.
- *Correlation soundness:* The customer always receives an answer which is relative to his credit request. Thus, it never occurs that the service sends him an evaluation related to another credit request or that it mixes data related to different credit requests.
- *Interruptibility:* The customer may require the abort of the process after that he has selected the credit request service.

To verify such abstract properties, and other behavioral properties more specific for the case study, we need to apply the following abstraction rules:

$$
\begin{array}{rll}
\mathit{Action} & \mathit{creditRequest}\langle \$1 \rangle & \rightarrow \mathit{request}(cr, \$1) \\
\mathit{Action} & \mathit{offer}\langle \$1, *, * \rangle & \rightarrow \mathit{response}(cr, \$1) \\
\mathit{Action} & \mathit{update}\langle \$1, * \rangle & \rightarrow \mathit{fail}(cr, \$1) \\
\mathit{Action} & \mathit{negativeResp}\langle \$1, * \rangle & \rightarrow \mathit{fail}(cr, \$1) \\
\mathit{Action} & \mathit{cancel}\langle \$1 \rangle & \rightarrow \mathit{cancel}(cr, \$1) \\
\mathit{Action} & \mathit{balanceNotValid}\langle \$1 \rangle & \rightarrow \mathit{fail}(cr, \$1) \\
\mathit{Action} & \mathit{empEvaluation}\langle \$1, *, *, yes \rangle & \rightarrow \mathit{response}(eeval, \$1) \\
\mathit{Action} & \mathit{empEvaluation}\langle \$1, *, *, no \rangle & \rightarrow \mathit{fail}(eeval, \$1) \\
\mathit{Action} & \mathit{supEvaluation}\langle \$1, *, *, yes \rangle & \rightarrow \mathit{response}(seval, \$1) \\
\mathit{Action} & \mathit{supEvaluation}\langle \$1, *, *, no \rangle & \rightarrow \mathit{fail}(seval, \$1) \\
\mathit{Action} & \mathit{validateBalance}\langle \$1, yes \rangle & \rightarrow \mathit{response}(beval, \$1) \\
\mathit{Action} & \mathit{validateBalance}\langle \$1, no \rangle & \rightarrow \mathit{fail}(beval, \$1) \\
\mathit{Action} & \mathit{taskAddedToETL}\langle \$1 \rangle & \rightarrow \mathit{request}(eval, \$1) \\
\mathit{Action} & \mathit{taskAddedToSTL}\langle \$1 \rangle & \rightarrow \mathit{request}(eval, \$1) \\
\mathit{Action} & \mathit{taskAddedToSTL}\langle \$1 \rangle & \rightarrow \mathit{request}(tostl, \$1) \\
\mathit{Action} & \mathit{removeTaskSTL}\langle \$1 \rangle & \rightarrow \mathit{cancel}(eval, \$1) \\
\mathit{Action} & \mathit{removeTaskETL}\langle \$1 \rangle & \rightarrow \mathit{cancel}(eval, \$1) \\
\mathit{Action} & \mathit{reqUpdate}\langle \$1, *, *, *, *, *, * \rangle & \rightarrow \mathit{request}(upd, \$1) \\
\mathit{Action} & \mathit{update}\langle \$1, * \rangle & \rightarrow \mathit{response}(upd, \$1) \\
\mathit{Action} & \mathit{securities}\langle \$1, * \rangle & \rightarrow \mathit{request}(sec, \$1) \\
\mathit{Action} & \mathit{balance}\langle \$1, * \rangle & \rightarrow \mathit{request}(bal, \$1) \\
\mathit{Action} & \mathit{reqProcessing}\langle \$1, *, *, *, * \rangle & \rightarrow \mathit{request}(rproc, \$1) \\
\mathit{State} & \mathit{login} & \rightarrow \mathit{accepting\_request}(login)
\end{array}
$$

We comment on some of the rules, the remaining ones are interpreted similarly. The first rule prescribes that whenever an action over the endpoint *portal•creditRequest*, with any sent data $\langle id \rangle$ matching $\langle \$1 \rangle$, occurs in the label of a transition, then it is replaced by the abstract action $\mathit{request}(cr, id)$. Similarly, the second rule prescribes that whenever an action over the endpoint $\mathit{cust} \bullet \mathit{offer}$, with any sent data $\langle id, \mathit{offer}, \mathit{motivation} \rangle$ matching $\langle \$1, *, * \rangle$, occurs in the label of a transition, then it is replaced by the abstract action $\mathit{response}(cr, id)$. This way, data *offer* and *motivation* are discharged in the 'abstraction process', while session identifier *id* is used to correlate responses from the contacted *Portal* service. To correlate cancellations to the corresponding credit requests, the fifth rule permits replacing actions involving operation *cancel*, with any sent data $\langle id \rangle$ matching $\langle \$1 \rangle$, by abstract action $\mathit{cancel}(cr, id)$. The last rule works similarly, but it applies to labels of states rather than to labels of transitions.

Now, we can check the following properties for the finance case study.

- - - *Availability* - -

  $AG(\mathit{accepting\_request}(login))$

  This formula means that the service *CreditInstitute* is always capable to accept a

credit request. Indeed, atomic proposition $accepting\_request(login)$ means that a state is able to accept a credit request for interaction *login*.

- - - *Responsiveness and correlation soundness* - -

  Both properties are expressed by the following SocL formula:

  $$AG\,[request(cr, \underline{id})]\,AF_{response(cr,id) \vee fail(cr,id) \vee cancel(cr,id)}\,true$$

  This formula means that *CreditInstitute* always guarantees an answer (i.e. an offer or a negative response, sent by means of actions $response(cr, id)$ or $fail(cr, id)$, respectively) to each received credit request, unless the customer cancels his own request (by means of action $cancel(cr, id)$). The answers from *CreditInstitute* and the request of cancellation from *Customer* belong to the same interaction *cr* of the credit request and are properly *correlated* by variable *id*.

- - - *Interruptibility* - -

  $$AG\,[request(cr, \underline{id})]\,EF_{cancel(cr,id)}\,true$$

Other behavioral properties, more specific for this case study, are expressed in SocL as follows.

1. The customer can receive an offer (action $response(cr, id)$) only after its credit request has been successfully evaluated by a supervisor (action $response(seval, id)$).

   $$AG\,[request(cr, \underline{id})]\,\neg E(true_{\neg\, response(seval,id)}\,U_{response(cr,id)}\,true)$$

2. The customer can receive a negative response only after its credit request has been negatively evaluated by an employee or a supervisor (as indicated by the failure of the interactions *eeval* and *seval*, respectively) or the given balance has deemed not to be valid (as indicated by the failure of the interaction *beval*).

   $$AG\,[request(cr, \underline{id})]\,\neg E(true_{\neg\,(fail(eeval,id) \vee fail(seval,id) \vee fail(beval,id))}\,U_{fail(cr,id)}\,true)$$

3. If a credit request is accepted to be evaluated (i.e. it is added to some task list as indicated by the interaction *eval*) and the customer requires the cancellation (action $cancel(cr, id)$), then compensation must be activated, i.e. the task must be removed from the list (action $cancel(eval, id)$).

   $$AG\,[request(eval, \underline{id})]\,EF\,[cancel(cr, id)]AF_{cancel(eval,id)}\,true$$

4. If a credit request is demanded to be updated (interaction *upd*), the customer will be notified or a cancellation will be invoked.

   $$AG\,[request(upd, \underline{id})]\,AF_{response(upd,id) \vee cancel(cr,id)}\,true$$

5. Before processing a credit request (interaction *rproc*), the customer must insert securities (interaction *sec*) and balance data (interaction *bal*).

   $AG\,[request(cr, \underline{id})]\,\neg E(true_{\neg\,(request(sec,id) \vee request(bal,id))} U_{request(rproc,id)} true)$

6. A credit request can always succeed.

   $AG\,[request(cr, \underline{id})]\,AF_{\neg cancel(cr,id) \vee response(cr,id)}\,true$

7. A supervisor can always be involved for evaluating a credit request (interaction *tostl* starts when a request is added to the supervisor tasks list).

   $AG\,[request(cr, \underline{id})]\,AF_{\neg cancel(cr,id) \vee request(tostl,id)}\,true$

As expected, all the abstract properties we presented before do hold for the COWS specification of the finance case study, except for the last two properties, because, e.g., a supervisor can evaluate a credit request negatively. In case a property does not hold, CMC produces a clear and detailed explanation of the returned results, i.e. a so called *counterexample*. For example, Figure 4.8 shows an excerpt of the output returned by CMC when checking the second-last property.

4.2.6 Further Issues

We linger here on some side issues related to our methodology which are nevertheless useful to understand the overall schema into which our research and experimentation has to be considered. Several other research directions remain open, and we plan to further investigate them even if not necessarily in the short term.

4.2.6.1 Possible extensions of SocL for supporting multiple substitutions

In the presentation of SocL in Section 4.2.2, when introducing the *action formulae semantics*, we have defined that a set of abstract labels A satisfies an action $\underline{a}$ of the logic (potentially containing variable binders) if and only if exactly one of the labels of A matches the action $\underline{a}$, thus identifying a unique substitution ρ for the variables of the action.

This simplification relies on the assumption that inside a single transition of an L^2TS two or more actions with the same type and interaction name never occur. Now, this assumption is reasonable for the COWS computational model because of the interleaving semantics of the language. In a more general setting, e.g. in the case of languages with a truly parallel interpretation model (like UML statecharts parallel states) or with composite sequential actions (like UML statechart actions), the assumption and, hence, the simplification, would not be feasible any longer.

```
The Formula: "AG [request(cr,$id)] AF {not cancel(cr,%id) or response(cr,%id)}true"
is: FALSE
(states generated= 255, computations fragments generated= 403)
cmc> ------------------------------------------
The formula:  AG  [ request(cr,$id) ] AF {not cancel(cr,%id) or response(cr,%id)} true
is FOUND_FALSE in State C1
...
C9  -->  C10 { portal.creditRequest!<sessionID#1#>,
               portal.creditRequest?<sessionID#1#> }
             {{  {{ request(cr,sessionID#1#) }} }}
and the formula:  AF {not cancel(cr,sessionID#1#) or response(cr,sessionID#1#)} true
is FOUND_FALSE in State C10
because
C10  -->  C12 { portal.getCreditRequest!<sessionID#1#,data,15000,customer>,
               portal.getCreditRequest?<sessionID#1#,CUST_DATA,AMOUNT,CUST> }
               {{  {{  }} }}
C12  -->  C14 { portal.securities!<sessionID#1#,secValues>,
               portal.securities?<sessionID#1#,SEC_DATA> }
               {{  {{ request(sec,sessionID#1#) }} }}
C14  -->  C17 { portal.balance!<sessionID#1#,balance>,
               portal.balance?<sessionID#1#,BALANCE> }
               {{  {{ request(bal,sessionID#1#) }} }}
C17  -->  C23 { validation.validateBalance!<sessionID#1#,portal,balance>,
               validation.validateBalance?<ID,BANK,BALANCE> }
               {{  {{  }} }}
C23  -->  C28 { portal.validateBalance!<sessionID#1#,yes>,
               portal.validateBalance?<sessionID#1#,yes> }
               {{  {{ response(beval,sessionID#1#) }} }}
C31  -->  C112 { portal.reqProcessing#1#!<sessionID#1#,data,secValues,balance,15000,customer>,
               portal.reqProcessing#1#?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST> }
               {{  {{  }} }}
C112  -->  C244 { portal.empEvaluation!<sessionID#1#,rating,additionalInfo,no>,
                  portal.empEvaluation?<sessionID#1#,RATING,ADDITIONAL_INFO,DECISION> }
                  {{  {{ fail(eeval,sessionID#1#) }} }}
C249  -->  C251 { customer.negativeResp!<sessionID#1#,additionalInfo>,
                  customer.negativeResp?<sessionID#1#,MOTIVATIONS> }
                  {{  {{ fail(cr,sessionID#1#) }} }}
------------------------------------------
```

Table 4.8: CMC: a counterexample

In fact, we can generalize our methodology by relaxing this simplification. This would require a slightly more complex definition of the semantics of SocL, since we would have to consider the possibility of applying multiple substitutions to formulae in parallel. Formally, the satisfaction relation $\models$ for action formulae would be defined over a set A of closed actions and a *set* P *of substitutions*:

- $\mathrm{A} \models \underline{a} \rhd \mathrm{P}$ iff $\mathrm{P} = \{\, \rho \mid \mathsf{match}(\underline{a}, a') = \rho\,,\ a' \in \mathrm{A} \,\}$ is not empty;
- $\mathrm{A} \models \chi \rhd \{\emptyset\}$ iff $\mathrm{A} \models \chi$.

where the relation $\mathrm{A} \models \chi$ is defined in Section 4.2.2.1 (Definition 4.2.5). Hence, the satisfaction relation for SocL formulae must be modified accordingly; we report here the definition of $\models$ for the next operator (the cases for the until operators are similar):

- $q \models EX_\gamma \phi$ iff $\exists \sigma \in \mathit{path}(q) :\ \sigma\{1\} \models \gamma \rhd \mathrm{P}$, and $\exists \rho \in \mathrm{P} :\ \sigma(1) \models \phi\rho$;
- $q \models AX_\gamma \phi$ iff $\forall \sigma \in \mathit{path}(q) :\ \sigma\{1\} \models \gamma \rhd \mathrm{P}$, and $\forall \rho \in \mathrm{P} :\ \sigma(1) \models \phi\rho$.

Of course, the above semantics leads to a model checking algorithm with higher complexity compared to that presented in Section 4.2.4.

4.2.6.2 Towards a real-life formal verification environment

The development of CMC is still in progress and the current prototypical versions of the tool are mainly being used at ISTI-CNR for academic and experimental purposes. So far the focus of the development has been on the design of qualitative features one would desire for a verification tool, thus experimenting with various logics, system modelling languages and user interfaces. For the moment we do not plan to transform the prototypes into real-life development tools. Such an evolution would require addressing issues like strong code optimizations, scalability over massively large systems, exhaustive testing and validation issues.

For what concerns the adoption of the overall methodology into an effective software engineering practice, we see two main difficulties:

- generating COWS specifications from within an industrial context
- correctly identifying and encoding the full set of properties the model should satisfy.

For the first aspect, a reasonable approach might rely on some kind of mechanical translation of higher-level description languages, like WS-BPEL, already accepted and adopted in industrial contexts, into a computational model supported by CMC, like e.g. COWS (see Section 3.3 for a preliminary work in this direction).

For the second aspect, the solution necessarily passes through a formalization of the requirement collection phase, allowing for the collection of the properties to be verified on the system. A mechanical/interactive translation of the initial requirements (e.g. from UML sequence diagrams, structured informal natural language, or other logics) into SocL formulae is definitely a goal which should be pursued and which does not seem to be too far from the current state of art.

4.2.7 Concluding remarks

We have tackled the problem of analysing the functional behaviour of services and SOC systems. To this aim, we have defined a state- and action-based branching time temporal logic, SocL, capable of representing distinctive aspects of services and have used it to express a set of desirable functional properties of services. With respect to previous state- and action-based logics SocL adds the possibility that formulae can be parameterized by data values. In this sense, SocL resembles other logics stating properties of message-passing systems, such as that introduced in [106], in which a formula referring input and output actions acts as a binder for those variables that occur free in expressions within the sequel of the formula. Parametric formulae are also considered in the modal logic for mobile agents introduced in [79], although limited to variables ranging over *localities*,

and in its extension MOSL [78] with more general message-passing capabilities, including communication of data, locations and processes.

In SocL, parameterization with data values concerns correlation data, which is exploited to effectively express properties of service-oriented systems. In fact, being inherently loosely coupled, SOC systems do not provide intrinsic mechanisms to enable linking together actions executed as part of the same interaction. Emerging standards like WS-BPEL and WS-CDL have hence put forward the idea that messages exchanged by actions as part of the same interaction must contain some common data that the interacting partners can identify. This simple concept has in practice many instantiations. For example, in WS-Addressing [103] correlation data are implicitly dealt with (by the underlying communication protocols) thus resulting in a less flexible mechanism with respect to that provided by WS-BPEL, where instead correlation data must be explicitly dealt with by the developer and included among the data used for invoking services' operations. These different levels of abstraction are somehow reflected by the process calculi for SOC proposed so far, which may be roughly classified in session-based, like those in [57, 124, 35, 45, 199], and correlation-based, like those in [104, 50, 131], respectively.

We have also developed an efficient on-the-fly model checker for the SocL logic that could be very fruitful to analyse functional behaviours of SOC applications at modelling stage. By means of the two case studies introduced in Section 2.4 we have illustrated an application of our methodology: first, specify the service-oriented application to be analysed by means of a formal language like, e.g., COWS; then, by also taking into account the properties to be checked, transform this concrete specification into a more abstract one in terms of L^2TSs; finally, use the model checker to perform the verification. In fact, this approach shows a novel use of temporal logics and model checkers. The properties to be checked are expressed in the logic SocL in a way that is independent from the model of the system under analysis; then, through an abstraction process, the model is tailored to be checked against the properties of interest. Although SocL can also be used in a more standard way to directly express the expected properties of a given COWS specification in terms of the concrete actions occurring therein, the possibility to express service properties in a model independent way is a first important advantage of our approach.

Another advantage is that, since the logic interpretation domain (i.e. L^2TSs) is independent from the service specification language (i.e. COWS), the approach can be tailored to be used in conjunction with other SOC specification languages. To this aim, one has to define first an L^2TS-based operational semantics for the language of interest and then a suitable set of abstraction rules mapping the concrete actions of the language into the abstract actions of SocL. In fact, a first attempt along this direction has been done in [191] for accommodating an UML based computational model within the approach described here.

Furthermore, SocL permits expressing properties about any kind of interaction patterns, such as *one–way*, *request–response*, *one request–multiple responses*, *one request–one of two possible responses*, etc. In fact, properties of complex interaction patterns can

be expressed by correlating SocL observable actions using interaction names and correlation values. A similar approach to express properties of the interaction protocols that are responsible for interconnecting the different parties involved in a composite service architecture has been recently proposed in [2]. The authors employ SRML [90] to specify service architectures, and a logic adapted from UCTL [190] to express the properties of the interaction protocols. However, the logic they propose does not allow correlation to be taken into account and, hence, such general properties as those shown in Section 4.2.1 cannot be expressed.

We leave for future work the extension of our environment to support a more compositional verification methodology. In fact, systems of services can currently be analysed only 'as a whole', while it is not possible to analyse isolated services (e.g. a provider service without a client). This is somewhat related to the original semantics of COWS that follows a 'reduction' style; in Section 4.4 we present an alternative operational semantics (see also [175]), based on a 'symbolic' approach, that should permit to overcome this limitation.

4.3 A bisimulation-based observational semantics

In Chapter 3, we have shown through many examples that COWS's priority mechanisms, which allow some actions to take precedence over others, can be very fruitful to model SOC scenarios. For example, when a message arrives, the problem arises of rightly handling race conditions among those service instances and the corresponding service definition which are able to receive the message. By assigning to receive activities priority values which depend on the messages available, and by exploiting a parallel composition operator that gives precedence to actions with greater priority, service instances take precedence over the corresponding service definition when both can process the same message, thus preventing creation of wrong new instances (see Section 3.2.2.2).

Notably, receives would have *dynamically* assigned priority values since these values depend on the matching ability of their argument pattern. Indeed, while computation proceeds, some of the variables used in the argument pattern of a receive can be assigned values, because of execution of syntactically preceding receives or of concurrent threads sharing these variables. This, on the one hand, restricts the set of messages matching the pattern but, on the other hand, increases the priority of the receive in comparison with other receives matching the same message. Furthermore, priority is *local* since receives having a more defined pattern have a higher execution priority with respect to only the other receives matching the same message. In fact, e.g. receives non-matching the message and invoke activities can be executed in any order.

Moreover, in Section 3.2.3.3, we have shown other situations where local pre-emption is needed. For example, when a fault arises in a scope, (some of) the remaining activities of the enclosing scope should be terminated before starting the execution of the relative fault handler. This is modelled in COWS by exploiting the same parallel operator as

before together with actions for forcing immediate termination of concurrent activities which take the greatest priority. The same mechanism, of course, can also be used for exception and compensation handling.

However, to the best of our knowledge, such priority mechanisms that combine dynamic priority with local pre-emption, have not yet been investigated in the literature [69], apart from COWS. Therefore, we study the impact of COWS's priority mechanisms on observational semantics for SOC. The obtained semantic theories are directly usable to check interchangeability of services and conformance against service specifications. They can also be used to reduce the size of the model representing services, thus e.g. facilitating model checking of service properties (see the verification methodology presented in Section 4.2).

Since we want to investigate how COWS distinctive features affect well-established semantic theories, we present the observational semantics of COWS in three steps, as we have done in Chapter 3 for its operational semantics.

1. In Section 4.3.1, to understand the effect of non-binding and localised receives, pattern-matching, and global scope on the semantics, we first define strong and weak open barbed bisimilarities [181, 113] for μCOWS^m. We then provide more manageable labelled bisimilarities and prove that they are *sound* and *complete* with respect to barbed (contextual) ones. Both semantics recall those for asynchronous π-calculus of [7]. A major complication is that, due to the locality of received endpoints (i.e. only the output capability of names may be transmitted, as in the *localised π-calculus* [147]), the definition of labelled bisimilarities involves a family of relations indexed by sets of names. This is somewhat similar to the definition of quasi-open bisimilarity for π-calculus [181].

2. In Section 4.3.2, to understand the effect of actions with dynamic changing priority on the semantics, we move on μCOWS. Again, we provide sound and complete characterisations of the barbed bisimilarities in terms of corresponding labelled bisimilarities. The obtained semantics inhabits between asynchrony and synchrony because, with respect to a purely asynchronous setting, the priority mechanism permits partially recovering the capability to observe receive actions.

3. In Section 4.3.3, we extend our investigation to COWS. The primitives with greatest priority causing termination require specific conditions on the labelled bisimilarities for these to be congruences. The resulting observations are hence more fine-grained than the previous ones. The results of coincidence still hold.

4.3.1 Observational semantics of μCOWS^m

Before defining an observational theory for μCOWS^m, whose syntax and operational semantics are presented in Section 3.2.1, we have to slightly adapt its operational semantics to this purpose. More specifically, to properly define the labelled transition relation $\xrightarrow{\alpha}$,

$$\frac{s \xrightarrow{\mathtt{n} \lhd [\bar{m}]\,\bar{v}} s' \quad n \in \bar{v} \quad n \notin (\mathtt{n} \cup \bar{m})}{[n]\,s \xrightarrow{\mathtt{n} \lhd [n,\bar{m}]\,\bar{v}} s'}\ (open_{inv}) \qquad \frac{s \xrightarrow{\mathtt{n} \rhd [\bar{y}]\,\bar{w}} s' \quad x \in \bar{w} \quad x \notin \bar{y}}{[x]\,s \xrightarrow{\mathtt{n} \rhd [x,\bar{y}]\,\bar{w}} s'}\ (open_{rec})$$

Table 4.9: μCOWS^m operational semantics (additional rules)

we have to add the two operational rules shown in Table 4.9 to those of Table 3.4. Label α is now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd [\bar{n}]\,\bar{v} \quad | \quad \mathtt{n} \rhd [\bar{x}]\,\bar{w} \quad | \quad \sigma$$

The meaning of new labels is as follows: $\mathtt{n} \lhd [\bar{n}]\,\bar{v}$ denotes bound invocations, which transmit private names $\bar{n}$ and can be generated by rule *(open$_{inv}$)*, while $\mathtt{n} \rhd [\bar{x}]\,\bar{w}$ denotes delimited receive activities, with delimited arguments $\bar{x}$, which can proceed thanks to rule *(open$_{rec}$)*. As usual, σ denotes execution of a communication and $\emptyset$ denotes a computational step. In the sequel, we will write $\mathtt{n} \lhd \bar{v}$ (resp. $\mathtt{n} \rhd \bar{w}$) instead of $\mathtt{n} \lhd [\,]\,\bar{v}$ (resp. $\mathtt{n} \rhd [\,]\,\bar{w}$) and use bu($\alpha$) to denote the set of names/variables that occur bound in α, i.e. bu($\mathtt{n} \lhd [\bar{n}]\,\bar{v}$) = $\bar{n}$, bu($\mathtt{n} \rhd [\bar{x}]\,\bar{w}$) = $\bar{x}$ and bu(σ) = $\emptyset$. As before, u(α) denotes the set of names and variables occurring in α.

Notably, bound invocation actions do not appear in rule *(com)* (Table 3.4), and therefore cannot directly interact with receive actions, and similarly delimited receive actions cannot synchronise with invoke actions. Such interactions are instead inferred by using structural congruence to pull name/variable delimitations outside of both interacting activities. Since rules *(open$_{inv}$)* and *(open$_{rec}$)* have no impact on inferring computational steps, they have been omitted from the rules defining the operational semantics of μCOWS^m in Section 3.2.1.2. However, when it comes to developing behavioural equivalences, such rules turn out to be indispensable. It is also worth noticing that when rule *(open$_{rec}$)* is applied to a closed μCOWS^m service, the resulting term could be open.

A couple of properties of the operational semantics that will be exploited in the rest of the section follow.

Property 4.3.1 *Let s be a μCOWS^m term. The following facts hold:*

- *If $s \xrightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s'$, then $\bar{n} \subseteq \bar{v}$ and $\mathtt{n} \notin \bar{n}$.*
- *If $s \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} s'$, then $\bar{x} \subseteq \bar{w}$. Moreover, if s is closed then $\bar{w} \backslash \bar{x}$ does not contain variables (i.e. it is either a tuple of values or the empty tuple).*

The above properties can be proved by a straightforward induction on the depth of the shortest inference for the transitions in the hypothesis.

We introduce now an observational semantics for μCOWS^m. Specifically, we define natural notions of strong and weak open barbed bisimilarities and prove their coincidence with more manageable characterisations in terms of labelled bisimilarities.

4.3.1.1 Strong open barbed bisimilarity

We want to define a notion of *(open) barbed bisimilarity* for the calculus along the line of [113, 181]. To this aim, we must first identify an appropriate *basic observable*, namely a predicate that points out the interaction capabilities of a term. Since communication is asynchronous, an obvious starting point is considering as observable only the output capabilities of terms, like for asynchronous π-calculus [7]. The intuition is that an asynchronous observer cannot directly observe the receipt of data that he has sent. Thus, our notion of observation is as follows.

Definition 4.3.1 (Observable for μCOWS^m) *Let s be a μCOWS^m closed term. Predicate $s \downarrow_{\mathtt{n}}$ holds true if s can immediately perform an invoke over the (public) endpoint $\mathtt{n}$, that is if there exist s', $\bar{n}$ and $\bar{v}$ such that $s \xrightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s'$.*

A desirable property of a behavioural equivalence, that enables compositional reasoning, is to be preserved by all contexts of the language. An equivalence that enjoys this property is called *congruence*. A μCOWS^m (closed) context is a service $\mathbb{C}$ with a 'hole' $[\![\cdot]\!]$, i.e. a term generated by the following grammar:

$$\mathbb{C} ::= [\![\cdot]\!] \mid \mathbb{G} \mid \mathbb{C} \,|\, s \mid s \,|\, \mathbb{C} \mid [u]\,\mathbb{C} \mid * \,\mathbb{C} \qquad \mathbb{G} ::= \mathtt{n}?\bar{w}.\,\mathbb{C} \mid \mathbb{G} + g \mid g + \mathbb{G}$$

such that, once the hole is filled with a closed service s, the resulting term $\mathbb{C}[\![s]\!]$ is a μCOWS^m closed service.

The above definitions of observation and context lead to the following notion of barbed bisimilarity.

Definition 4.3.2 (Open barbed bisimilarity) *A symmetric binary relation $\mathcal{R}$ on μCOWS^m closed terms is an* open barbed bisimulation *if whenever $s_1 \mathcal{R} s_2$ the following holds:*

(Barb preservation) *if $s_1 \downarrow_{\mathtt{n}}$ then $s_2 \downarrow_{\mathtt{n}}$;*

(Computation closure) *if $s_1 \xrightarrow{\emptyset} s_1'$ then there exists s_2' such that $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \mathcal{R} s_2'$;*

(Context closure) $\mathbb{C}[\![s_1]\!]\,\mathcal{R}\,\mathbb{C}[\![s_2]\!]$, *for every closed context $\mathbb{C}$.*

Two closed terms s_1 and s_2 are open barbed bisimilar*, written $s_1 \simeq_m s_2$, if $s_1 \mathcal{R} s_2$ for some open barbed bisimulation $\mathcal{R}$. $\simeq_m$ is called open barbed bisimilarity.*

Of course, because of context closure, $\simeq_m$ is a congruence for μCOWS^m closed terms.

4.3.1.2 Strong labelled bisimilarity

The definition of $\simeq_m$ suffers from universal quantification over all possible language contexts, which makes the reasoning on terms very hard. Hence, we provide a purely coinductive notion of bisimulation that only requires considering transitions of the labelled

transition system defining the semantics of the terms under analysis. The notion of labelled bisimulation introduced for asynchronous π-calculus in [7] does not turn out to be suitable for μCOWS^m, since the bisimilarity defined on top of it would not properly characterise $\simeq_m$. Consider, for example, the following two μCOWS^m terms:

$$s_1 \triangleq [\mathtt{n}](\mathtt{m}!\langle \mathtt{n}\rangle \mid \mathtt{n}!\langle\rangle) \qquad\qquad s_2 \triangleq [\mathtt{n}]\,\mathtt{m}!\langle \mathtt{n}\rangle$$

They only differ for the fact that the first one is able to perform an invocation along the private endpoint $\mathtt{n}$. However, they can exhibit the same barbs, and no context can tell them apart since it cannot be able to perform a receive along (the received name) $\mathtt{n}$ because of the constraint on the 'localisation' of names (indeed, contexts of the form $[\![\cdot]\!] \mid \mathtt{m}?\langle x\rangle.\, x?\langle\rangle.\, \mathbf{0}$ are not allowed). Hence, s_1 and s_2 are barbed bisimilar. The natural asynchronous labelled bisimilarity derived from the π-calculus one would instead tell them apart and, hence, need to be weakened. Therefore we define a labelled bisimulation as a family of relations indexed with sets of names corresponding to the names that cannot be used by contexts (to test) for reception since they are dynamically exported private names.

Definition 4.3.3 (Names-indexed family of relations) *A names-indexed family $\mathcal{F}$ of relations is a set of symmetric binary relations $\mathcal{R}_N$ on μCOWS^m closed terms, one for each set of names N, i.e. $\mathcal{F} = \{\mathcal{R}_N\}_N$.*

Definition 4.3.4 (Labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_N\}_N$ is a* labelled bisimulation *if, whenever $s_1 \mathcal{R}_N s_2$ and $s_1 \xrightarrow{\alpha} s_1'$, where* bu$(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*
   - *(a)* $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} s_2'$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma \,:\, s_1' \cdot \sigma \,\mathcal{R}_N\, s_2' \cdot \sigma$
   - *(b)* $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma \,:\, s_1' \cdot \sigma \,\mathcal{R}_N\, (s_2' \mid \mathtt{n}!(\bar{w} \cdot \sigma))$
2. *if $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$ where $\mathtt{n} \notin N$ then $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s_2'$ and $s_1' \,\mathcal{R}_{N \cup \bar{n}}\, s_2'$*
3. *if $\alpha = \emptyset$ then $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \,\mathcal{R}_N\, s_2'$*

*Two closed terms s_1 and s_2 are N-*bisimilar*, written $s_1 \sim_m^N s_2$, if $s_1 \mathcal{R}_N s_2$ for some $\mathcal{R}_N$ in a labelled bisimulation. They are* labelled bisimilar*, written $s_1 \sim_m s_2$, if they are $\emptyset$-bisimilar. $\sim_m^N$ is called N-bisimilarity, while $\sim_m$ is called labelled bisimilarity.*

The resulting definition somewhat recalls that of quasi-open bisimilarity for π-calculus [181]. Clause 1 states that a receive action can be simulated either in a normal way (clause 1.(a)) or by a computational step leading to a term that, when composed with the invoke activity consumed by the receive, stands in the appropriate relation (clause 1.(b)).

Execution of receives whose argument contains variables leads to open terms, which the operational semantics is not defined for. Since the freed variables are placeholders for values to be received, we require the two continuations to be related for any matching tuple of values (similarly to late bisimulation for π-calculus [150]). We say that the bisimulation is given in a *late* style because in clause 1 the choice of the tuple of values $\bar{v}$ takes place after the choice of s_2'; that is, s_2' does not depend on the tuple of values $\bar{v}$. Clause 2, and the use of names-indexed families of relations, handles the fact that dynamically exported private names cannot be used by a receiver within the endpoint of a receive (whose syntax, as we have seen in Section 3.2.1.1, does not allow to use variables). With abuse of notation, $\mathtt{n} \notin \mathcal{N}$ in clause 2, with $\mathtt{n} = p \bullet o$, stands for $p \notin \mathcal{N} \wedge o \notin \mathcal{N}$. Thus, invocations along endpoints using either of the names in $\mathcal{N}$ are unobservable, hence these endpoints cannot be used to tell the executing terms apart. Finally, clause 3 deals with computational steps. Notably, actions σ different from $\emptyset$ are not taken into account, since they cannot be performed by closed terms (see rules *(com)* and *(del_{com})*).

To illustrate our labelled bisimilarity, let us consider a tailored version of the input absorption law characterizing asynchronous bisimulation in asynchronous π-calculus (i.e. the equation $a(b).\,\bar{a}b + \tau = \tau$ presented in [7]):

$$[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \;\sim_m\; \emptyset \tag{4.3}$$

where, for the sake of presentation, we exploit the context $\emptyset + [\![\cdot]\!] \triangleq [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle + [\![\cdot]\!])$ and the term $\emptyset \triangleq [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle)$. Communication along the private endpoint $\mathtt{m}$ models the τ action of π-calculus, while activities $\mathtt{n}?\langle x, v\rangle$ and $\mathtt{n}!\langle x, v\rangle$ recall the π-calculus actions $a(b)$ and $\bar{a}b$, respectively. Intuitively, the equality means that a service that emits the data it has received behaves as a service that simply performs an unobservable action. Although the two terms in (4.3) are syntactically different, it turns out that they are bisimilar. Indeed, the only transition that the term $\emptyset$ can perform is:

$$\emptyset \xrightarrow{\emptyset} \mathbf{0}$$

Trivially, the other term can reply by executing the activity on the left-hand side of +:

$$[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \xrightarrow{\emptyset} \mathbf{0}$$

Moreover, the term on the left in (4.3) can also perform the following transition:

$$[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \xrightarrow{\mathtt{n} \rhd [x]\,\langle x,v\rangle} [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid \mathtt{n}!\langle x, v\rangle)$$

To this, $\emptyset$ can reply with an $\emptyset$-transition and, then, for all v', $[\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid \mathtt{n}!\langle v', v\rangle)$ and $(\mathbf{0} \mid \mathtt{n}!\langle v', v\rangle)$ are bisimilar. Indeed, both of them can only perform a transition labelled by $\mathtt{n} \lhd \langle v', v\rangle$ and evolve to $[\mathtt{m}]\,\mathtt{m}!\langle\rangle$ and $\mathbf{0}$, respectively, that cannot perform any further transition.

As another example regarding labelled bisimilarity, we can now prove that

$$[\mathtt{n}]\,(\,\mathtt{m}!\langle\mathtt{n}\rangle \mid \mathtt{n}!\langle\rangle\,) \sim_m [\mathtt{n}]\,\mathtt{m}!\langle\mathtt{n}\rangle$$

In fact, the family of relations $\{\mathcal{R}_\emptyset, \mathcal{R}_{\{\mathtt{n}\}}\}$, where $\mathcal{R}_\emptyset = \{\,(\,[\mathtt{n}]\,(\,\mathtt{m}!\langle\mathtt{n}\rangle \mid \mathtt{n}!\langle\rangle\,),\, [\mathtt{n}]\,\mathtt{m}!\langle\mathtt{n}\rangle\,)\,\}$ and $\mathcal{R}_{\{\mathtt{n}\}} = \{\,(\,\mathtt{n}!\langle\rangle\,,\,\mathbf{0}\,)\,\}$, is a labelled bisimulation.

We want now to prove that labelled bisimilarity is a congruence for μCOWS^m. To this aim we also need to consider *open* terms, i.e. terms with free variables, although we have only defined (strong) labelled bisimulation and bisimilarity over closed terms. We proceed as in [148, Section 4.4], where a similar situation is faced in the setting of CCS. Therefore, we extend the definition of $\sim_m^{\mathcal{N}}$ as follows.

Definition 4.3.5 *Let s_1 and s_2 be two μCOWS^m terms containing free variables $\bar{x}$ at most. Then $s_1 \sim_m^{\mathcal{N}} s_2$ if, for all values $\bar{v}$ such that $|\bar{x}|=|\bar{v}|$, $s_1\cdot\{\bar{x}\mapsto\bar{v}\} \sim_m^{\mathcal{N}} s_2\cdot\{\bar{x}\mapsto\bar{v}\}$.*

In other words, $\sim_m^{\mathcal{N}}$ is generalised to open terms as an hyperequivalence.

To prove the congruence result, we introduce also the notion of *bisimulation up-to structural congruence*: it is defined as a labelled bisimulation except for the fact that the $\mathcal{R}_{\mathcal{N}}$ in the three clauses of Definition 4.3.4 is replaced by the (compound) relation $\equiv \mathcal{R}_{\mathcal{N}} \equiv$. The next lemma shows that a bisimulation up-to $\equiv$ can be used as a sound proof-technique for labelled bisimulation. In the sequel, for the sake of simplicity, we explicitly write index $\mathcal{N}$ in a relation $\mathcal{R}_{\mathcal{N}}$ only when it is necessary or is modified in the considered case. Moreover, for the sake of readability, we only outline here the techniques used in the proofs and refer the interested reader to Appendix B.2 for a full account.

Lemma 4.3.1 *Let $\mathcal{F}$ be a labelled bisimulation up-to $\equiv$; then, $\mathcal{F}$ is a labelled bisimulation.*

Proof. We need to prove that the family of relations

$$\{\,\{\,(s_1, s_2) \;:\; s_1 \equiv s_1',\ s_2' \equiv s_2,\ s_1'\mathcal{R}s_2'\,\} \;:\; \mathcal{R}\in\mathcal{F}\,\}$$

is a labelled bisimulation. The key point of the proof is that if $s_1 \xrightarrow{\alpha} s'$ then, by rule *(cong)* and since $s_1 \equiv s_1'$, we get that $s_1' \xrightarrow{\alpha} s'$. The proof then proceeds by case analysis on α, exploiting the fact that $s_1'\mathcal{R}s_2'$. □

Theorem 4.3.1 *$\sim_m$ is a congruence for μCOWS^m closed terms.*

Proof (sketch). We shall prove that, given two μCOWS^m closed terms s_1 and s_2, if $s_1 \sim_m s_2$ then $\mathbb{C}[\![s_1]\!] \sim_m \mathbb{C}[\![s_2]\!]$ for every (possibly open) context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$. □

It is worth noticing that Theorem 4.3.1 does not hold in case of open terms. For example, consider the following equation:

$$\emptyset + \mathtt{n}?\langle x\rangle.\,\mathtt{n}!\langle x\rangle \;\sim_m\; [\mathtt{m}]\,(\mathtt{m}!\langle x\rangle \mid \mathtt{m}?\langle x\rangle)$$

Although the two terms above are open, we can easily prove that they are bisimilar. Indeed, we can prove that $\emptyset + \mathtt{n}?\langle v\rangle.\,\mathtt{n}!\langle v\rangle \;\sim_m\; [\mathtt{m}]\,(\mathtt{m}!\langle v\rangle \mid \mathtt{m}?\langle v\rangle)$ holds for all v (we can proceed as for (4.3)). However, the context $[x]\,[\![\cdot]\!]$ can tell the two terms apart, since $[x]\,(\emptyset + \mathtt{n}?\langle x\rangle.\,\mathtt{n}!\langle x\rangle)$ can perform action $\mathtt{n} \rhd [x]\,\langle x\rangle$, while $[x,\mathtt{m}]\,(\mathtt{m}!\langle x\rangle \mid \mathtt{m}?\langle x\rangle)$ cannot perform any transition.

Now, we prove that open barbed bisimilarity and labelled bisimilarity coincide. In other words, labelled bisimilarity is *sound* and *complete* with respect to the barbed (contextual) one.

Theorem 4.3.2 (Soundness of $\sim_m$ w.r.t. $\simeq_m$) *Given two μCOWS^m closed terms s_1 and s_2, if $s_1 \sim_m s_2$ then $s_1 \simeq_m s_2$.*

Proof (sketch). By Theorem 4.3.1, we have that $\sim_m$ is context closed. Thus, we only need to prove that $\sim_m$ is barb preserving and computation closed. □

Theorem 4.3.3 (Completeness of $\sim_m$ w.r.t. $\simeq_m$) *Given two μCOWS^m closed terms s_1 and s_2, if $s_1 \simeq_m s_2$ then $s_1 \sim_m s_2$.*

Proof (sketch). We define a family of relations $\mathcal{F} = \{\mathcal{R}_N : N \text{ set of names}\}$ such that $\simeq_m$ is included in $\mathcal{R}_\emptyset$ and show that it is a labelled bisimulation. Let N be the set $\{n_1,\ldots,n_m\}$, then $s_1 \mathcal{R}_N s_2$ if there exist $\mathtt{m}_1,\ldots,\mathtt{m}_m$ fresh such that

$$[n_1,\ldots,n_m]\,(s_1 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle) \simeq_m [n_1,\ldots,n_m]\,(s_2 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle)$$

Take $s_1 \mathcal{R}_N s_2$ and a transition $s_1 \xrightarrow{\alpha} s_1'$; then, the proof proceeds by case analysis on α. □

Corollary 4.3.1 *$\sim_m$ and $\simeq_m$ coincide.*

Proof. Directly from Theorems 4.3.2 and 4.3.3. □

4.3.1.3 Weak open barbed and labelled bisimilarities

Our semantic theories extend in a standard way to the weak case. Therefore, weak transitions are defined as follows: $\Longrightarrow$ means $(\xrightarrow{\emptyset})^*$, i.e. zero or more $\emptyset$-transitions; $\overset{\alpha}{\Longrightarrow}$ means $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$; $\overset{\hat{\alpha}}{\Longrightarrow}$ means $\overset{\alpha}{\Longrightarrow}$ if $\alpha \neq \emptyset$ and $\Longrightarrow$ if $\alpha = \emptyset$. Predicate $s \Downarrow_{\mathtt{n}}$ holds true if there exist s' such that $s \Longrightarrow s'$ and $s' \downarrow_{\mathtt{n}}$.

Then, *weak open barbed bisimilarity*, written $\approxeq_m$, is obtained by replacing $s \downarrow_{\mathtt{n}}$ with $s \Downarrow_{\mathtt{n}}$, and $\xrightarrow{\emptyset}$ with $\Longrightarrow$ in Definition 4.3.2.

To define weak labelled bisimilarity we replace $\xrightarrow{\alpha}$ with $\overset{\hat{\alpha}}{\Longrightarrow}$ in the three clauses of Definition 4.3.4. As for π-calculus, the only subtle point is the case of receive actions,

which have to be simulated by $\Longrightarrow \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} \Longrightarrow$ and requires the substitution to be applied immediately after the transition $\xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}}$, i.e. before further $\emptyset$-transitions (otherwise, just like in π-calculus, the resulting relation would not be an equivalence).

Definition 4.3.6 (Weak labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_\mathcal{N}\}_\mathcal{N}$ is a* weak labelled bisimulation *if, whenever $s_1 \mathcal{R}_\mathcal{N} s_2$ and $s_1 \xrightarrow{\alpha} s_1'$, where* $\mathrm{bu}(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*

   (a) $\exists\, s_2'' \;:\; s_2 \Longrightarrow \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} s_2''$ *and*
   $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma\ \exists\, s_2' \;:\; s_2'' \cdot \sigma \Longrightarrow s_2'$ *and* $s_1' \cdot \sigma\, \mathcal{R}_\mathcal{N}\, s_2'$

   (b) $\exists\, s_2' \;:\; s_2 \Longrightarrow s_2'$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma \;:\; s_1' \cdot \sigma\, \mathcal{R}_\mathcal{N}\, (s_2' \mid \mathtt{n}!(\bar{w} \cdot \sigma))$

2. *if $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$ where $\mathtt{n} \notin \mathcal{N}$ then $\exists\, s_2' \;:\; s_2 \xRightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s_2'$ and $s_1'\, \mathcal{R}_{\mathcal{N} \cup \bar{n}}\, s_2'$*

3. *if $\alpha = \emptyset$ then $\exists\, s_2' \;:\; s_2 \Longrightarrow s_2'$ and $s_1'\, \mathcal{R}_\mathcal{N}\, s_2'$*

Two closed terms s_1 and s_2 are weak $\mathcal{N}$-bisimilar*, written $s_1 \approx_m^\mathcal{N} s_2$, if $s_1 \mathcal{R}_\mathcal{N} s_2$ for some $\mathcal{R}_\mathcal{N}$ in a weak labelled bisimulation. They are* weak labelled bisimilar*, written $s_1 \approx_m s_2$, if they are weak $\emptyset$-bisimilar. $\approx_m^\mathcal{N}$ is called weak $\mathcal{N}$-bisimilarity, while $\approx_m$ is called weak labelled bisimilarity.*

Results of congruence and coincidence still hold for the weak case. We omit the corresponding proofs because they do not require new techniques and, indeed, are the standard generalisation of the strong ones.

We conclude with an example inspired to the law $!(a(b).\,\bar{a}b) = \mathbf{0}$ that holds for weak bisimilarity in asynchronous π-calculus [7]. In fact, the analogous of equality (4.3) for the weak case is:

$$*\,[x]\,\mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle \approx_m \mathbf{0} \tag{4.4}$$

To prove validity, the most significant case is simulating the transition

$$*\,[x]\,\mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle \xrightarrow{\mathtt{n} \rhd [x]\,\langle x, v\rangle} *\,[x]\,(\mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle) \mid \mathtt{n}!\langle x, v\rangle$$

The term on the right of (4.4) replies with an empty transition (i.e. it does not perform any action) and it is easy to show that, for all v', $(*\,[x]\,(\mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle) \mid \mathtt{n}!\langle v', v\rangle)$ and $(\mathbf{0} \mid \mathtt{n}!\langle v', v\rangle)$ are weak bisimilar.

4.3.2 Observational semantics of μCOWS

As in the previous section, we extend the operational semantics of μCOWS presented in Section 3.2.2 by adding the rules in Tables 4.9 and 4.10 to those of Table 3.6, where

$$\frac{s_1 \xrightarrow{\mathtt{n} \rhd \bar{v}} s_1' \qquad s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2'}{s_1 \mid s_2 \xrightarrow{\emptyset} s_1' \mid s_2'}\ \textit{(match)} \qquad \frac{s \xrightarrow{\mathtt{n}\sigma\ell\bar{v}} s' \qquad n \in \mathtt{n}}{[n]\, s \xrightarrow{\sigma} [n]\, s'}\ \textit{(private)}$$

Table 4.10: μCOWS operational semantics (additional rules)

condition $u \notin (\mathrm{u}(\alpha) \cup \mathrm{ce}(\alpha))$ replaces $u \notin \mathrm{u}(\alpha)$ in the premise of rule *(del)*, and condition $|\sigma| \geqslant 1$ is added to the premise of *(com$_2$)*. Label α is now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd [\bar{n}]\,\bar{v} \quad | \quad \mathtt{n} \rhd [\bar{x}]\,\bar{w} \quad | \quad \mathtt{n}\sigma\ell\bar{v} \quad | \quad \sigma$$

We use $\mathrm{ce}(\alpha)$ to denote the names composing the endpoint in case α denotes execution of a communication, i.e. $\mathrm{ce}(\alpha)$ is $\emptyset$ except for $\alpha = \mathtt{n}\sigma\ell\bar{v}$ for which we let $\mathrm{ce}(\mathtt{n}\sigma\ell\bar{v}) = \mathtt{n}$.

This way, in μCOWS, two different kinds of communication labels can be generated: $\mathtt{n}\sigma\ell\bar{v}$ and σ. The former label, produced by rule *(com$_2$)* (shown in Table 3.6), carries information about the communication that has taken place (i.e. the endpoint, the transmitted values, the generated substitution and its length) and is used to check the presence of conflicting receives in parallel components. The check for presence of a conflict is not needed when either the performed receive has the highest priority (i.e. the substitution has length 0) or the communication takes place along a private endpoint. In the former case, label $\emptyset$ is immediately generated by rule *(match)* (this is guaranteed by condition $|\sigma| \geqslant 1$ in the premise of rule *(com$_2$)*). In the latter case, when the delimitation of a name belonging to the endpoint of a communication label is encountered (i.e. the communication is identified as private), the transition label $\mathtt{n}\sigma\ell\bar{v}$ is turned into σ by applying rule *(private)* (this is guaranteed by condition $u \notin \mathrm{ce}(\alpha)$ in the premise of rule *(del)*).

As we have done for μCOWS^m in the previous section, we define now open barbed and labelled bisimilarities for μCOWS.

4.3.2.1 Strong open barbed bisimilarity

The notion of basic observable defined for μCOWS^m (Definition 4.3.1) can be used also to define the open barbed bisimilarity for μCOWS.

Definition 4.3.7 (Open barbed bisimilarity) *A symmetric binary relation $\mathcal{R}$ on μCOWS closed terms is an* open barbed bisimulation *if whenever $s_1 \mathcal{R} s_2$ the following holds:*

(Barb preservation)	*if $s_1 \downarrow_{\mathtt{n}}$ then $s_2 \downarrow_{\mathtt{n}}$;*
(Computation closure)	*if $s_1 \xrightarrow{\emptyset} s_1'$ (resp. $s_1 \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_1'$) then there exists s_2' such that $s_2 \xrightarrow{\emptyset} s_2'$ (resp. $s_2 \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_2'$ or $\ell = \lvert\bar{v}\rvert \wedge s_2 \xrightarrow{\emptyset} s_2'$) and $s_1' \mathcal{R} s_2'$;*
(Context closure)	*$\mathbb{C}[\![s_1]\!]\,\mathcal{R}\,\mathbb{C}[\![s_2]\!]$, for every closed context $\mathbb{C}$.*

Two closed terms s_1 and s_2 are open barbed bisimilar*, written $s_1 \simeq_\mu s_2$, if $s_1 \mathcal{R} s_2$ for some open barbed bisimulation $\mathcal{R}$. $\simeq_\mu$ is called open barbed bisimilarity.*

The major difference with the definition of barbed bisimilarity for μCOWS^m (Definition 4.3.2) is that here we also take care of computations of the form $\mathtt{n}\,\emptyset\,\ell\,\bar{v}$.

We show now that in μCOWS it is not true any longer that receive activities are always unobservable. In fact, in Section 4.3.1 we have shown that, for μCOWS^m, $\sim_m$ enjoys the equality (4.3). Hence, by Theorem 4.3.3, we get that $[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \simeq_m \emptyset$, which means that receive activities cannot be observed (similarly to asynchronous π-calculus). In μCOWS, however, the context $\mathbb{C} \triangleq [y, z]\,\mathtt{n}?\langle y, z\rangle.\,\mathtt{m}!\langle\rangle \mid \mathtt{n}!\langle v', v\rangle \mid [\![\cdot]\!]$ can tell the two terms above apart. In fact, we have

$$\mathbb{C}[\![\emptyset]\!] \xrightarrow{\mathtt{n}\,\emptyset\,2\,\langle v', v\rangle} \mathtt{m}!\langle\rangle \mid \emptyset$$

where the term $(\mathtt{m}!\langle\rangle \mid \emptyset)$ satisfies the predicate $\downarrow_{\mathtt{m}}$. Instead, the other term cannot properly reply because the receive $\mathtt{n}?\langle x, v\rangle$ has higher priority than $\mathtt{n}?\langle y, z\rangle$ when synchronising with the invocation $\mathtt{n}!\langle v', v\rangle$. Thus, $\mathbb{C}[\![[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,)]\!]$ can only evolve to terms that cannot immediately satisfy the predicate $\downarrow_{\mathtt{m}}$. From this, we have

$$[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \not\simeq_\mu \emptyset \tag{4.5}$$

This means that, receive activities that exercise a priority (i.e. receives whose arguments contain some values) can be detected by an interacting observer.

Now, consider the term $[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle\,)$. Since $\mathtt{n}?\langle x, x'\rangle$ does not exercise any priority on parallel terms, we have that

$$[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle\,) \simeq_\mu \emptyset \qquad \mathbb{C}[\![[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle\,)]\!] \simeq_\mu \mathbb{C}[\![\emptyset]\!]$$

For similar reasons, we have that $\emptyset + \mathtt{n}?\langle\rangle.\,\mathtt{n}!\langle\rangle \simeq_\mu \emptyset$ and $\mathbb{D}[\![\emptyset + \mathtt{n}?\langle\rangle.\,\mathtt{n}!\langle\rangle]\!] \simeq_\mu \mathbb{D}[\![\emptyset]\!]$ for $\mathbb{D} \triangleq \mathtt{n}?\langle\rangle.\,\mathtt{m}!\langle\rangle \mid \mathtt{n}!\langle\rangle \mid [\![\cdot]\!]$.

Therefore, differently from μCOWS^m, communication in μCOWS is neither purely asynchronous nor purely synchronous. Indeed, receives having the smallest priority (i.e. whose arguments are, possible empty, tuples of variables) cannot be observed, while, by exploiting proper contexts, the other receives can be detected.

4.3.2.2 Strong labelled bisimilarity

For what we have seen in the previous paragraph, the equivalence $\sim_m$ is not suitable for characterising the open barbed bisimilarity for μCOWS. Indeed, $\sim_m$ enjoys equality (4.3) while $\simeq_\mu$ enjoys disequality (4.5). As consequence, $\sim_m$ is not preserved by all closed language contexts (which would prevent compositional reasoning).

Thus, we provide a new notion of bisimulation defined on top of the labelled transition system defining the semantics of μCOWS.

Definition 4.3.8 (Labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_\mathcal{N}\}_\mathcal{N}$ is a* labelled bisimulation *if, whenever $s_1 \mathcal{R}_\mathcal{N} s_2$ and $s_1 \xrightarrow{\alpha} s_1'$, where* $\mathrm{bu}(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*

(a) $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\, \bar{w}} s_2'$ *and*
$\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\, \mathcal{R}_{\mathcal{N}}\, s_2'\cdot\sigma$

(b) $|\bar{x}| = |\bar{w}|$ *and* $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ *and*
$\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x}, \bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\, \mathcal{R}_{\mathcal{N}}\, (s_2' \mid \mathtt{n}!\bar{v})$

2. *if* $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$ *where* $\ell = |\bar{v}|$ *then one of the following holds:*
   (a) $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s_2'$ *and* $s_1'\, \mathcal{R}_{\mathcal{N}}\, s_2'$ *(b)* $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ *and* $s_1'\, \mathcal{R}_{\mathcal{N}}\, s_2'$

3. *if* $\alpha = \mathtt{n} \lhd [\bar{n}]\, \bar{v}$ *where* $\mathtt{n} \notin \mathcal{N}$ *then* $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\, \bar{v}} s_2'$ *and* $s_1'\, \mathcal{R}_{\mathcal{N} \cup \bar{n}}\, s_2'$

4. *if* $\alpha = \emptyset$ *or* $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$*, where* $\ell \neq |\bar{v}|$*, then* $\exists\, s_2' \,:\, s_2 \xrightarrow{\alpha} s_2'$ *and* $s_1'\, \mathcal{R}_{\mathcal{N}}\, s_2'$

Two closed terms s_1 *and* s_2 *are* $\mathcal{N}$-bisimilar*, written* $s_1 \sim_\mu^{\mathcal{N}} s_2$*, if* $s_1 \mathcal{R}_{\mathcal{N}} s_2$ *for some* $\mathcal{R}_{\mathcal{N}}$ *in a labelled bisimulation. They are* labelled bisimilar*, written* $s_1 \sim_\mu s_2$*, if they are* $\emptyset$*-bisimilar.* $\sim_\mu^{\mathcal{N}}$ *is called* $\mathcal{N}$*-bisimilarity, while* $\sim_\mu$ *is called labelled bisimilarity.*

Clause 1 deals with both observable and unobservable receives. In fact, all receives can be simulated in a normal way (clause 1.(a)); additionally, receives such that $|\bar{x}| = |\bar{w}|$, i.e. $\bar{w}$ contains only variables or is the empty tuple (since $\bar{x} \subseteq \bar{w}$ and $\bar{w} \backslash \bar{x}$ does not contain variables), can be simulated by an internal action leading to a term that, when composed with the invoke activity consumed by the receive, stands in the appropriate relation (clause 1.(b)). For similar reasons, clause 2 permits replying with an $\emptyset$-transition to communications involving an unobservable receive ($\ell = |\ \bar{v}\ |$ implies that the tuple argument of the receive is either empty or only contains variables). Indeed, such clause is explained by the following equation: $\mathbb{C}[\![[x, x']\,(\emptyset + \mathtt{n}?\langle x, x'\rangle.\, \mathtt{n}!\langle x, x'\rangle)]\!] \sim_\mu \mathbb{C}[\![\emptyset]\!]$ for $\mathbb{C} \triangleq [\![\cdot]\!] \mid \mathtt{n}!\langle v', v\rangle$. In fact, if we do not have that clause, we would have that

$$\mathbb{C}[\![[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\, \mathtt{n}!\langle x, x'\rangle\,)]\!] \xrightarrow{\mathtt{n}\,\emptyset\,2\,\langle v', v\rangle} \quad \text{and} \quad \mathbb{C}[\![\emptyset]\!] \xrightarrow{\mathtt{n}\,\emptyset\,2\,\langle v', v\rangle} \not\longrightarrow$$

which would imply that $\mathbb{C}[\![[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\, \mathtt{n}!\langle x, x'\rangle\,)]\!] \not\sim_\mu \mathbb{C}[\![\emptyset]\!]$.

Differently from labelled bisimulation for μCOWS^m, here in clause 1 the two continuations are not related for any tuple of values, but only for those tuples that can be effectively received (i.e. that do not give rise to communication conflicts). Indeed, for example, the following two μCOWS terms:

$$\begin{aligned} s_1 &\triangleq [x]\,(\,\mathtt{n}?\langle v\rangle \;+\; \mathtt{n}?\langle x\rangle.\, [\mathtt{m}]\,(\mathtt{m}!\langle x\rangle \mid \mathtt{m}?\langle x\rangle + \mathtt{m}?\langle v\rangle.\, \mathtt{m}'!\langle\rangle)\,) \\ s_2 &\triangleq [x]\,(\,\mathtt{n}?\langle v\rangle \;+\; \mathtt{n}?\langle x\rangle.\, [\mathtt{m}]\,(\mathtt{m}!\langle x\rangle \mid \mathtt{m}?\langle x\rangle)\,) \end{aligned}$$

are both barbed and labelled bisimilar. Instead, if the definition of bisimulation does not have condition $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\ \bar{x}\ |)$ in clause 1, we would have that $s_1 \simeq_\mu s_2$ but $s_1 \not\sim_\mu s_2$. Indeed, after execution of action $\mathtt{n} \rhd [\bar{x}]\, \bar{x}$, it is not correct to consider

the continuations under substitutions $\{x \mapsto v'\}$ for all v', because such action cannot be performed in case the received value v' is v (it is pre-empted by $\mathtt{n}?\langle v\rangle$).

In clauses 1.(b) and 2.(b) we require that s_2 can only reply with an $\emptyset$-transition and not with a transition labelled by $\mathtt{m}\,\emptyset\,\ell\,\bar{v}$. Otherwise, the relation would not be preserved by parallel composition, because communication $\mathtt{m}\,\emptyset\,\ell\,\bar{v}$ is subject to conflict check and, hence, could be blocked by a receive with higher priority performed by a parallel term.

Some illustrative equalities follow.

1. $\mathtt{n}!\langle\rangle \mid \mathtt{n}?\langle\rangle \not\sim_\mu \mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle$
2. $[\mathtt{n}]\,(\mathtt{n}!\langle\rangle \mid \mathtt{n}?\langle\rangle) \sim_\mu [\mathtt{m}]\,(\mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle) \sim_\mu [\mathtt{m}]\,(\mathtt{m}!\langle\bar{v}\rangle \mid [\bar{x}]\,\mathtt{m}?\langle\bar{x}\rangle)$
3. $[x]\,(\,\emptyset + \mathtt{n}?\langle x, v\rangle.\,\mathtt{n}!\langle x, v\rangle\,) \not\sim_\mu \emptyset$
4. $[x, x']\,(\,\emptyset + \mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle\,) \sim_\mu \emptyset$
5. $\emptyset + \mathtt{n}?\langle\rangle.\,\mathtt{n}!\langle\rangle \sim_\mu \emptyset$
6. $\mathbb{C}[\![[x, x']\,(\emptyset + \mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle)]\!] \sim_\mu \mathbb{C}[\![\emptyset]\!]$ for $\mathbb{C} \triangleq [y, z]\,\mathtt{n}?\langle y, z\rangle.\,\mathtt{n}'!\langle\rangle \mid \mathtt{n}!\langle v', v\rangle \mid [\![\cdot]\!]$

Remark 4.3.1 (On computational steps) μCOWS supports two different kinds of computational steps, labelled by $\emptyset$ and $\mathtt{n}\,\emptyset\,\ell\,\bar{v}$. To be a congruence, labelled bisimilarity deals with them separately. Indeed, let τ denote any computational step, and suppose to weaken Definition 4.3.8 by replacing $\emptyset$ with τ in clause 1, by removing clause 2, and by modifying clause 4 in order to only consider the case $\alpha = \tau$. We would have that $*\,\mathtt{n}!\langle v\rangle \mid \emptyset \sim_\mu *\,\mathtt{n}!\langle v\rangle \mid [x]\,\mathtt{n}?\langle x\rangle.\,\mathtt{n}!\langle x\rangle$. In fact, the relevant cases are:

- if $*\,\mathtt{n}!\langle v\rangle \mid \emptyset \xrightarrow{\emptyset} *\,\mathtt{n}!\langle v\rangle$ then $*\,\mathtt{n}!\langle v\rangle \mid [x]\,\mathtt{n}?\langle x\rangle.\,\mathtt{n}!\langle x\rangle \xrightarrow{\mathtt{n}\,\emptyset\,1\,\langle v\rangle} *\,\mathtt{n}!\langle v\rangle \mid \mathtt{n}!\langle v\rangle \equiv *\,\mathtt{n}!\langle v\rangle$, and vice versa;
- if $*\,\mathtt{n}!\langle v\rangle \mid [x]\,\mathtt{n}?\langle x\rangle.\,\mathtt{n}!\langle x\rangle \xrightarrow{\mathtt{n}\rhd[\langle x\rangle]\,\langle x\rangle} *\,\mathtt{n}!\langle v\rangle \mid \mathtt{n}!\langle x\rangle$ then $*\,\mathtt{n}!\langle v\rangle \mid \emptyset \xrightarrow{\emptyset} *\,\mathtt{n}!\langle v\rangle$ and for all v' holds that $*\,\mathtt{n}!\langle v\rangle \mid \mathtt{n}!\langle x\rangle \cdot \{x \mapsto v'\}$ and $*\,\mathtt{n}!\langle v\rangle \mid \mathtt{n}!\langle v'\rangle$ are bisimilar.

However, the context $\mathtt{n}?\langle v\rangle \mid [\![\cdot]\!]$ can tell the two terms apart, because the transition labelled $\mathtt{n}\,\emptyset\,1\,\langle v\rangle$ can be blocked by the activity $\mathtt{n}?\langle v\rangle$ (which has higher priority than $\mathtt{n}?\langle x\rangle$). This means that the modified labelled bisimilarity is not a congruence for μCOWS. Of course, the two terms above are not bisimilar according to Definition 4.3.8.

Now, we prove that labelled bisimilarity $\sim_\mu$ is a congruence for μCOWS. To this aim, besides the generalisation of $\sim_\mu$ to open terms and a lemma showing that a bisimulation up-to $\equiv$ can be used as a sound proof-technique for labelled bisimulation, we need a lemma establishing preservation of predicate noConf(_, _, _, _) by the relations belonging to a bisimulation. As usual, we only outline here the techniques used in the proofs and refer the interested reader to Appendix B.2 for a full account.

Definition 4.3.9 *Let s_1 and s_2 be two μ*COWS *terms containing free variables $\bar{x}$ at most. Then $s_1 \sim_\mu^{\mathcal{N}} s_2$ if, for all values $\bar{v}$ such that $|\bar{x}|=|\bar{v}|$, $s_1 \cdot \{\bar{x} \mapsto \bar{v}\} \sim_\mu^{\mathcal{N}} s_2 \cdot \{\bar{x} \mapsto \bar{v}\}$.*

Lemma 4.3.2 *Let $\mathcal{F}$ be a labelled bisimulation up-to $\equiv$; then, $\mathcal{F}$ is a labelled bisimulation.*

Proof. The proof proceeds as the proof of Lemma 4.3.1. □

Lemma 4.3.3 *Let s_1 and s_2 be two μ*COWS *closed terms and $\mathcal{R}$ be a relation belonging to a labelled bisimulation such that $s_1 \mathcal{R} s_2$. Then,* $\mathrm{noConf}(s_1, \mathtt{n}, \bar{v}, \ell) = \mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell)$ *for any $\mathtt{n}$, $\bar{v}$ and ℓ, with $\ell \leqslant |\bar{v}|$.*

Proof (sketch). The lemma is proved by contradiction. □

Theorem 4.3.4 *$\sim_\mu$ is a congruence for μ*COWS *closed terms.*

Proof (sketch). We shall prove that, given two μCOWS closed terms s_1 and s_2, if $s_1 \sim_\mu s_2$ then $\mathbb{C}[\![s_1]\!] \sim_\mu \mathbb{C}[\![s_2]\!]$ for every (possibly open) context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$ and proceeds similarly to that of Theorem 4.3.1. □

Now, we prove that labelled bisimilarity is *sound* and *complete* with respect to open barbed bisimilarity.

Theorem 4.3.5 (Soundness of $\sim_\mu$ w.r.t. $\simeq_\mu$) *Given two μ*COWS *closed terms s_1 and s_2, if $s_1 \sim_\mu s_2$ then $s_1 \simeq_\mu s_2$.*

Proof (sketch). By Theorem 4.3.4, we have that $\sim_\mu$ is context closed. Thus, we only need to prove that $\sim_\mu$ is barb preserving and computation closed. □

Theorem 4.3.6 (Completeness of $\sim_\mu$ w.r.t. $\simeq_\mu$) *Given two μ*COWS *closed terms s_1 and s_2, if $s_1 \simeq_\mu s_2$ then $s_1 \sim_\mu s_2$.*

Proof (sketch). We define a family of relations $\mathcal{F} = \{\mathcal{R}_{\mathcal{N}} : \mathcal{N} \text{ set of names}\}$ such that $\simeq_\mu$ is included in $\mathcal{R}_\emptyset$, and show that it is a labelled bisimulation. Let $\mathcal{N}$ be the set $\{n_1, \ldots, n_m\}$, then $s_1 \mathcal{R}_{\mathcal{N}} s_2$ if there exist $\mathtt{m}_1, \ldots, \mathtt{m}_m$ fresh such that

$$[n_1, \ldots, n_m](s_1 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle) \simeq_\mu [n_1, \ldots, n_m](s_2 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle)$$

Take $s_1 \mathcal{R}_{\mathcal{N}} s_2$ and a transition $s_1 \xrightarrow{\alpha} s_1'$; then, the proof proceeds by case analysis on α. □

Corollary 4.3.2 *$\sim_\mu$ and $\simeq_\mu$ coincide.*

Proof. Directly from Theorems 4.3.5 and 4.3.6. □

4.3.2.3 Weak open barbed and labelled bisimilarities

As in Section 4.3.1.3, the observational semantic theories introduced for μCOWS extend in a standard way to the weak case.

Weak open barbed bisimilarity, namely $\approx_\mu$, is obtained by replacing $s \downarrow_{\mathtt{n}}$ with $s \Downarrow_{\mathtt{n}}$, $\xrightarrow{\emptyset}$ with $\Longrightarrow$, and $\xrightarrow{\mathtt{n}\emptyset\ell\bar{v}}$ with $\overset{\mathtt{n}\emptyset\ell\bar{v}}{\Longrightarrow}$ in Definition 4.3.7.

To define weak labelled bisimilarity we replace $\xrightarrow{\alpha}$ with $\overset{\hat{\alpha}}{\Longrightarrow}$ in the four clauses of Definition 4.3.8.

Definition 4.3.10 (Weak labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_\mathcal{N}\}_\mathcal{N}$ is a* weak labelled bisimulation *if, whenever $s_1 \mathcal{R}_\mathcal{N} s_2$ and $s_1 \xrightarrow{\alpha} s_1'$, where* $\mathrm{bu}(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*

   (a) $\exists\, s_2'' \,:\, s_2 \overset{\mathtt{n}\rhd[\bar{x}]\,\bar{w}}{\longrightarrow\!\!\!\!\Longrightarrow} s_2''$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\sigma, |\bar{x}|)$
   $\exists\, s_2' \,:\, s_2''\cdot\sigma \Longrightarrow s_2'$ *and* $s_1'\cdot\sigma\, \mathcal{R}_\mathcal{N}\, s_2'$

   (b) $|\bar{x}| = |\bar{w}|$ *and* $\exists\, s_2' \,:\, s_2 \Longrightarrow s_2'$ *and*
   $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\, \mathcal{R}_\mathcal{N}\, (s_2' \mid \mathtt{n}!\bar{v})$

2. *if $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$ where $\ell = |\bar{v}|$ then one of the following holds:*
   (a) $\exists\, s_2' \,:\, s_2 \overset{\mathtt{n}\emptyset\ell\bar{v}}{\Longrightarrow} s_2'$ *and* $s_1'\, \mathcal{R}_\mathcal{N}\, s_2'$ *(b)* $\exists\, s_2' \,:\, s_2 \Longrightarrow s_2'$ *and* $s_1'\, \mathcal{R}_\mathcal{N}\, s_2'$

3. *if $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$ where $\mathtt{n} \notin \mathcal{N}$ then* $\exists\, s_2' \,:\, s_2 \overset{\mathtt{n}\lhd[\bar{n}]\,\bar{v}}{\Longrightarrow} s_2'$ *and* $s_1'\, \mathcal{R}_{\mathcal{N}\cup\bar{n}}\, s_2'$

4. *if $\alpha = \emptyset$ or $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$, where $\ell \neq |\bar{v}|$, then* $\exists\, s_2' \,:\, s_2 \overset{\hat{\alpha}}{\Longrightarrow} s_2'$ *and* $s_1'\, \mathcal{R}\, s_2'$

Two closed terms s_1 and s_2 are weak $\mathcal{N}$-bisimilar*, written $s_1 \approx_\mu^\mathcal{N} s_2$, if $s_1 \mathcal{R}_\mathcal{N} s_2$ for some $\mathcal{R}_\mathcal{N}$ in a weak labelled bisimulation. They are* weak labelled bisimilar*, written $s_1 \approx_\mu s_2$, if they are weak $\emptyset$-bisimilar. $\approx_\mu^\mathcal{N}$ is called weak $\mathcal{N}$-bisimilarity, while $\approx_\mu$ is called weak labelled bisimilarity.*

The results of congruence and coincidence still hold for the weak case. We omit the corresponding proofs because they do not require new techniques and, indeed, are the standard generalisation of the strong ones.

We conclude with an example that is the analogous of equality (4.4) for μCOWS:

$$*\,[x, x']\,\mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle \approx_\mu \mathbf{0}$$

To prove validity, the most significant case is simulating the transition

$$*\,[x, x']\,\mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle \xrightarrow{\mathtt{n}\rhd[x,x']\,\langle x,x'\rangle} *\,[x, x']\,(\mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle) \mid \mathtt{n}!\langle x, x'\rangle$$

The term on the right replies with an empty transition and it is easy to show that, for all v and v', $(*\,[x, x']\,(\mathtt{n}?\langle x, x'\rangle.\,\mathtt{n}!\langle x, x'\rangle) \mid \mathtt{n}!\langle v, v'\rangle)$ and $(\mathbf{0} \mid \mathtt{n}!\langle v, v'\rangle)$ are weak bisimilar.

4.3.3 Observational semantics of COWS

We extend the operational semantics of COWS presented in Section 3.2.3 as we have done in the previous sections, i.e. we add the rules in Tables 4.9 and 4.10 to those of Table 3.12, where condition $e \notin (\mathrm{e}(\alpha) \cup \mathrm{ce}(\alpha))$ replaces $e \notin \mathrm{e}(\alpha)$ in the premise of rule *(del$_2$)* and condition $|\sigma| \geqslant 1$ is added to the premise of *(com$_2$)*. Thus, labels are now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd [\bar{n}]\,\bar{v} \quad | \quad \mathtt{n} \rhd [\bar{x}]\,\bar{w} \quad | \quad \mathtt{n}\,\sigma\,\ell\,\bar{v} \quad | \quad \sigma \quad | \quad k \quad | \quad \dagger$$

When considering observational semantics for COWS we soon discover that $\sim_\mu$ is not preserved by those contexts forcing termination of the activities in the hole. For example, $\emptyset \sim_\mu \{\!|\emptyset|\!\}$ trivially holds. However, the COWS context $[k]\,(\mathbf{kill}(k) \mid [\![\cdot]\!])$ can tell the two terms apart. Indeed, $[k]\,(\mathbf{kill}(k) \mid \emptyset) \sim_\mu [k]\,(\mathbf{kill}(k) \mid \{\!|\emptyset|\!\})$ does not hold since, after execution of the kill activity (that has highest priority), we would get $\mathbf{0} \sim_\mu \{\!|\emptyset|\!\}$ which is trivially false. Therefore, $\sim_\mu$ is not a congruence for COWS.

4.3.3.1 Strong open barbed bisimilarity

Open barbed bisimilarity is by definition closed under all contexts and its definition only needs to be tuned for considering also †-transitions in the 'Computation closure'.

Definition 4.3.11 (Open barbed bisimilarity) *A symmetric binary relation $\mathcal{R}$ on* COWS *closed terms is an* open barbed bisimulation *if whenever $s_1 \mathcal{R} s_2$ the following holds:*

(Barb preservation) *if $s_1 \downarrow_{\mathtt{n}}$ then $s_2 \downarrow_{\mathtt{n}}$;*

(Computation closure) *if $s_1 \xrightarrow{\emptyset} s_1'$, $s_1 \xrightarrow{\dagger} s_1'$, $s_1 \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_1'$ respectively, then there exists s_2' such that $s_2 \xrightarrow{\emptyset} s_2'$, $s_2 \xrightarrow{\dagger} s_2'$, $s_2 \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_2'$ or $\ell = |\bar{v}| \wedge s_2 \xrightarrow{\emptyset} s_2'$ respectively, and $s_1' \mathcal{R} s_2'$;*

(Context closure) *$\mathbb{C}[\![s_1]\!]\,\mathcal{R}\,\mathbb{C}[\![s_2]\!]$, for every closed context $\mathbb{C}$.*

Two closed terms s_1 and s_2 are open barbed bisimilar*, written $s_1 \simeq s_2$, if $s_1 \mathcal{R} s_2$ for some open barbed bisimulation $\mathcal{R}$. $\simeq$ is called open barbed bisimilarity.*

4.3.3.2 Strong labelled bisimilarity

Labelled bisimilarity must explicitly take care of the terms resulting from application of function *halt*(_) (that gets the same effect as of plunging a term within the context $[k]\,(\mathbf{kill}(k) \mid [\![\cdot]\!])$). It must also consider that if a closed term s performs a transition labelled by $\mathtt{n} \lhd [\bar{m}]\,\bar{v}$, then s contains an invoke of the form $\mathtt{n}!\bar{\epsilon}$, with $[\![\bar{\epsilon}]\!] = \bar{v}$, which can be either protected or not. These differences w.r.t. to Definition 4.3.8 are highlighted with a gray background.

Definition 4.3.12 (Labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_{\mathcal{N}}\}_{\mathcal{N}}$ is a* labelled bisimulation *if $s_1 \mathcal{R}_{\mathcal{N}} s_2$ then $halt(s_1)\,\mathcal{R}_{\mathcal{N}}\, halt(s_2)$ and if $s_1 \xrightarrow{\alpha} s_1'$, where* $\mathrm{bu}(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*

   (a) $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n}\rhd[\bar{x}]\,\bar{w}} s_2'$ *and*
   $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\, \mathcal{R}_{\mathcal{N}}\, s_2'\cdot\sigma$

   (b) $|\bar{x}|=|\bar{w}|$ *and* $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and*
   $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\, \mathcal{R}_{\mathcal{N}}\, (s_2' \mid \mathtt{n}!\bar{v})$ *or* $s_1'\cdot\sigma\, \mathcal{R}_{\mathcal{N}}\, (s_2' \mid \{\!|\mathtt{n}!\bar{v}|\!\})$

2. *if $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$ where $\ell = |\bar{v}|$ then one of the following holds:*
   (a) $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s_2'$ *and* $s_1'\,\mathcal{R}_{\mathcal{N}}\, s_2'$ *(b)* $\exists\, s_2' \,:\, s_2 \xrightarrow{\emptyset} s_2'$ *and* $s_1'\,\mathcal{R}_{\mathcal{N}}\, s_2'$

3. *if $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$ where $\mathtt{n} \notin \mathcal{N}$ then* $\exists\, s_2' \,:\, s_2 \xrightarrow{\mathtt{n}\lhd[\bar{n}]\,\bar{v}} s_2'$ *and* $s_1'\,\mathcal{R}_{\mathcal{N}\cup\bar{n}}\, s_2'$

4. *if $\alpha = \emptyset$, $\alpha = \dagger$ or $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$, where $\ell \neq |\bar{v}|$, then* $\exists\, s_2' \,:\, s_2 \xrightarrow{\alpha} s_2'$ *and* $s_1'\,\mathcal{R}_{\mathcal{N}}\, s_2'$

*Two closed terms s_1 and s_2 are $\mathcal{N}$-*bisimilar*, written $s_1 \sim^{\mathcal{N}} s_2$, if $s_1 \mathcal{R}_{\mathcal{N}} s_2$ for some $\mathcal{R}_{\mathcal{N}}$ in a labelled bisimulation. They are* labelled bisimilar*, written $s_1 \sim s_2$, if they are $\emptyset$-bisimilar. $\sim^{\mathcal{N}}$ is called $\mathcal{N}$-bisimilarity, while $\sim$ is called labelled bisimilarity.*

Notably, *halt*-closure takes into account execution of outer kill activities (i.e. kills performed by contexts), while the inner ones that are active in the considered terms are taken into account by clause 4.

Remark 4.3.2 (On computational steps) COWS supports three different kinds of computational steps (labelled by $\emptyset$, $\mathtt{n}\,\emptyset\,\ell\,\bar{v}$, or $\dagger$) and they are dealt with separately because the priority of forced termination over communication permits to observe the kind of computational steps that take place. Indeed, let τ denote any computational step, and suppose to weaken computation closure in Definition 4.3.11 and clause 4 of Definition 4.3.12 by only requiring that if $s_1 \xrightarrow{\tau}$ then $s_2 \xrightarrow{\tau}$. We would have that $[k]\,\mathbf{kill}(k) \not\sim \emptyset$ while $[k]\,\mathbf{kill}(k) \sim \emptyset$. In fact, the above terms are easily distinguished by the context $[k']\,(\mathbf{kill}(k') \mid [\![\cdot]\!])$, because the term obtained by filling the context with the term on the left can perform the following transitions

$$[k']\,(\mathbf{kill}(k') \mid [k]\,\mathbf{kill}(k)) \xrightarrow{\tau} [k']\,\mathbf{kill}(k') \xrightarrow{\tau} \mathbf{0}$$

while, due to priority of kill over communication, the other term can only reply as follows

$$[k']\,(\mathbf{kill}(k') \mid \emptyset) \xrightarrow{\tau} \mathbf{0} \not\xrightarrow{\tau}$$

This means that $[k'](\mathbf{kill}(k') \mid [k]\,\mathbf{kill}(k)) \not\simeq [k'](\mathbf{kill}(k') \mid \emptyset)$, and hence $[k]\,\mathbf{kill}(k) \not\simeq \emptyset$. Moreover, it would hold that $[k'](\mathbf{kill}(k') \mid [k]\,\mathbf{kill}(k)) \not\sim [k'](\mathbf{kill}(k') \mid \emptyset)$, which also implies that $\sim$ would not be a congruence for COWS.

Thus, to enable $\sim$ to tell $[k]\,\mathbf{kill}(k)$ and $\emptyset$ apart, we must allow $\sim$ to distinguish computational steps (as in clause 4 of Definition 4.3.12). This way, we also have that the following holds:

$$\{\!|\, [k]\,\mathbf{kill}(k) \,|\!\} \not\sim \{\!|\, \emptyset \,|\!\}$$

However, they have the same barbs and no context can take them apart. Thus,

$$\{\!|\, [k]\,\mathbf{kill}(k) \,|\!\} \simeq \{\!|\, \emptyset \,|\!\}$$

Therefore, to have a sound and complete characterisation of $\simeq$ in terms of $\sim$, we must allow $\simeq$ to distinguish computational steps (as in the computation closure required by Definition 4.3.11).

We can now prove that labelled bisimilarity is a congruence for COWS.

Definition 4.3.13 *Let s_1 and s_2 be two* COWS *term containing free variables $\bar{x}$ at most. Then $s_1 \sim^{\mathcal{N}} s_2$ if, for all values $\bar{v}$ such that $|\bar{x}|=|\bar{v}|$, $s_1 \cdot \{\bar{x} \mapsto \bar{v}\} \sim^{\mathcal{N}} s_2 \cdot \{\bar{x} \mapsto \bar{v}\}$.*

Theorem 4.3.7 *$\sim$ is a congruence for* COWS.

Proof (sketch). We shall prove that, given two COWS closed terms s_1 and s_2, if $s_1 \sim s_2$ then $\mathbb{C}[\![s_1]\!] \sim \mathbb{C}[\![s_2]\!]$ for every context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$. □

We prove now that barbed bisimilarity and labelled bisimilarity coincide.

Theorem 4.3.8 (Soundness of $\sim$ w.r.t. $\simeq$) *Given two* COWS *closed terms s_1 and s_2, if $s_1 \sim s_2$ then $s_1 \simeq s_2$.*

Proof. The proof proceeds as the proof of Theorem 4.3.5, by also exploiting the fact that, by Definition 4.3.12, if $s_1 \xrightarrow{\dagger} s_1'$ then $s_2 \xrightarrow{\dagger} s_2'$ and $s_1' \sim s_2'$, for some s_2'. □

Theorem 4.3.9 (Completeness of $\sim$ w.r.t. $\simeq$) *Given two* COWS *closed terms s_1 and s_2, if $s_1 \simeq s_2$ then $s_1 \sim s_2$.*

Proof. The proof proceeds as the proof of Theorem 4.3.6, by also exploiting the fact that, by Definition 4.3.11, if $s_1 \xrightarrow{\dagger} s_1'$ then $s_2 \xrightarrow{\dagger} s_2'$ and $s_1' \simeq s_2'$, for some s_2'. □

Corollary 4.3.3 *$\sim$ and $\simeq$ coincide.*

Proof. Directly from Theorems 4.3.8 and 4.3.9. □

4.3.3.3 Weak open barbed and labelled bisimilarities

Extension to the weak case is standard. Let τ denote any computational step, i.e. either $\emptyset$ or $\dagger$. With respect to the strong case, and to the analogous extension for μCOWS^m and μCOWS, we relax the requirement for the computational steps, by merely requiring each τ to be matched by zero or more τ. Specifically, weak transitions are defined as follows: $\Longrightarrow$ means $(\stackrel{\tau}{\longrightarrow})^*$, i.e. zero or more τ-transitions; $\stackrel{\alpha}{\Longrightarrow}$ means $\Longrightarrow\stackrel{\alpha}{\longrightarrow}\Longrightarrow$; $\stackrel{\hat{\alpha}}{\Longrightarrow}$ means $\stackrel{\alpha}{\Longrightarrow}$ if $\alpha \neq \tau$ and $\Longrightarrow$ if $\alpha = \tau$. Thus, we obtain the following definition of weak bisimulations.

Weak open barbed bisimilarity, namely $\approx$, is obtained by replacing $s \downarrow_\mu$ with $s \Downarrow_\mu$, $\stackrel{\emptyset}{\longrightarrow}$ and $\stackrel{\dagger}{\longrightarrow}$ with $\Longrightarrow$, and $\stackrel{\mathtt{n}\emptyset\ell\bar{v}}{\longrightarrow}$ with $\stackrel{\mathtt{n}\emptyset\ell\bar{v}}{\Longrightarrow}$ in Definition 4.3.11.

Definition 4.3.14 (Weak labelled bisimilarity) *A names-indexed family of relations $\{\mathcal{R}_\mathcal{N}\}_\mathcal{N}$ is a* weak labelled bisimulation *if $s_1 \mathcal{R}_\mathcal{N} s_2$ then $halt(s_1)\,\mathcal{R}_\mathcal{N}\, halt(s_2)$ and if $s_1 \stackrel{\alpha}{\longrightarrow} s_1'$, where* bu$(\alpha)$ *are fresh, then:*

1. *if $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ then one of the following holds:*

   (a) $\exists\, s_2'' \,:\, s_2 \stackrel{\mathtt{n}\rhd[\bar{x}]\,\bar{w}}{\Longrightarrow} s_2''$ *and* $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\sigma, |\bar{x}|)$
   $\exists\, s_2' \,:\, s_2''\cdot\sigma \Longrightarrow s_2'$ *and* $s_1'\cdot\sigma\,\mathcal{R}_\mathcal{N}\, s_2'$

   (b) $|\bar{x}|=|\bar{w}|$ *and* $\exists\, s_2' \,:\, s_2 \Longrightarrow s_2'$ *and*
   $\forall\, \bar{v}$ *s.t.* $\mathcal{M}(\bar{x},\bar{v}) = \sigma$ *and* $\mathrm{noConf}(s_2, \mathtt{n}, \bar{w}\cdot\sigma, |\bar{x}|) \,:\, s_1'\cdot\sigma\,\mathcal{R}_\mathcal{N}\,(s_2' \mid \mathtt{n}!\bar{v})$
   or $s_1'\cdot\sigma\,\mathcal{R}_\mathcal{N}\,(s_2' \mid \{\!|\mathtt{n}!\bar{v}|\!\})$

2. *if $\alpha = \mathtt{n}\emptyset\ell\,\bar{v}$ where $\ell = |\bar{v}|$ then one of the following holds:*
   (a) $\exists\, s_2' \,:\, s_2 \stackrel{\mathtt{n}\emptyset\ell\bar{v}}{\Longrightarrow} s_2'$ *and* $s_1'\,\mathcal{R}_\mathcal{N}\, s_2'$ (b) $\exists\, s_2' \,:\, s_2 \Longrightarrow s_2'$ *and* $s_1'\,\mathcal{R}_\mathcal{N}\, s_2'$

3. *if $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$ where $\mathtt{n} \notin \mathcal{N}$ then* $\exists\, s_2' \,:\, s_2 \stackrel{\mathtt{n}\lhd[\bar{n}]\,\bar{v}}{\Longrightarrow} s_2'$ *and* $s_1'\,\mathcal{R}_{\mathcal{N}\cup\bar{n}}\, s_2'$

4. *if $\alpha = \emptyset$, $\alpha = \dagger$ or $\alpha = \mathtt{n}\emptyset\ell\,\bar{v}$, where $\ell \neq |\bar{v}|$, then* $\exists\, s_2' \,:\, s_2 \stackrel{\hat{\alpha}}{\Longrightarrow} s_2'$ *and* $s_1'\,\mathcal{R}_\mathcal{N}\, s_2'$

Two closed terms s_1 and s_2 are weak $\mathcal{N}$-bisimilar*, written $s_1 \approx^\mathcal{N} s_2$, if $s_1 \mathcal{R}_\mathcal{N} s_2$ for some $\mathcal{R}_\mathcal{N}$ in a weak labelled bisimulation. They are* weak labelled bisimilar*, written $s_1 \approx s_2$, if they are weak $\emptyset$-bisimilar. $\approx^\mathcal{N}$ is called weak $\mathcal{N}$-bisimilarity, while $\approx$ is called weak labelled bisimilarity.*

Again, results of congruence and coincidence hold.

4.3.3.4 Analysing the 'Morra game' scenario

We now study the relationship between the high- and low-level specifications of the Morra service introduced in Section 3.1. We can prove that (3.1) $\not\approx$ (3.2). Indeed, (3.1) can perform transitions $\stackrel{evens \bullet throw \rhd [x_{id}, y_p, y_{num}]\,\langle x_{id}, y_p, y_{num}\rangle}{\longrightarrow}$ and $\stackrel{odds \bullet throw \rhd [x_p, x_{num}]\,\langle first, x_p, x_{num}\rangle}{\longrightarrow}$

and, because of application of substitutions $\{x_{id} \mapsto first, y_p \mapsto cbB, y_{num} \mapsto 1\}$ and $\{x_p \mapsto cbA, x_{num} \mapsto 2\}$, evolve to

$$(\, cbA \bullet res!\langle first, win(2,1,1)\rangle \mid cbB \bullet res!\langle first, win(2,1,0)\rangle \,)$$

Instead, (3.2) can properly simulate the above transitions but it can only evolve to

$$\{\!|\, cbA \bullet res!\langle first, w\rangle \mid cbB \bullet res!\langle first, l\rangle \,|\!\}$$

Of course, the latter term behaves differently from the former one in presence of kill activities. In fact, given the context $\mathbb{C} \triangleq [k']\,(\,[\mathrm{n}]\,(\mathrm{n}!\langle\rangle \mid \mathrm{n}?\langle\rangle.\,\mathbf{kill}(k')) \mid [\![\cdot]\!]\,)$, we have that $\mathbb{C}[\![(3.1)]\!] \not\approx \mathbb{C}[\![(3.2)]\!]$.

If we modify the last two invoke activities in the high-level specification (3.1) as follows:

$$\begin{array}{l} *\,[x_{id}, x_p, x_{num}, y_p, y_{num}] \\ \quad (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\ \qquad \mid \{\!|\, x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle \,|\!\}\,) \end{array} \tag{4.6}$$

the problem persists, because (4.6) after the first two transitions always provides a response, while (3.2) could fail to provide a response in presence of kill activities. Instead, by replacing M in (3.2) by

$$\begin{array}{l} [x_{id}, x_p, x_{num}, y_p, y_{num}] \\ (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \mid \{\!|\, [k]\,(\ldots) \,|\!\}\,) \end{array}$$

we can equate (4.6) with the so obtained low-level specification.

We can also prove that M is congruent to the following variant:

$$\begin{array}{l} [x_{id}, x_p, x_{num}, y_p, y_{num}] \\ (\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\ \quad \mid m \bullet req2f!\langle x_{id}, x_{num}, y_{num}\rangle \mid m \bullet req5f!\langle x_{id}, x_{num}, y_{num}\rangle \\ \quad \mid [x_o, x_e]\,(\, m \bullet resp2f?\langle x_{id}, x_o, x_e\rangle.\, \{\!|\, x_p \bullet res!\langle x_{id}, x_o\rangle \mid y_p \bullet res!\langle x_{id}, x_e\rangle |\!\} \\ \qquad\qquad + m \bullet resp5f?\langle x_{id}, x_o, x_e\rangle.\, \{\!|\, x_p \bullet res!\langle x_{id}, x_o\rangle \mid y_p \bullet res!\langle x_{id}, x_e\rangle |\!\}\,)\,) \end{array}$$

This somehow suggests us that we can use kill, delimitation and protection to model choice among receive activities that prefix protected terms, like in the following case

$$[\bar{x}]\,(\,\mathrm{n}?\bar{w}.\,\{\!|s|\!\} + \mathrm{n}'?\bar{w}'.\,\{\!|s'|\!\}\,) \;\approx\; [k, \bar{x}]\,(\,\mathrm{n}?\bar{w}.\,(\mathbf{kill}(k) \mid \{\!|s|\!\}) \mid \mathrm{n}'?\bar{w}'.\,(\mathbf{kill}(k) \mid \{\!|s'|\!\})\,)$$

4.3.4 Concluding remarks

We have investigated the impact of COWS's dynamic priority mechanisms combined with local pre-emption on the definitions of semantic theories for SOC systems. We have introduced natural notions of strong and weak open barbed bisimilarities for COWS, and then proved their coincidence with more manageable characterisations in terms of labelled bisimilarities. We have also demonstrated our approach through the analysis of the example specified in Section 3.1.

We leave for future work the identification of appropriate sets of sound and general equational laws that can facilitate the task of analysing SOC systems through the semantic theories introduced in this section. In fact, as shown in Section 3.3, WS-BPEL applications can be modelled using COWS, thus we hope eventually to be able to verify them. We also plan to develop efficient symbolic characterisations of the labelled bisimilarities over the symbolic operational semantics for COWS introduced in Section 4.4.

One major distinctive feature of COWS is the parallel operator that takes priority of actions into account, where actions have assigned priority values that can dynamically change and have a scope. In Sections 3.2.2 and 3.2.3, we have largely discussed the relevance of this feature to deal with correlation, and service instantiation and termination. Many process calculi with priority have been proposed in the literature; a comprehensive survey, with terminology and classification of different approaches, can be found in [69]. In previous proposals, dynamic priorities are basically used to model scheduling approaches and real-time aspects (see e.g. [24, 42, 85]) while in COWS they are used for coordination, as well as for orchestration, purposes. For example, in the service *5F* of Section 3.1 they enable implementing a sort of 'default' behaviour, that returns *err* when a throw is not admissible. To the best of our knowledge (see also [69]), the interplay between dynamic priorities and local pre-emption, and their impact on semantic theories of processes have never been explored before.

In some calculi for SOC, strong or weak notions of labelled bisimilarity have been used for proving existence of normal forms for services [199], for program transformation [92], for checking conformance of a calculus of orchestration to a calculus of choreography [50], and for proving compliance between service implementations and specifications [124, 45]. The latter two mentioned works share our same aims, but consider process calculi that, to link together actions executed as part of the same interaction, rely on *sessions* rather than on correlation data. Behavioural theories based on testing preorders have instead been exploited to check if a service exposing a given contract can play a specific role within a service choreography [41].

COWS's barbed bisimilarities follow the approach of open barbed bisimilarities [113, 181] rather than that of barbed congruence [152], i.e. quantification over contexts occurs recursively inside the definition of bisimilarity. This approach is commonly used in recent works on defining barbed bisimilarities and, then, their labelled characterizations for process calculi with intricate features as e.g. Distributed π-calculus [96] and KLAIM [76]. This because it simplifies the proof of completeness since it allows to choose a context at any computational step. Moreover, for asynchronous π-calculus, it has been proved in [95] that the two approaches coincide. However, since this is not true for (synchronous) π-calculus [181], and since our calculus is not completely asynchronous, it is probably worth considering also the approach used in [152, 7]. We leave this for future investigation.

COWS's labelled bisimilarities strongly recall the bisimilarities for asynchronous π-calculus introduced in [7]. However, COWS's priority mechanisms make some receive

actions observable (which leads to a novel notion of observation that refines the purely asynchronous one), and require specific conditions on the labelled bisimilarities for these to be congruences.

4.4 A symbolic semantics for COWS

In the previous sections, we have presented a logic and a model checker to express and check functional properties of services specified in COWS, and some observational semantic theories for COWS. To some extent, such tools suffer from a lack of compositionality and efficiency. Indeed, generally speaking, model and equivalence checkers, and other similar verification tools, do not work directly on syntactic specifications but rather on abstract representations of the behaviour of processes. For value-passing languages, such as COWS, using an inappropriate representation can lead to unfeasible verifications. In fact, according to the COWS's original operational semantics introduced in Chapter 3, if the communicable values range over an infinite value set (e.g. natural numbers and strings), the behaviour of a service that performs a receive activity is modelled by an infinite abstract representation. Such representation is a Labelled Transition System whose initial state has infinite outgoing edges, each labelled with an input label having a different value as argument and leading to a different state.

Hence, by taking inspiration from Hennessy and Lin [105], in this section we define a *symbolic* operational semantics for COWS. Differently from the symbolic semantics for more standard calculi, such as value-passing CCS or π-calculus, ours deals at once with, besides receive transitions, a number of complex features, such as, e.g., generation and exportation of fresh names, pattern-matching, expressions evaluation, and priorities among conflicting receives. The new semantics avoids infinite representations of COWS terms due to the value-passing nature of communication in COWS and associates a finite representation to each finite COWS term. It is then more amenable for automatic manipulation by analytical tools, such as e.g. equivalence and model checkers. Our major result is a theorem of 'operational correspondence'. We prove that, under appropriate conditions, any transition of the original semantics can be generated using the symbolic one, and vice versa. In general, however, additional transitions can be derived using the symbolic semantics since it also accounts for services ability to interact with the environment.

4.4.1 A symbolic approach to cope with verification problems

When the considered specification language is a value-passing process algebra and the value-space is infinite, using standard Labelled Transition Systems (LTSs) for the semantics can lead to infinite representations. This is, in fact, the case of COWS. For example, consider a non-persistent (i.e. a non-replicated) variant of the high-level specification of

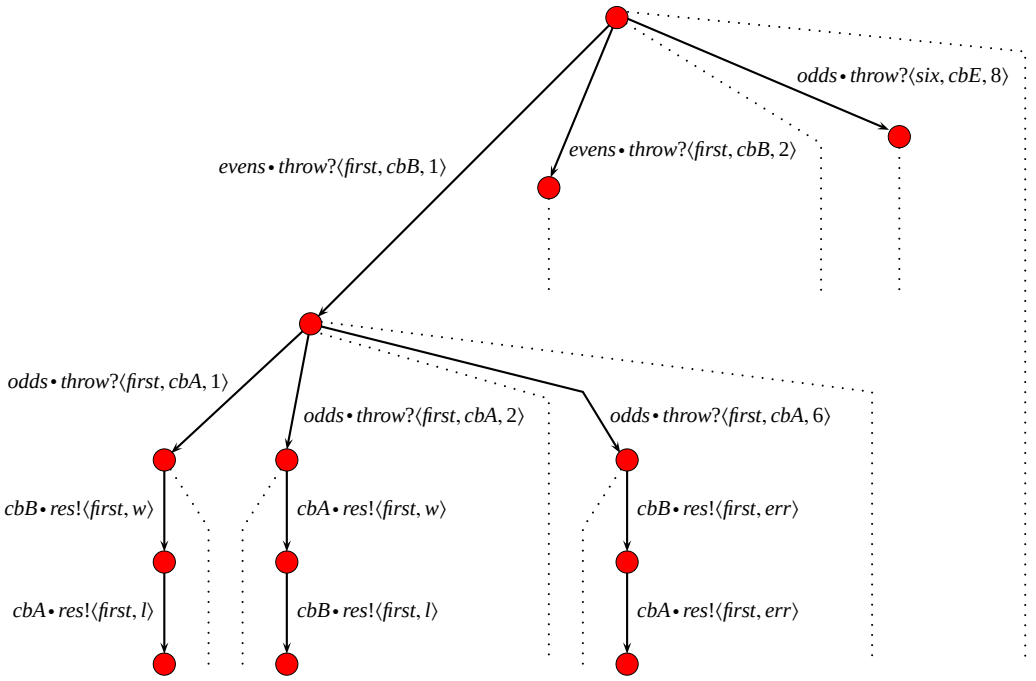

Figure 4.7: LTS for the Morra service (high-level specification)

the Morra service (that is, the COWS term presented in Section 3.1):

$$
\begin{array}{l}
[x_{id}, x_p, x_{num}, y_p, y_{num}] \\
(\, odds \bullet throw?\langle x_{id}, x_p, x_{num}\rangle \mid evens \bullet throw?\langle x_{id}, y_p, y_{num}\rangle \\
\quad \mid x_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 1)\rangle \mid y_p \bullet res!\langle x_{id}, win(x_{num}, y_{num}, 0)\rangle\,)
\end{array}
\tag{4.7}
$$

Its operational behaviour can be represented by the infinite LTS in Figure 4.7, where nodes denote states and edges denote transitions between states. Notably, for the sake of presentation, the LTS shown in the figure relies on an operational semantics in *early* style, where substitutions are applied when receive actions are inferred. However, the problem of infinite representations remains also in case of *late* semantics, due to the fact that the continuation of a receive action with argument a variable x has to be considered under all possible substitutions for x.

To tackle the problems above, in [105] Hennessy and Lin have introduced the so-called *symbolic* LTSs and used them to define finite semantical representations of terms of the value-passing CCS. For example, the symbolic LTSs corresponding to the COWS service (4.7) is shown in Figure 4.8. The symbolic actions $evens \bullet throw?\langle \underline{x}_{id}, \underline{y}_p, \underline{y}_{num}\rangle$ and $odds \bullet throw?\langle \underline{x}_{id}, \underline{x}_p, \underline{x}_{num}\rangle$ denote reception of unknown values $\underline{x}_{id}$, $\underline{y}_p$, $\underline{y}_{num}$, $\underline{x}_p$ and $\underline{x}_{num}$ along endpoints $evens \bullet throw$ and $odds \bullet throw$, respectively; the condition-guarded symbolic actions $(\underline{z}_1 = win(\underline{x}_{num}, \underline{y}_{num}, 1)\,,\ \underline{x}_p \bullet res!\langle \underline{x}_{id}, \underline{z}_1\rangle)$ and $(\underline{z}_2 = win(\underline{x}_{num}, \underline{y}_{num}, 0)\,,\ \underline{y}_p \bullet res!\langle \underline{x}_{id}, \underline{z}_2\rangle)$ denote sending of unknown values $\underline{z}_1$ and $\underline{z}_2$ such that $\underline{z}_1 = win(\underline{x}_{num}, \underline{y}_{num}, 1)$ and $(\underline{z}_2 = win(\underline{x}_{num}, \underline{y}_{num}, 0)$.

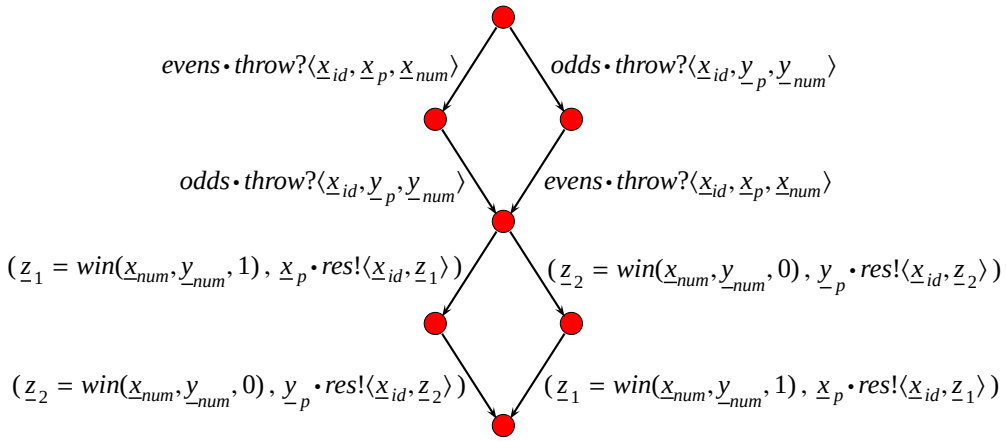

Figure 4.8: Symbolic LTS for the Morra service (high-level specification)

To deal with infinite branching in the representations of COWS terms, we have developed a symbolic operational semantics for COWS. To achieve this goal, the main issue is to give receive activities a proper semantics, because variables in their arguments are placeholders for something to be received. For example, let us consider the service $p \bullet o?\langle x \rangle.s$. If $p \bullet o?\langle x \rangle.s \xrightarrow{p \bullet o \rhd \langle x \rangle} s$ then the behaviour of the continuation service s must be considered under all substitutions of the form $\{x \mapsto v\}$ (i.e. the semantics of s can intuitively be thought of as a function $\lambda x.\, s$ from values to services). In case of the standard semantics for π-calculus [150], for example, this problem is not tackled at the operational semantics level, but it is postponed to the observational semantics level. In fact, in the definition of late bisimulation for π-calculus, whenever P is bisimilar to Q, if $P \xrightarrow{a(x)} P'$ then there is Q' such that $Q \xrightarrow{a(x)} Q'$ and $P'\{u/x\}$ is bisimilar to $Q'\{u/x\}$ for every u. Thus, continuations P' and Q' are considered under all substitutions for x. Instead, here we aim at defining an operational semantics for COWS that properly handles input transitions, and allow finite state LTSs to be associated to finite COWS terms.

The basic idea is to allow receive activities to evolve by performing a communication with the 'external world' (i.e. a COWS context), this way they do not need to synchronise with invoke activities within the considered term. To avoid infinite branching (as in the case of early operational semantics), we replace variables with *unknown values* rather than with specific values. We denote by $\underline{x}$ the unknown value that replaces the variable x. This way, the term $[x]\,(\,p \bullet o?\langle x \rangle.\, q \bullet o'!\langle x \rangle\,)$ can evolve as follows:

$$[x]\,(\,p \bullet o?\langle x \rangle.\, q \bullet o'!\langle x \rangle\,) \xrightarrow{p \bullet o \rhd [x]} q \bullet o'!\langle \underline{x} \rangle \xrightarrow{q \bullet o' \lhd \langle \underline{x} \rangle} \mathbf{0}$$

Also receive activities having a value as argument (e.g. $p \bullet o?\langle v \rangle$) and invoke activities (e.g. $p \bullet o!\langle v \rangle$) can evolve by communicating with the external world. Of course, these kinds of communication do not produce substitutions.

When an external communication takes place, the behaviour of the continuation service depends on the *admittable values* for the unknown value. To take care of the real

Conditions: $\Phi, \Phi', \ldots$	*Exported private names:* $\Delta, \Delta', \ldots$
Unknown values: $\underline{x}, \underline{y}, \ldots$	*Variable/Names/Unknown values:* $\underline{u}, \underline{u}', \ldots$
Values/Unknown values: $\underline{v}, \underline{v}', \ldots$	*Variable/Values/Unknown values:* $\underline{w}, \underline{w}', \ldots$
Names/Unknown values: $\underline{n}, \underline{m}', \ldots, \underline{p}, \underline{o}, \ldots$	*Endpoints/Unknown values:* $\underline{\mathtt{n}}, \underline{\mathtt{m}}', \ldots, \underline{\mathtt{u}}, \underline{\mathtt{u}}', \ldots$

Constrained services: $\Phi, \Delta \vdash s$

Services: $s ::= \mathbf{kill}(k) \mid \underline{u} \bullet \underline{u}'!\epsilon \mid g \mid s \mid s \mid \{\!|s|\!\} \mid [e]\, s \mid * s$

Receive-guarded choice: $g ::= \mathbf{0} \mid p \bullet o?\underline{w}.s \mid g + g$

Table 4.11: Constrained services

values that the unknown values can assume, we define a *symbolic semantics* for COWS, where the label on each transition has two components: the *condition* that must hold for the transition to be enabled and, as usual, the *action* of the transition. Moreover, to store the conditions that must hold to reach a state and the names exported along the path, we define the semantics over configurations of the form $\Phi, \Delta \vdash s$, called *constrained services*, where the condition Φ and the set of names Δ are used to determine the actions that s can perform. Thus, the symbolic transitions are of the form $\Phi, \Delta \vdash s_1 \overset{\Phi', \alpha}{\rightarrowtail} \Phi', \Delta' \vdash s_2$, meaning "if the condition Φ' (such that Φ is a subterm of Φ') holds then s_1 can perform the action α leading to s_2 by extending the set of exported private names Δ to the set Δ'".

The symbolic LTS associated to a COWS term conveys in a distilled form all the semantics information on the behaviour of terms. More specifically, besides receive transitions, symbolic representations take into account generation and exportation of fresh names, pattern-matching, expressions evaluation, and priorities among conflicting receives. Dealing at once with all the above features at operational semantics level makes the development of a symbolic semantics for COWS more complex than for more standard calculi, such as value-passing CCS or π-calculus.

4.4.2 A symbolic operational semantics for COWS

We apply now the symbolic approach outlined in the previous section to COWS. For the sake of simplicity, here we consider a *monadic* version of COWS, i.e. communication activities are of the form $\mathtt{u}!\epsilon$ and $\mathtt{n}?w.s$ (we discuss in Section 4.4.4.2 how to tailor the symbolic semantics to handle polyadic communication).

The symbolic operational semantics of COWS is defined over configurations of the form $\Phi, \Delta \vdash s$, called *constrained services* and defined in Table 4.11, where Φ is the *condition* that must hold to reach the current state, Δ is the *set of private names* previously exported, and s is a service whose actions are determined by Φ and Δ. The set Δ will be omitted when empty, writing e.g. $\Phi \vdash s$ instead of $\Phi, \emptyset \vdash s$. We define the semantics over an enriched set of services that also includes those auxiliary terms resulting from replacing (free occurrences of) variables with *unknown values* in terms produced by the syntax introduced in Section 3.2.3, where now expressions contain also unknown values.

In the extended syntax we use $\underline{x}$ to denote an unknown value and $\underline{t}$ to denote an unknown value or a term t (where t can be n, v, u, w, $\mathfrak{n}$ or $\mathfrak{u}$). Therefore, $\underline{u} \bullet \underline{u'}!\epsilon$ and $p \bullet o?\underline{w}.s$ denote invoke and receive activities, respectively.

As in the standard semantics, the only *binding* construct is delimitation: let $\Phi, \Delta \vdash \mathbb{C}[\![[d]\, s]\!]$ be a constrained service (where $\mathbb{C}$ is a context[2]), $[d]$ binds d in the scope s, in the condition Φ and in the set Δ. We denote by $\mathrm{bn}(t)$ the set of names that occur bound in a term t, and by $\mathrm{uvar}(t)$ the set of variables that have been replaced by corresponding unknown values in t (i.e. if $\underline{x}$ is an unknown value in t, then $x \in \mathrm{uvar}(t)$). For simplicity sake, in the sequel we assume that bound variables in constrained services are pairwise distinct and different from variables corresponding to the unknown values of the constrained services, and bound names are all distinct and different from the free ones (of course, these conditions are not restrictive and can always be fulfilled by possibly using α-conversion). This assumption avoids that distinct unknown values are denoted by the same $\underline{x}$ in a condition Φ of a constrained service (see Example "Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables" in Section 4.4.3), and permits identifying the name delimitation binding each private name within a condition Φ and a set Δ of a constrained service (see Remark 4.4.1).

The symbolic operational semantics of COWS is defined only for closed services, and is given in terms of a structural congruence and of a (bi-)labelled transition relation. The structural congruence $\equiv$ is the trivial extension of that defined in Section 3.2.3 to the enriched syntax of services used here. To define the labelled transition relation, we exploit the trivial extension to the enriched syntax of function $\mathit{halt}(\_)$ and predicate $\mathrm{noKill}(\_, \_)$ defined in Section 3.2.3. We also extend function $[\![\_]\!]$ to deal with unknown values. Now, it takes a closed expression and returns a pair $(\Phi, \underline{v})$: the (possibly unknown) value $\underline{v}$ is the result of the evaluation provided that the condition Φ holds. Specifically, let ϵ be an expression, if ϵ does not contain unknown values and can be computed, then $[\![\epsilon]\!] = (\mathit{true}, v)$ where v is the result of the evaluation, as in the original COWS semantics. Similarly, if ϵ is an unknown value $\underline{x}$, then $[\![\epsilon]\!] = (\mathit{true}, \underline{x})$. If ϵ contains unknown values and is not a single unknown value (i.e. $\epsilon \neq \underline{x}$ for every $\underline{x}$), then $[\![\epsilon]\!] = ((\underline{y} \neq \mathbf{bn} \wedge y \notin \mathbf{uv} \wedge \underline{y} = \epsilon \wedge \Phi'), \underline{y})$ where $\underline{y}$ is a fresh unknown value that must be different from all private names (i.e. $\underline{y} \neq \mathbf{bn}$) and from all existent unknown values (i.e. $y \notin \mathbf{uv}$)[3], and Φ' is a condition that permits dealing with expression operators partially defined[4]. Function $[\![\_]\!]$, and hence condition Φ', cannot be explicitly defined because the exact syntax of expressions is deliberately not specified. Then, consider as an example the following

[2]Recall that a context $\mathbb{C}$ is a service with a 'hole' $[\![\cdot]\!]$ such that, once the hole is filled with a service s, the resulting term $\mathbb{C}[\![s]\!]$ is a COWS service.

[3]Notably, here $\underline{y}$ can be any unknown value, provide that it satisfies conditions $\underline{y} \neq \mathbf{bn}$ and $y \notin \mathbf{uv}$. Notice that condition $y \notin \mathbf{uv}$ is a syntactical condition on the variable name y. Later we shall explain the exact meaning of the above conditions and show how they are evaluated in the last step of the inference of a transition.

[4]Of course, if all operators used in the considered expression are total functions, then condition Φ' is *true*.

simple language for expressions:

$$\epsilon \;::=\; x \;\mid\; \underline{x} \;\mid\; i \;\mid\; \epsilon + \epsilon \;\mid\; \epsilon - \epsilon \;\mid\; \epsilon * \epsilon \;\mid\; \epsilon / \epsilon \;\mid\; (\epsilon)$$

where i is an integer value. For the above language function $[\![\_]\!]$ is such that:

- $[\![(5-2)*3]\!] = (\mathit{true}, 9)$;
- $[\![5 - x]\!]$ is undefined, because the expression $5 - x$ is not closed;
- $[\![5 - \underline{x}]\!] = ((\underline{y} \neq \mathbf{bn} \;\wedge\; y \notin \mathbf{uv} \;\wedge\; \underline{y} = 5 - \underline{x}), \underline{y})$;
- $[\![5/0]\!]$ is undefined;
- $[\![5/\underline{x}]\!] = ((\underline{y} \neq \mathbf{bn} \;\wedge\; y \notin \mathbf{uv} \;\wedge\; \underline{y} = 5/\underline{x} \;\wedge\; \underline{x} \neq 0), \underline{y})$, where condition $\underline{x} \neq 0$ is due to the fact that operator / is not defined when its second argument is 0.

We also define a function confRec(_, _), that takes a service s and an endpoint $\mathtt{n}$ as an arguments and returns the set of (possibly unknown) values that are parameters of receive activities over the endpoint $\mathtt{n}$ active in s. This function plays the role of predicate noConf(_, _, _, _) of the standard semantics and, indeed, is exploited to disable transitions in case of communication conflicts (by setting transition conditions to *false*). The function is inductively defined as follows:

$$\mathrm{confRec}(\mathbf{0}, \mathtt{n}) = \mathrm{confRec}(\mathbf{kill}(k), \mathtt{n}) = \mathrm{confRec}(\underline{\mathtt{u}}!\epsilon, \mathtt{n}) = \mathrm{confRec}(\mathtt{n}?x.s, \mathtt{n}) = \emptyset$$

$$\mathrm{confRec}(g + g', \mathtt{n}) = \mathrm{confRec}(g, \mathtt{n}) \cup \mathrm{confRec}(g', \mathtt{n}) \qquad \mathrm{confRec}(\mathtt{n}?\underline{v}.s, \mathtt{n}) = \{\underline{v}\}$$

$$\mathrm{confRec}(\mathtt{n}'?\underline{w}.s, \mathtt{n}) = \emptyset \;\text{ if } \mathtt{n} \neq \mathtt{n}' \qquad \mathrm{confRec}(\{\!|s|\!\}, \mathtt{n}) = \mathrm{confRec}(s, \mathtt{n})$$

$$\mathrm{confRec}(s \mid s', \mathtt{n}) = \mathrm{confRec}(s, \mathtt{n}) \cup \mathrm{confRec}(s', \mathtt{n}) \qquad \mathrm{confRec}([e]\, s, \mathtt{n}) = \emptyset \;\text{ if } e \in \mathtt{n}$$

$$\mathrm{confRec}([e]\, s, \mathtt{n}) = \mathrm{confRec}(s, \mathtt{n}) \backslash \{e\} \;\text{ if } e \notin \mathtt{n} \qquad \mathrm{confRec}(* s, \mathtt{n}) = \mathrm{confRec}(s, \mathtt{n})$$

The labelled transition relation over constrained services, written $\overset{\Phi,\alpha}{\rightarrowtail}$, relies on a labelled transition relation $\overset{\Phi,\alpha}{\longrightarrow}$, that is the least relation over services induced by the rules in Table 4.12. Conditions Φ and actions α are generated by the following grammar:

$$\begin{aligned}\Phi \;::=\;& \mathit{true} \;\mid\; \mathit{false} \;\mid\; \underline{v} = \underline{v}' \;\mid\; \underline{v} \neq \underline{v}' \;\mid\; \underline{x} \neq \mathbf{bn} \;\mid\; x \notin \mathbf{uv} \\ &\mid\; x \notin \{x_i\}_{i \in I} \;\mid\; \underline{x} = \epsilon \;\mid\; \Phi \wedge \Phi'\end{aligned}$$

$$\alpha \;::=\; \mathtt{n} \lhd \underline{v} \;\mid\; \mathtt{n} \lhd [n] \;\mid\; \mathtt{n} \rhd \underline{w} \;\mid\; \mathtt{n} \rhd [x] \;\mid\; \mathtt{n}\, \sigma\, \ell\, \underline{v} \;\mid\; k \;\mid\; \dagger$$

where, now, a substitutions σ can be either the empty substitution $\emptyset$ or a substitution $\{x \mapsto \underline{v}\}$ that maps the variable x to the (possibly unknown) value $\underline{v}$.

The meaning of labels is as follows:

$$\mathbf{kill}(k) \xrightarrow{true\,,k} \mathbf{0} \quad \textit{(s-kill)} \qquad \mathtt{n}?\underline{w}.s \xrightarrow{true\,,\mathtt{n}\rhd\underline{w}} s \quad \textit{(s-rec)}$$

$$\frac{s \xrightarrow{\Phi\,,\mathtt{n}\rhd x} s'}{[x]\,s \xrightarrow{\Phi\wedge \underline{x}\neq\mathbf{bn}\wedge \underline{x}\neq \mathrm{confRec}(s,\mathtt{n})\,,\mathtt{n}\rhd[x]} s'\cdot\{x\mapsto \underline{x}\}} \quad \textit{(s-rec}_{com}\textit{)} \qquad \frac{g \xrightarrow{\Phi\,,\alpha} s}{g+g' \xrightarrow{\Phi\,,\alpha} s} \quad \textit{(s-choice)}$$

$$\frac{[\![\epsilon]\!] = (\Phi, \underline{v})}{\underline{\mathtt{n}}!\epsilon \xrightarrow{\Phi\,,\underline{\mathtt{n}}\lhd \underline{v}} \mathbf{0}} \quad \textit{(s-inv)} \qquad \frac{s \xrightarrow{\Phi\,,\underline{\mathtt{n}}\lhd n} s' \quad n\notin\underline{\mathtt{n}}}{[n]\,s \xrightarrow{\Phi\,,\underline{\mathtt{n}}\lhd[n]} s'} \quad \textit{(s-open)}$$

$$\frac{s \xrightarrow{\Phi\,,\mathtt{n}\{x\mapsto\underline{v}\}\,!\,\underline{v}} s'}{[x]\,s \xrightarrow{\Phi\,,\mathtt{n}\emptyset\,!\,\underline{v}} s'\cdot\{x\mapsto\underline{v}\}} \quad \textit{(s-del}_{com}\textit{)} \qquad \frac{s \xrightarrow{\Phi\,,k} s'}{[k]\,s \xrightarrow{\Phi\,,\dagger} [k]\,s'} \quad \textit{(s-del}_{kill1}\textit{)}$$

$$\frac{s \xrightarrow{\Phi\,,k} s' \quad k\neq e}{[e]\,s \xrightarrow{\Phi\,,k} [e]\,s'} \quad \textit{(s-del}_{kill2}\textit{)} \qquad \frac{s \xrightarrow{\Phi\,,\dagger} s'}{[e]\,s \xrightarrow{\Phi\,,\dagger} [e]\,s'} \quad \textit{(s-del}_{kill3}\textit{)}$$

$$\frac{s \xrightarrow{\Phi\,,\alpha} s' \quad e\notin \mathrm{e}(\alpha) \quad \alpha\neq k\,,\dagger \quad \mathrm{noKill}(s,e)}{[e]\,s \xrightarrow{\Phi\,,\alpha} [e]\,s'} \quad \textit{(s-del)} \qquad \frac{s \xrightarrow{\Phi\,,\alpha} s'}{\{\!|s|\!\} \xrightarrow{\Phi\,,\alpha} \{\!|s'|\!\}} \quad \textit{(s-prot)}$$

$$\frac{s_1 \xrightarrow{\Phi_1\,,\mathtt{n}\rhd\underline{v}'} s_1' \qquad s_2 \xrightarrow{\Phi_2\,,\underline{\mathtt{n}}'\lhd\underline{v}} s_2'}{s_1 \mid s_2 \xrightarrow{\Phi_1\wedge\Phi_2\wedge\mathtt{n}=\underline{\mathtt{n}}'\wedge\underline{v}'=\underline{v}\,,\mathtt{n}\emptyset 0\,\underline{v}} s_1' \mid s_2'} \quad \textit{(s-match)}$$

$$\frac{s_1 \xrightarrow{\Phi_1\,,\mathtt{n}\rhd x} s_1' \qquad s_2 \xrightarrow{\Phi_2\,,\underline{\mathtt{n}}'\lhd\underline{v}} s_2'}{s_1 \mid s_2 \xrightarrow{\Phi_1\wedge\Phi_2\wedge\mathtt{n}=\underline{\mathtt{n}}'\wedge\underline{v}\neq\mathrm{confRec}(s_1\mid s_2,\mathtt{n})\,,\mathtt{n}\{x\mapsto\underline{v}\}\,!\,\underline{v}} s_1' \mid s_2'} \quad \textit{(s-com)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\mathtt{n}\sigma\,!\,\underline{v}} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\underline{v}\neq\mathrm{confRec}(s_2,\mathtt{n})\,,\mathtt{n}\sigma\,!\,\underline{v}} s_1' \mid s_2} \quad \textit{(s-par}_{com1}\textit{)} \qquad \frac{s_1 \xrightarrow{\Phi\,,k} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi\,,k} s_1' \mid \mathit{halt}(s_2)} \quad \textit{(s-par}_{kill}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\mathtt{n}\rhd[x]} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\underline{x}\neq\mathrm{confRec}(s_2,\mathtt{n})\,,\mathtt{n}\rhd[x]} s_1' \mid s_2} \quad \textit{(s-par}_{com2}\textit{)} \qquad \frac{s\equiv s_1 \quad s_1 \xrightarrow{\Phi\,,\alpha} s_2 \quad s_2\equiv s'}{s \xrightarrow{\Phi\,,\alpha} s'} \quad \textit{(s-cong)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\alpha} s_1' \qquad \alpha\neq k\,,\,\mathtt{n}\sigma\,!\,\underline{v}\,,\,\mathtt{n}\rhd[x]}{s_1 \mid s_2 \xrightarrow{\Phi\,,\alpha} s_1' \mid s_2} \quad \textit{(s-par)}$$

Table 4.12: COWS symbolic semantics (rules for $\xrightarrow{\Phi\,,\alpha}$)

- *Conditions*: *true* (resp. *false*) denotes the condition always (resp. never) satisfied, $\underline{v} = \underline{v}'$ (resp. $\underline{v} \neq \underline{v}'$) denotes an equality (resp. inequality) between (possibly unknown) values, $\underline{x} \neq \mathbf{bn}$ means that the unknown value $\underline{x}$ must be different from all bound names of the considered service, $x \notin \mathbf{uv}$ means that the set of variables corresponding to the unknown values of the considered constrained service may not contain the variable x, $x \notin \{x_i\}_{i\in I}$ means that x must not be in the set $\{x_i\}_{i\in I}$, $\underline{x} = \epsilon$ states that the unknown value $\underline{x}$ is equal to the evaluation of the closed non-evaluable expression ϵ (conditions of this form are generated by the evaluation function, e.g. condition $\underline{y} = 5/\underline{x}$ is generated by evaluation of expression $5/\underline{x}$), and as usual $\wedge$ denotes the logic conjunction. In the sequel, we will use notation $\underline{v} \neq \{\underline{v}_1, \ldots, \underline{v}_n\}$ to indicate the condition $\underline{v} \neq \underline{v}_1 \wedge \ldots \wedge \underline{v} \neq \underline{v}_n$ (where $\underline{v} \neq \emptyset$ indicates *true*). Moreover, we will use a function $\mathcal{B}(\_,\_,\_)$ that, given a condition Φ, a service s and a set of variables $\{x_i\}_{i\in I}$, returns a condition obtained by conjuncting Φ with all inequalities between the unknown values of Φ and the bound names of s and with all conditions $x \notin \{x_i\}_{i\in I}$ for each $x \notin \mathbf{uv}$ in Φ. Formally, $\mathcal{B}(\_,\_,\_)$ is defined as follows:

$$\mathcal{B}(\mathit{true}, s, \{x_i\}_{i\in I}) = \mathit{true} \qquad \mathcal{B}(\mathit{false}, s, \{x_i\}_{i\in I}) = \mathit{false}$$
$$\mathcal{B}(\underline{v} = \underline{v}', s, \{x_i\}_{i\in I}) = \underline{v} = \underline{v}' \qquad \mathcal{B}(\underline{v} \neq \underline{v}', s, \{x_i\}_{i\in I}) = \underline{v} \neq \underline{v}'$$
$$\mathcal{B}(\underline{x} \neq \mathbf{bn}, s, \{x_i\}_{i\in I}) = \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq \mathrm{bn}(s) \qquad \mathcal{B}(x \notin \mathbf{uv}, s, \{x_i\}_{i\in I}) = x \notin \{x_i\}_{i\in I}$$
$$\mathcal{B}(x \notin \{y_j\}_{j\in J}, s, \{x_i\}_{i\in I}) = x \notin \{y_j\}_{j\in J} \qquad \mathcal{B}(\underline{x} = \epsilon, s, \{x_i\}_{i\in I}) = \underline{x} = \epsilon$$
$$\mathcal{B}(\Phi \wedge \Phi', s, \{x_i\}_{i\in I}) = \mathcal{B}(\Phi, s, \{x_i\}_{i\in I}) \wedge \mathcal{B}(\Phi', s, \{x_i\}_{i\in I})$$

- *Actions*: $\underline{\mathtt{n}} \lhd [n]$ denotes execution of a bound invoke activity over the endpoint $\underline{\mathtt{n}}$, while $\mathtt{n} \rhd [x]$ denotes taking place of external communication over the endpoint $\mathtt{n}$ with receive parameter x (that will be replaced by the unknown value $\underline{x}$). The remaining labels have the usual meaning. Notably, due to the restraint on monadic communication, here the natural number ℓ can only be either 0 or 1.

We comment on the aspects of the symbolic semantics rules that mainly differ from the standard ones. Bound invocations, that transmit private names, can be generated by rule *(s-open)*. Notably, bound invocation actions do not appear in rules *(s-match)* and *(s-com)*, and therefore cannot directly interact with receive actions. Such interactions are instead inferred by using structural congruence to pull name delimitation outside both interacting activities. Although the bound transitions and rule *(s-open)* can be omitted, we include them both to give a proper semantics to terms $[m]\mathtt{n}!m$ and to support the development of behavioural equivalences for COWS. Communication can be either *internal* or *external* to a service. Internal communication can take place when two matching receive and invoke activities (rules *(s-match)* and *(s-com)*) are simultaneously executed. External communication can take place when a value is transmitted to the environment (rules *(s-inv)* and *(s-open)*) or when a receive activity matches an unknown value provided by the environment (rules *(s-rec)* and *(s-rec$_{com}$)*). Differently from the standard semantics, conflicting

$$\frac{s \xrightarrow{\Phi', \alpha} s' \qquad \alpha \neq \underline{\mathtt{n}} \lhd [n] \ , \ \underline{\mathtt{n}} \lhd \underline{v} \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \alpha} \Phi'', \Delta \vdash s'} \ (\textit{constServ})$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd [n]} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd [n]} \Phi'', \Delta \cup \{n\} \vdash s'} \ (\textit{constServ}_{exp})$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd \underline{v}} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd \underline{v}} \Phi'', \Delta \vdash s'} \ (\textit{constServ}_{inv})$$

Table 4.13: COWS symbolic semantics (rules for $\succ\!\!\xrightarrow{\Phi, \alpha}$)

receives cannot be dealt with by using a predicate in the premises of rules for communication and interleaving, because unknown values can be involved. Here, the check for conflicting receives is simply a condition of the form $\underline{v} \neq \mathrm{confRec}(s, \mathtt{n})$ (rules *(s-rec$_{com}$)*, *(s-com)*, *(s-par$_{com1}$)* and *(s-par$_{com2}$)*).

The labelled transition relation $\succ\!\!\xrightarrow{\Phi, \alpha}$ is the least relation over constrained services induced by the rules reported in Table 4.13, where notation $\underline{\mathtt{n}} \notin \Delta$ means that set Δ does not contain the names of endpoint $\underline{\mathtt{n}}$. Rule *(constServ)* states that a constrained service $\Phi, \Delta \vdash s$ can perform all the 'non-invoke' transitions performed by s with an enriched condition Φ'' obtained by composing Φ and the condition on the label Φ'. Condition Φ'' takes care of the relationship between unknown values and private names. Indeed, by private names definition, each unknown value, that is a value coming from the environment, must be different from all bound (private) names of the considered service. If the transition $s \xrightarrow{\Phi', \alpha} s'$ introduces a new unknown value $\underline{x}$ (rules *(s-inv)* and *(s-rec$_{com}$)*), it is not sufficient to add the condition $\underline{x} \neq \mathrm{bn}(s')$ (i.e. the unknown value is different from all bound names of the current service), but we need also to consider bound names that could be subsequently generated. For example, let us consider the following transition:

$$\mathit{true} \vdash [x]\,\mathtt{n}?x.s \mid *\,[n]\,\mathtt{n}'!n \succ\!\!\xrightarrow{\underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq n \,,\, \mathtt{n} \rhd [x]} \underline{x} \neq \mathbf{bn} \ \wedge \ \underline{x} \neq n \vdash s \cdot \{x \mapsto \underline{x}\} \mid *\,[n]\,\mathtt{n}'!n$$

Now, if the obtained service performs the transition:

$$s \cdot \{x \mapsto \underline{x}\} \mid *\,[n]\,\mathtt{n}'!n \xrightarrow{\Phi, \alpha} s' \cdot \{x \mapsto \underline{x}\} \mid *\,[n]\,\mathtt{n}'!n \mid [n']\,\mathtt{n}'!n' \mid [n'']\,\mathtt{n}'!n''$$

then, let $\Phi' \vdash s''$ be the obtained constrained service, the condition Φ' must contain $\underline{x} \neq n'$ and $\underline{x} \neq n''$. To update after any transition the condition of a constrained service with inequalities between unknown values and private names, we use the condition $\underline{x} \neq \mathbf{bn}$, that simply states that $\underline{x}$ has been introduced in the considered term (rules *(s-inv)* and *(s-rec$_{com}$)*),

and function $\mathcal{B}(\_,\_,\_)$, that adds the inequalities for each unknown value (rules *(constServ)*, *(constServ$_{exp}$)* and *(constServ$_{inv}$)*). Moreover, function $\mathcal{B}(\_,\_,\_)$ adds conditions of the form $x \notin \{x_i\}_{i\in I}$ to guarantee that unknown values introduced by rule *(s-inv)* because of expression evaluation differ from those of the considered constrained service (i.e. uvar(Φ) if the constrained service is $\Phi, \Delta \vdash s$; for further details see Example "Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables" in Section 4.4.3).

Rules *(constServ$_{exp}$)* and *(constServ$_{inv}$)* deal with the localized receiving feature of COWS. Indeed, if a COWS term communicates a private (partner or operation) name to the environment, then the latter (that is a COWS context) may use the name to define a sending endpoint, but not a receiving one. For example, consider the following constrained service:

$$true \vdash [p]\,(\,q \bullet o!p \;\mid\; p \bullet o'!v\,)$$

It can perform the activity $q \bullet o!p$ (rule *(s-open)*) and become the term $true, \{p\} \vdash p \bullet o'!v$ which is stuck. In fact, to further evolve it needs the environment to be able to perform first a receive $q \bullet o?x$ and then a receive along the endpoint $x \bullet o'$, that is disallowed by the syntax. Therefore, to block invoke activities performed along endpoints using previously exported private names, we record all exported private names in the set Δ of the constrained service and perform the check $\underline{\mathtt{n}} \notin \Delta$ when an invoke activity along $\underline{\mathtt{n}}$ communicating with the environment is executed.

Remark 4.4.1 The assumption "bound names are all distinct and different from the free ones" is used to guarantee the correlation between conditions and services. For example, if we do not rely on this assumption, for the constrained service $\underline{x} \neq n \vdash [n]\, s \mid [n]\, s'$ we are not able to understand what is the binder for the n in the condition $\underline{x} \neq n$. In fact, by definition of bound names, the constrained service $\underline{x} \neq n \vdash [n]\, \underline{x} \bullet o!n$ is α-equivalent to $\underline{x} \neq m \vdash [m]\, \underline{x} \bullet o!m$, not to $\underline{x} \neq n \vdash [m]\, \underline{x} \bullet o!m$.

Remark 4.4.2 In the definition of relation $\overset{\Phi,\alpha}{\rightarrowtail}$, the conditions are never evaluated. Thus, at operational semantics level, we do not distinguish unfeasible transitions (whose condition holds *false*) from feasible ones. For example, transitions having the following conditions are unfeasible: $(o_{req} = o_{resp}), (\underline{x} \neq \underline{x})$ and $(\underline{x} = \underline{y} \wedge \underline{x} \neq \underline{y})$. Of course, to identify unfeasible transitions, we can replace the condition Φ'' in the conclusion of rules *(constServ)*, *(constServ$_{exp}$)* and *(constServ$_{inv}$)* with $\mathcal{E}(\Phi'')$, where $\mathcal{E}(\_)$ is a function for evaluating conditions.

Remark 4.4.3 Since the transition relation $\overset{\Phi,\alpha}{\rightarrowtail}$ is defined over constrained services, i.e. configuration of the form $\Phi, \Delta \vdash s$, the operational semantics can be naturally interpreted on L^{2}TS [80]. Indeed, each *edge label* (of the form Φ, α) indicates the condition which must hold for the transition to be enabled and the performed action, while each *state label* (of the form Φ, Δ) indicates the condition which must hold to reach the considered state from the initial one and the set of previously exported private names.

We can now formalize the correspondence between the original semantics introduced in Section 3.2.3 and the symbolic semantics. We exploit here a function $\mathcal{E}(\_)$ for evaluating conditions: it takes a condition Φ and returns *false* if certainly Φ does not hold; otherwise, it returns Φ. For example, $\mathcal{E}(\Phi' \wedge (5 = 3))$ is *false* whatever Φ' may be. Since a condition Φ can be of the form $\underline{x} = \epsilon$ and the syntax of expressions ϵ is not specified, function $\mathcal{E}(\_)$ cannot be explicitly defined (like function $[\![\_]\!]$). For the proof of semantics correspondence, we use the following lemma concerning function $\mathcal{B}(\_, \_, \_)$. For the sake of simplicity, a condition Φ is deemed *favourable* if $\mathrm{uvar}(\Phi) = \emptyset$ and $\mathcal{E}(\Phi) \neq \mathit{false}$, i.e. it does not contain unknown values and can be positively evaluated.

Lemma 4.4.1 *Let Φ be a favourable condition, then $\mathcal{E}(\mathcal{B}(\Phi, s, \emptyset)) \neq \mathit{false}$ for any s.*

Proof. The thesis follows from the fact that, under the considered hypotheses, $\mathcal{B}(\_, \_, \_)$ acts as an homomorphism on the first argument, except when the argument is $\underline{x} \neq \mathbf{bn}$ and $x \notin \mathbf{uv}$. We do not need to consider the former case because, by the hypothesis $\mathrm{uvar}(\Phi) = \emptyset$, we have that Φ does not contain unknown values. For the latter case, we get that $\mathcal{B}(x \notin \mathbf{uv}, s, \emptyset) = x \notin \mathbf{uv}$ since the third argument of $\mathcal{B}(\_, \_, \_)$ is $\emptyset$. □

Our major result is a theorem of 'operational correspondence'. It is quite standard and states that for each transition of the original LTS associated to a COWS term there exists a corresponding symbolic transition of the symbolic LTS that does not involve unknown values and bound names, and vice versa. Notice that, since the original semantics does not take bound invocations into account, only constrained services of the form $\Phi \vdash s$ are considered in the theorem. For the sake of readability, we only outline here the techniques used in the proof of the theorem and refer the interested reader to Appendix B.3 for a full account.

Theorem 4.4.1 *Let* $\mathrm{uvar}(\alpha) = \emptyset$ *and* $\alpha \neq \mathtt{n} \lhd [n]$. $s \xrightarrow{\alpha} s'$ *if and only if, for any favourable condition* Φ, $\Phi \vdash s \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$ *for some favourable condition* Φ'.

Proof (sketch). The proof of the "if" part proceeds by induction on the length of the inference of $s \xrightarrow{\alpha} s'$. The "only if" part of the theorem proceeds as follows. By the premises of rules *(constServ)* and *(constServ$_{inv}$)*, we get that $s \xrightarrow{\Phi'', \alpha} s'$ where $\Phi' = \mathcal{B}(\Phi \wedge \Phi'', s', \emptyset)$. By hypothesis $\mathcal{E}(\Phi') \neq \mathit{false}$, hence $\mathcal{E}(\Phi'') \neq \mathit{false}$ too. Then, the proof proceeds by induction on the length of the inference of $s \xrightarrow{\Phi'', \alpha} s'$. □

4.4.3 Examples

To shed light on the technical development of COWS symbolic semantics, we show some simple illustrative examples. In the sequel, for the sake of readability, we shall evaluate conditions, writing e.g. $\underline{x} \neq n$ instead of $(p = p \wedge o = o \wedge \mathit{true} \wedge \underline{x} \neq n)$.

External communication. According to the operational semantics introduced in Section 3.2.3, the service $[x]\,\mathtt{n}?x.\,\mathtt{m}!x$ is blocked (because it cannot perform the receive activity). Instead, according to the symbolic semantics defined in this section, the constrained service $\mathit{true} \vdash [x]\,\mathtt{n}?x.\,\mathtt{m}!x$ can evolve as follows:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{n}?x.\,\mathtt{m}!x \xrightarrow{\mathit{true}\,,\,\mathtt{n}\rhd x} \mathtt{m}!x}\ (\textit{s-rec})}{[x]\,\mathtt{n}?x.\,\mathtt{m}!x \xrightarrow{\underline{x}\neq \mathrm{confRec}((\mathtt{n}?x.\,\mathtt{m}!x),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \mathtt{m}!\underline{x}}\ (\textit{s-rec}_{com})}{\mathit{true} \vdash [x]\,\mathtt{n}?x.\,\mathtt{m}!x \succ \xrightarrow{\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\neq\mathbf{bn} \vdash \mathtt{m}!\underline{x}}\ (\textit{constServ})$$

since $(\underline{x} \neq \mathrm{confRec}((\mathtt{n}?x.\,\mathtt{m}!x),\mathtt{n})) = (\underline{x} \neq \emptyset) = \mathit{true}$. Then, the continuation can perform the following transition:

$$\dfrac{\dfrac{\dfrac{[\![\underline{x}]\!] = (\mathit{true},\underline{x})}{\mathtt{m}!\underline{x} \xrightarrow{\mathit{true}\,,\,\mathtt{m}\lhd\underline{x}} \mathbf{0}}\ (\textit{s-inv})}{\underline{x}\neq\mathbf{bn} \vdash \mathtt{m}!\underline{x} \succ \xrightarrow{\underline{x}\neq\mathbf{bn}\,,\,\mathtt{m}\lhd\underline{x}} \underline{x}\neq\mathbf{bn} \vdash \mathbf{0}}\ (\textit{constServ}_{inv})$$

Notice that, although the external communication generates the condition $\underline{x} \neq \mathbf{bn}$ (that means that the received unknown value must be different from all delimited names), the condition is never exploited because the term does not contain delimited names.

External communication within name delimitations. Consider the constrained service $\mathit{true} \vdash [n]\,[x]\,\mathtt{n}?x.\,x \bullet o!n$. Differently from the previous example, the above service contains a delimited name (i.e. n). Thus, this time, condition $\underline{x} \neq \mathbf{bn}$ is exploited to generate the specific condition $\underline{x} \neq n$. Indeed, the service evolves as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{n}?x.\,x\bullet o!n \xrightarrow{\mathit{true}\,,\,\mathtt{n}\rhd x} x\bullet o!n}\ (\textit{s-rec})}{[x]\,\mathtt{n}?x.\,x\bullet o!n \xrightarrow{\underline{x}\neq\mathrm{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\bullet o!n}\ (\textit{s-rec}_{com})}{[n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n \xrightarrow{\underline{x}\neq\mathrm{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} [n]\,\underline{x}\bullet o!n}\ (\textit{s-del}_{pass})}{\mathit{true} \vdash [n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n \succ \xrightarrow{\underline{x}\neq n\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\neq n\,\wedge\,\underline{x}\neq\mathbf{bn} \vdash [n]\,\underline{x}\bullet o!n}\ (\textit{constServ})$$

since $(\underline{x} \neq \mathrm{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n}) = \mathit{true}$ and $\mathcal{B}(\underline{x}\neq\mathbf{bn},([n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n),\emptyset) = \underline{x}\neq n\,\wedge\,\underline{x}\neq\mathbf{bn}$. Then, the continuation can evolve only provided that condition $\underline{x} \neq n$ holds.

Internal communication. Consider the constrained service $\mathit{true} \vdash [p]\,[x]\,(\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,)$, where $p \notin \mathtt{n}$. In this case, due to the delimitation $[p]$, the receive activity cannot

communicate with the environment, but can synchronize with the internal invoke:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p\bullet o?x.\,\mathtt{n}!x \xrightarrow{true\,,\,p\bullet o\rhd x} \mathtt{n}!x}\,(s\text{-}rec) \quad \dfrac{[\![v]\!] = (true, v)}{p\bullet o!v \xrightarrow{true\,,\,p\bullet o\lhd v} \mathbf{0}}\,(s\text{-}inv)}{p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v \xrightarrow{\Phi\,,\,p\bullet o\,\{x\mapsto v\}\,1\,v} \mathtt{n}!x}\,(s\text{-}com)}{[x]\,(\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \mathtt{n}!x\cdot\{x\mapsto v\}}\,(s\text{-}del_{com})}{[p]\,[x]\,(\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} [p]\,\mathtt{n}!v \equiv \mathtt{n}!v}\,(s\text{-}del_{pass})}{[p]\,[x]\,(\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \mathtt{n}!v}\,(s\text{-}cong)}{true \vdash [p]\,[x]\,(\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \succ\!\!\xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \Phi \vdash \mathtt{n}!v}\,(constServ)$$

where $\Phi = (\,true \wedge true \wedge p = p \wedge o = o \wedge v \neq \mathrm{confRec}(p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v, p\bullet o)\,)$. Since $\mathrm{confRec}(p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v, p\bullet o) = \emptyset$, condition Φ holds *true*.

External and internal communication. Consider the constrained service $true \vdash [x]\,(\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,)$. In this case, both internal and external communication can take place. Its initial transitions are the following ones:

$$(ext.\ com.) \quad true \vdash [x]\,(\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x} \neq \mathbf{bn} \vdash \mathtt{m}!\underline{x} \mid \mathtt{n}!v$$

$$(ext.\ com.) \quad true \vdash [x]\,(\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{true\,,\,\mathtt{n}\lhd v} true \vdash [x]\,(\mathtt{n}?x.\,\mathtt{m}!x\,)$$

$$(int.\ com.) \quad true \vdash [x]\,(\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{\Phi\,,\,\mathtt{n}\,\emptyset\,1\,v} \Phi \vdash \mathtt{m}!v$$

where $\Phi = (\,true \wedge true \wedge \mathtt{n} = ce \wedge v \neq \mathrm{confRec}(\mathtt{n}?x.\mathtt{m}!x \mid \mathtt{n}!v, \mathtt{n})\,)$. Since $\mathrm{confRec}(\mathtt{n}?x.\mathtt{m}!x \mid \mathtt{n}!v, \mathtt{n}) = \emptyset$, condition Φ holds *true*.

Conflicting receive. Consider the constrained service $true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,)$. Due to the presence of the receive $\mathtt{n}?v$, that has greater priority to synchronize with an invocation $\mathtt{n}!v$, the receive $\mathtt{n}?x$ can communicate with the environment only if the received value is not v (indeed, $\mathrm{confRec}((\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v), \mathtt{n}) = \{v\}$):

$$true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{\underline{x}\neq\mathbf{bn}\,\wedge\,\underline{x}\neq v\,,\,\mathtt{n}\rhd[x]} \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq v \vdash \mathtt{n}?v \mid \mathtt{n}!v$$

Other possible transitions are as follows:

$$true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{true\,,\,\mathtt{n}\lhd v} true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x\,)$$

$$true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{true\,,\,\mathtt{n}\rhd v} true \vdash [x]\,(\mathtt{n}?x \mid \mathtt{n}!v\,)$$

$$true \vdash [x]\,(\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\!\xrightarrow{true\,,\,\mathtt{n}\,\emptyset\,0\,v} true \vdash [x]\,\mathtt{n}?x$$

On constrained services. Consider the (plain) service $[x,y]\,(\mathtt{n}?q \mid \mathtt{n}?x \mid x\bullet o!v \mid q\bullet o?y)$ where $\mathtt{n} \neq q\bullet o$. It can perform the following transition:

$$[x,y]\,(\mathtt{n}?q \mid \mathtt{n}?x \mid x\bullet o!v \mid q\bullet o?y) \xrightarrow{\underline{x}\neq \mathbf{bn} \wedge \underline{x}\neq q\,,\,\mathtt{n}\rhd[x]} [y]\,(\mathtt{n}?q \mid \underline{x}\bullet o!v \mid q\bullet o?y)$$

The obtained service can further perform the following transition:

$$[y]\,(\mathtt{n}?q \mid \underline{x}\bullet o!v \mid q\bullet o?y) \xrightarrow{\underline{x}=q\,,\,q\bullet o\,\emptyset\,1\,v} \mathtt{n}?q$$

Condition $\underline{x} = q$ of this transition contradicts condition $\underline{x} \neq q$ of the previous one, but the service can however evolve. Instead, by considering constrained services, we would have:

$$\begin{array}{l} \mathit{true} \vdash [x,y]\,(\mathtt{n}?q \mid \mathtt{n}?x \mid x\bullet o!v \mid q\bullet o?y) \succ \xrightarrow{\underline{x}\neq \mathbf{bn} \wedge \underline{x}\neq q\,,\,\mathtt{n}\rhd[x]} \\ \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq q \vdash [y]\,(\mathtt{n}?q \mid \underline{x}\bullet o!v \mid q\bullet o?y) \succ \xrightarrow{\underline{x}=q \wedge \underline{x}\neq \mathbf{bn} \wedge \underline{x}\neq q\,,\,q\bullet o\,\emptyset\,1\,v} \mathit{false} \vdash \mathtt{n}?q \end{array}$$

because $\underline{x} = q \wedge \underline{x} \neq q$ holds *false*, and the second transition could not be performed. That's why we resort to constrained services.

Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables. Consider the service $s \triangleq [y,z]\,(\mathtt{n}!(5+\underline{x}) \mid \mathtt{n}?y.s' \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')$, where $\mathtt{n}$, $\mathtt{m}$ and $\mathtt{m}'$ are pairwise distinct. If $[\![5+\underline{x}]\!] = ((\underline{r} \neq \mathbf{bn} \wedge r \notin \mathbf{uv} \wedge \underline{r} = 5+\underline{x}), \underline{r})$ then

$$\mathtt{n}!(5+\underline{x}) \xrightarrow{(\underline{r}\neq \mathbf{bn} \wedge r\notin \mathbf{uv} \wedge \underline{r}=5+\underline{x})\,,\,\mathtt{n}\lhd \underline{r}} \mathbf{0}$$

Therefore, the constrained service $\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \vdash s$ can evolve as follows:

$$\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \vdash s \succ \xrightarrow{\Phi'\,,\,\mathtt{n}\,\emptyset\,1\,\underline{r}} \Phi' \vdash \underbrace{[z]\,(s' \cdot \{y \mapsto \underline{r}\} \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')}_{s''}$$

for $\Phi' = \mathcal{B}((\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \wedge \underline{r} \neq \mathbf{bn} \wedge r \notin \mathbf{uv} \wedge \underline{r} = 5+\underline{x}), s'', \{x,x',z'\}) = (\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \wedge \underline{r} \neq \mathbf{bn} \wedge r \notin \{x,x',z'\} \wedge \underline{r} = 5+\underline{x})$. Now, we cannot α-convert variable z into r, because we would violate the assumption that bound variables differ from variables corresponding to unknown values (in this case, variable z must be different from r because $\underline{r}$ is an unknown value occurring in the constrained service). Similarly, if $[\![5+\underline{x}]\!] = ((\underline{z} \neq \mathbf{bn} \wedge z \notin \mathbf{uv} \wedge \underline{z} = 5+\underline{x}), \underline{z})$, then the constrained service would become

$$\Phi'' \vdash [z]\,(s' \cdot \{y \mapsto \underline{z}\} \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')$$

for some Φ'', and the assumption would be violated again (because the service contains both z and $\underline{z}$). Finally, if $[\![5+\underline{x}]\!] = ((\underline{z'} \neq \mathbf{bn} \wedge z' \notin \mathbf{uv} \wedge \underline{z'} = 5+\underline{x}), \underline{z'})$, i.e. the unknown value returned by the evaluation function is not fresh, then the condition on the symbolic transition holds *false*, because $z' \notin \{x,x',z'\}$ does not hold.

4.4.4 Extensions of the symbolic operational semantics

In this section, we present two extensions of COWS symbolic semantics for dealing with open terms and polyadic communication.

4.4.4.1 Symbolic semantics for open terms

The symbolic operational semantics presented in Section 4.4.2 is defined only for closed terms. Indeed, for a reduction semantics it is reasonable that well-formed services may not contain free variables and labels. However, in order to be able to inspect also the behaviour of a service component, we need to define the semantics also for open terms.

For example, let us consider the following open term:

$$\mathtt{n}?x \mid \mathtt{n}!x$$

The term can only perform the receive activity $\mathtt{n}?x$ (by communicating with the environment), because activity $\mathtt{n}!x$ is stuck until variable x is not replaced by a value. However, since the scope of the variable is not declared in the term, the environment can substitute the variable with an unknown value in any moment. The resulting term is as follows:

$$\mathtt{n}?\underline{x} \mid \mathtt{n}!\underline{x}$$

Now, the term can perform also the activity $\mathtt{n}!\underline{x}$ (by communicating with the environment) and the internal communication due to synchronisation of activities $\mathtt{n}?\underline{x}$ and $\mathtt{n}!\underline{x}$.

Formally, the symbolic operational semantics for open terms is defined by the rules in Table 4.12 and the new rules in Table 4.14, where the transition label $\underline{x}$ represents execution of a substitution by the environment. We denote by $\mathrm{fv}(t)$ the set of variables in t, and we exploit a predicate noKill(_), a slightly modified variant of that defined in Section 3.2.3, whose most significant case is $\mathrm{noKill}(\mathbf{kill}(k)) = \mathbf{false}$ (this way, the predicate holds true if there are not free kill activities that can be immediately performed).

We comment on salient points. Rules *(constServ)*, *(constServ$_{exp}$)* and *(constServ$_{inv}$)* deal with invoke, input and communication actions that do not involve free variables. They differ from that shown in Table 4.13 for the addition of the predicate $\mathrm{noKill}(s)$ to their premises. Rule *(constServ$_{kill}$)* permits executing (bound and free) kill activities, while rules *(constServ$_{sub}$)*, *(constServ$_{rec}$)* and *(constServ$_{com}$)* deal with actions involving free variables (i.e. assignment of an unknown value to a free variable by the environment, execution of a receive having a free variable as argument, execution of a communication assigning a value to a free variable, respectively).

The presence of predicate $\mathrm{noKill}(s)$ in the rules of Table 4.14 guarantees the eager execution of unbounded kill activities. Indeed, for instance, the open term $(\mathbf{kill}(k) \mid \mathtt{n}?v)$ can only evolve as follows (rule *(constServ$_{kill}$)*):

$$\mathit{true} \vdash (\mathbf{kill}(k) \mid \mathtt{n}?v) \xrightarrow{\mathit{true}\,,\,k} \succ \mathit{true} \vdash \mathbf{0}$$

$$\frac{s \xrightarrow{\Phi', \alpha} s' \qquad \alpha = \mathtt{n} \rhd \underline{v}, \mathtt{n} \rhd [x] \ , \mathtt{n}\, \emptyset\, \ell\, \underline{v} \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \alpha} \Phi'', \Delta \vdash s'} \textit{(constServ)}$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd [n]} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd [n]} \Phi'', \Delta \cup \{n\} \vdash s'} \textit{(constServ}_{exp}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd \underline{v}} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd \underline{v}} \Phi'', \Delta \vdash s'} \textit{(constServ}_{inv}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \alpha} s' \qquad \alpha = k\,, \dagger \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \alpha} \Phi'', \Delta \vdash s'} \textit{(constServ}_{kill}\textit{)}$$

$$\frac{x \in \mathrm{fv}(s) \qquad \Phi' = \mathcal{B}((\Phi \wedge \underline{x} \neq \mathbf{bn}), s, \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi', \underline{x}} \Phi', \Delta \vdash s \cdot \{x \mapsto \underline{x}\}} \textit{(constServ}_{sub}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \mathtt{n} \rhd x} s' \qquad \mathrm{noKill}(s) \qquad \Phi'' = \mathcal{B}((\Phi \wedge \Phi' \wedge \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq \mathrm{confRec}(s, \mathtt{n})), s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \mathtt{n} \rhd x} \Phi'', \Delta \vdash s' \cdot \{x \mapsto \underline{x}\}} \textit{(constServ}_{rec}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \mathtt{n}\{x \mapsto \underline{v}\}\, \mathtt{l}\, \underline{v}} s' \qquad \Phi'' = \mathcal{B}((\Phi \wedge \Phi'), s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\xrightarrow{\Phi'', \mathtt{n}\{x \mapsto \underline{v}\}\, \mathtt{l}\, \underline{v}} \Phi'', \Delta \vdash s' \cdot \{x \mapsto \underline{v}\}} \textit{(constServ}_{com}\textit{)}$$

Table 4.14: Symbolic semantics for COWS open terms

We explain how the rules dealing with free variables work by means of some examples. By applying rule *(constServ$_{rec}$)*, the term $(\mathtt{n}?x \mid \mathtt{n}!x)$ can communicate with the environment (by receiving an unknown value) and evolve as follows:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!x) \succ\!\xrightarrow{\underline{x} \neq \mathbf{bn}, \mathtt{n} \rhd x} \underline{x} \neq \mathbf{bn} \vdash \mathtt{n}!\underline{x}$$

Notably, variable x is replaced by an unknown value, thus now the invoke activity can be performed. By applying rule *(constServ$_{sub}$)*, the same term becomes closed:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!x) \succ\!\xrightarrow{\underline{x} \neq \mathbf{bn}, \underline{x}} \underline{x} \neq \mathbf{bn} \vdash (\mathtt{n}?\underline{x} \mid \mathtt{n}!\underline{x})$$

Now, both receive and invoke activities can communicate with the environment and also internal communication can take place. Finally, if we slightly modify the term as ($\mathtt{n}?x \mid \mathtt{n}!v \mid s$), by applying rule *(constServ$_{com}$)*, we obtain the following transition:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!v \mid s) \xrightarrow{\Phi\,,\,\mathtt{n}\{x\mapsto v\}\,1\,v} \succ \Phi \vdash s\cdot\{x \mapsto v\}$$

where $\Phi = (\mathit{true} \wedge \mathit{true} \wedge \mathtt{n} = ce \wedge v \neq \mathrm{confRec}(\mathtt{n}?x \mid \mathtt{n}!v \mid s, \mathtt{n}))$. Also in this case the substitution for x is applied to the whole term.

4.4.4.2 Symbolic semantics for COWS with polyadic communication

We now tailor COWS syntax and symbolic semantics to deal with polyadic communication. We first extend the syntax of invoke and receive activities as follows: $\underline{u} \bullet \underline{u'}!\bar{\epsilon}$ stands for an invoke over the endpoint $\underline{u} \bullet \underline{u'}$ with parameter the tuple of expressions $\bar{\epsilon}$, while $p \bullet o?\overline{\underline{w}}.s$ stands for a receive over the endpoint $p \bullet o$ with parameter the tuple of variables/(unknown) values $\overline{\underline{w}}$ and continuation s. Tuples can be constructed using a concatenation operator defined as $\langle a_1, \ldots, a_n\rangle : \langle b_1, \ldots, b_m\rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$. To single out an element of a tuple, we will write $(\bar{a}, c, \bar{b})$ to denote the tuple $\langle a_1, \ldots, a_n, c, b_1, \ldots, b_m\rangle$, where $\bar{a}$ or $\bar{b}$ might not be present, and we will use $\bar{a}_i$ to denote the i-th element of the tuple $\bar{a}$. Finally, we denote by $\mathrm{v}(t)$ the set of variables in t.

The labelled transition relation $\xrightarrow{\Phi\,,\,\alpha}$ over services now is induced by the modified rules shown in Table 4.15 (the remaining ones are those of Table 4.12, except for rule *(s-match)* which we do not need anymore), where:

- conditions can also have the form $\Phi \vee \Phi'$; we will use $\overline{\underline{x}} \neq \mathbf{bn}$ to denote condition $\underline{x}_1 \neq \mathbf{bn} \wedge \ldots \wedge \underline{x}_n \neq \mathbf{bn}$ for $\overline{\underline{x}} = \langle \underline{x}_1, \ldots, \underline{x}_n\rangle$;
- action labels are generated by the following grammar:

$$\alpha \;::=\; \underline{\mathtt{n}} \lhd \overline{\underline{v}} \;\mid\; \underline{\mathtt{n}} \lhd [\bar{n}]\,\overline{\underline{v}} \;\mid\; \mathtt{n} \rhd \overline{\underline{w}} \;\mid\; \mathtt{n} \rhd [\bar{x}]\,\overline{\underline{w}} \;\mid\; \mathtt{n}\,\sigma\,\ell\,\overline{\underline{v}} \;\mid\; k \;\mid\; \dagger$$

All the above definitions shall extend to relation $\succ\xrightarrow{\Phi\,,\,\alpha}$.

The new rules exploit a modified version of functions $\mathcal{M}(\_\,,\_)$ and $\mathrm{noConf}(\_\,,\_\,,\_\,,\_)$ defined in Tables 3.3 and 3.7, now redefined by the rules in Table 4.16. The rules in the upper part of the table state that variables match any value, and two values $\underline{v}$ and $\underline{v'}$ do match only if condition $\underline{v} = \underline{v'}$ holds. When tuples $\overline{\underline{w}}$ and $\overline{\underline{v}}$ do match, $\mathcal{M}(\overline{\underline{w}}, \overline{\underline{v}})$ returns a pair (Φ, σ), where Φ is the condition so that the matching holds and σ is a substitution for the variables in $\overline{\underline{w}}$; otherwise, it is undefined. Function $\mathrm{noConf}(s, \mathtt{n}, \overline{\underline{v}}, \ell)$ now returns a condition that guarantees absence of conflicts for the inferred transition. Basically, $\mathrm{noConf}(s, \mathtt{n}, \overline{\underline{v}}, \ell)$ exploits function $\mathit{rec}(s, \mathtt{n}, \overline{\underline{v}}, \ell)$ to identify the conflicting receives of s, then for each argument $\overline{\underline{w}}$ of these receives it determines a condition (i.e. a logical disjunction of inequalities) that makes the conflicting matching between $\overline{\underline{w}}$ and $\overline{\underline{v}}$ false.

$$\mathtt{n}?\overline{\underline{w}}.s \xrightarrow{\mathit{true}\,,\,\mathtt{n}\rhd\overline{\underline{w}}} s \quad \textit{(s-rec)} \qquad \frac{\mathrm{v}(\overline{\underline{w}}) = \bar{x} \qquad |\bar{x}| \geqslant 1}{\mathtt{n}?\overline{\underline{w}}.s \xrightarrow{\bar{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[\bar{x}]\overline{\underline{w}}} s} \quad \textit{(s-rec}_{com}\textit{)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\mathtt{n}\rhd[\bar{x}]\overline{\underline{w}}} s' \qquad y \in \bar{x}}{[y]\,s \xrightarrow{\Phi\,,\,\mathtt{n}\rhd[\bar{x}]\overline{\underline{w}}} s' \cdot \{y \mapsto \underline{y}\}} \quad \textit{(s-del}_{sub1}\textit{)} \qquad \frac{[\![\epsilon_1]\!] = (\Phi_1, \underline{v}_1) \quad \ldots \quad [\![\epsilon_n]\!] = (\Phi_n, \underline{v}_n)}{\underline{\mathtt{n}}!\langle\epsilon_1,\ldots,\epsilon_n\rangle \xrightarrow{\Phi_1\wedge\ldots\wedge\Phi_n\,,\,\underline{\mathtt{n}}\lhd\langle\underline{v}_1,\ldots,\underline{v}_n\rangle} \mathbf{0}} \quad \textit{(s-inv)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\underline{\mathtt{n}}\lhd\overline{\underline{v}}} s' \quad n \in \overline{\underline{v}} \quad n \notin \underline{\mathtt{n}}}{[n]\,s \xrightarrow{\Phi\,,\,\underline{\mathtt{n}}\lhd[n]\overline{\underline{v}}} s'} \quad \textit{(s-open}_1\textit{)} \qquad \frac{s \xrightarrow{\Phi\,,\,\underline{\mathtt{n}}\lhd[\bar{m}]\overline{\underline{v}}} s' \quad n \in \overline{\underline{v}} \quad n \notin \underline{\mathtt{n}}}{[n]\,s \xrightarrow{\Phi\,,\,\underline{\mathtt{n}}\lhd[\langle n\rangle:\bar{m}]\overline{\underline{v}}} s'} \quad \textit{(s-open}_2\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi_1\,,\,\mathtt{n}\rhd\overline{\underline{w}}} s_1' \quad s_2 \xrightarrow{\Phi_2\,,\,\underline{\mathtt{n}}'\lhd\overline{\underline{v}}} s_2' \quad \mathcal{M}(\overline{\underline{w}},\overline{\underline{v}}) = (\Phi,\sigma) \quad \mathrm{noConf}(s_1 \mid s_2, \mathtt{n}, \overline{\underline{v}}, |\sigma|) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi_1\wedge\Phi_2\wedge\mathtt{n}=\underline{\mathtt{n}}'\wedge\Phi\wedge\Phi'\,,\,\mathtt{n}\sigma|\sigma|\overline{\underline{v}}} s_1' \mid s_2'} \quad \textit{(s-com)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\mathtt{n}\sigma\uplus\{x\mapsto\underline{v}\}\,\ell\,\overline{\underline{v}}} s'}{[x]\,s \xrightarrow{\Phi\,,\,\mathtt{n}\sigma\,\ell\,\overline{\underline{v}}} s' \cdot \{x \mapsto \underline{v}\}} \quad \textit{(s-del}_{sub2}\textit{)} \qquad \frac{s_1 \xrightarrow{\Phi\,,\,\alpha} s_1' \quad \alpha \neq k\,,\,\mathtt{n} \rhd [\bar{x}]\,\overline{\underline{w}}\,,\,\mathtt{n}\sigma\,\ell\,\overline{\underline{v}}}{s_1 \mid s_2 \xrightarrow{\Phi\,,\,\alpha} s_1' \mid s_2} \quad \textit{(s-par)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\,\mathtt{n}\sigma\,\ell\,\overline{\underline{v}}} s_1' \qquad \mathrm{noConf}(s_2, \mathtt{n}, \overline{\underline{v}}, \ell) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\Phi'\,,\,\mathtt{n}\sigma\,\ell\,\overline{\underline{v}}} s_1' \mid s_2} \quad \textit{(s-par}_{com1}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\,\mathtt{n}\rhd[\bar{x}]\overline{\underline{w}}} s_1' \qquad \mathrm{noConf}(s_2, \mathtt{n}, \overline{\underline{w}} \cdot \{\bar{x} \mapsto \overline{\underline{x}}\}, |\bar{x}|) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\Phi'\,,\,\mathtt{n}\rhd[\bar{x}]\overline{\underline{w}}} s_1' \mid s_2} \quad \textit{(s-par}_{com2}\textit{)}$$

Table 4.15: Symbolic semantics with polyadic communication (excerpt of rules)

Finally, it returns the logical conjunction of the determined conditions. We use the auxiliary function $gval(\_)$ that, given a tuple $\overline{\underline{w}}$, returns a collection of pairs of the form $(\underline{x}, i)$, where $\underline{x}$ is an unknown value such that $\overline{\underline{w}}_i = \underline{x}$. Notably, if $rec(s, \mathtt{n}, \overline{\underline{v}}, \ell) = \emptyset$ then function $\mathrm{noConf}(s, \mathtt{n}, \overline{\underline{v}}, \ell)$ returns the condition *true*, because there are not conflicting receives; while, if there is a $\overline{\underline{w}} \in rec(s, \mathtt{n}, \overline{\underline{v}}, \ell)$ such that $gval(\overline{\underline{w}}) = \emptyset$, then the function returns the condition *false*, because there are not conditions that can make the conflicting matching false.

We end this section with an example aimed at clarifying how pattern-matching and conflict checking functions work. Consider the following term:

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z}''\rangle$$

In this case, the invoke activity $\mathtt{n}!\langle v_1, v_2, v_3\rangle$ can synchronize with each receive activity of

$$\mathcal{M}(x, \underline{v}) = (true, \{x \mapsto \underline{v}\}) \qquad \mathcal{M}(\underline{v}, \underline{v}') = (\underline{v} = \underline{v}', \emptyset) \qquad \mathcal{M}(\langle\rangle, \langle\rangle) = (true, \emptyset)$$

$$\frac{\mathcal{M}(a_1, b_1) = (\Phi_1, \sigma_1) \quad \mathcal{M}(\bar{a}_2, \bar{b}_2) = (\Phi_2, \sigma_2)}{\mathcal{M}((a_1, \bar{a}_2), (b_1, \bar{b}_2)) = (\Phi_1 \wedge \Phi_2, \sigma_1 \uplus \sigma_2)}$$

$$\text{noConf}(s, \mathtt{n}, \overline{\underline{v}}, \ell) = \bigwedge_{\overline{\underline{w}} \in rec(s, \mathtt{n}, \overline{\underline{v}}, \ell)} \Big(\bigvee_{(\underline{x}, i) \in gval(\overline{\underline{w}})} \underline{x} \neq \overline{\underline{v}}_i \ \wedge \ (gval(\overline{\underline{w}}) = \emptyset \Rightarrow false) \Big)$$

$$rec(\mathtt{n}?\overline{\underline{w}}.s, \mathtt{n}, \overline{\underline{v}}, \ell) = \begin{cases} \{\overline{\underline{w}}\} & \text{if } \mathcal{M}(\overline{\underline{w}}, \overline{\underline{v}}) = (\Phi, \sigma) \wedge |\sigma| < \ell \\ \emptyset & \text{otherwise} \end{cases}$$

$$rec(\mathbf{0}, \mathtt{n}, \overline{\underline{v}}, \ell) = rec(\mathbf{kill}(k), \mathtt{n}, \overline{\underline{v}}, \ell) = rec(\mathtt{u}!\bar{\epsilon}, \mathtt{n}, \overline{\underline{v}}, \ell) = \emptyset \qquad rec(\mathtt{n}'?\overline{\underline{w}}.s, \mathtt{n}, \overline{\underline{v}}, \ell) = \emptyset \ \text{ if } \mathtt{n} \neq \mathtt{n}'$$

$$rec([e]\, s, \mathtt{n}, \overline{\underline{v}}, \ell) = rec(s, \mathtt{n}, \overline{\underline{v}}, \ell) \ \text{ if } e \notin \mathtt{n} \qquad rec([e]\, s, \mathtt{n}, \overline{\underline{v}}, \ell) = \emptyset \ \text{ if } e \in \mathtt{n}$$

$$rec(g + g', \mathtt{n}, \overline{\underline{v}}, \ell) = rec(g, \mathtt{n}, \overline{\underline{v}}, \ell) \cup rec(g', \mathtt{n}, \overline{\underline{v}}, \ell) \qquad rec(\{\!|s|\!\}, \mathtt{n}, \overline{\underline{v}}, \ell) = rec(s, \mathtt{n}, \overline{\underline{v}}, \ell)$$

$$rec(s \mid s', \mathtt{n}, \overline{\underline{v}}, \ell) = rec(s, \mathtt{n}, \overline{\underline{v}}, \ell) \cup rec(s', \mathtt{n}, \overline{\underline{v}}, \ell) \qquad rec(* s, \mathtt{n}, \overline{\underline{v}}, \ell) = rec(s, \mathtt{n}, \overline{\underline{v}}, \ell)$$

Table 4.16: Modified matching and conflicting receives rules

the term. Firstly, consider the receive $\mathtt{n}?\langle x, y, z\rangle$: its argument $\langle x, y, z\rangle$ matches the tuple $\langle v_1, v_2, v_3\rangle$ by generating the substitution $\{x \mapsto v_1, y \mapsto v_2, z \mapsto v_3\}$. The other two receive activities are in conflict, because they satisfy the matching with the invoke and generate substitutions with fewer pairs than 3. Thus, function $rec(\_, \_, \_, \_)$ applied to the whole term[5] returns the set $\{\langle x', \underline{y}', \underline{z}'\rangle, \langle x'', y'', \underline{z}''\rangle\}$. Then, function $\text{noConf}(\_, \_, \_, \_)$ returns the condition $(\underline{y}' \neq v_2 \vee \underline{z}' \neq v_3) \wedge \underline{z}'' \neq v_3$. Hence, a transition of the term is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\, \mathtt{n}?\langle x, y, z\rangle \mid [x']\, \mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\, \mathtt{n}?\langle x'', y'', \underline{z}''\rangle$$
$$\xrightarrow{(\underline{y}' \neq v_2 \vee \underline{z}' \neq v_3) \wedge \underline{z}'' \neq v_3\,,\, \mathtt{n}\,\emptyset\,3\,\langle v_1, v_2, v_3\rangle} [x']\, \mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\, \mathtt{n}?\langle x'', y'', \underline{z}''\rangle$$

Consider now the receive $\mathtt{n}?\langle x'', y'', \underline{z}''\rangle$: in this case the matching function returns condition $\underline{z}'' = v_3$ and substitution $\{x'' \mapsto v_1, y'' \mapsto v_2\}$. Function $rec(\_, \_, \_, \_)$ applied to the whole term returns the set $\{\langle x', \underline{y}', \underline{z}'\rangle\}$, because the only conflicting receive is $\mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle$. Thus, the corresponding transition is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\, \mathtt{n}?\langle x, y, z\rangle \mid [x']\, \mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\, \mathtt{n}?\langle x'', y'', \underline{z}''\rangle$$
$$\xrightarrow{(\underline{y}' \neq v_2 \vee \underline{z}' \neq v_3) \wedge \underline{z}'' = v_3\,,\, \mathtt{n}\,\emptyset\,2\,\langle v_1, v_2, v_3\rangle} [x, y, z]\, \mathtt{n}?\langle x, y, z\rangle \mid [x']\, \mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle$$

Moreover, the receive activities can communicate with the environment; in this case the conflict checks are performed by rule *(s-par$_{com2}$)*. For example, the transition corre-

[5]This means that the last rule applied in the inference is *(s-com)*. Of course, the last rule could be also *(s-par$_{com1}$)*; in this case, two or three conflict checks will be performed on subterms of the considered service.

sponding to the execution of $\mathtt{n}?\langle x, y, z\rangle$ is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z}''\rangle \xrightarrow{\underline{x}\neq\mathbf{bn} \wedge \underline{y}\neq\mathbf{bn} \wedge \underline{z}\neq\mathbf{bn} \wedge (\underline{y}'\neq\underline{y} \vee \underline{z}'\neq\underline{z}) \wedge \underline{z}''\neq\underline{z},\, \mathtt{n} \rhd [\langle x,y,z\rangle]\,\langle x,y,z\rangle} \mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z}''\rangle$$

Finally, as another example consider the following term:

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', v_2, v_3\rangle$$

If we try to infer the transition corresponding to the communication between $\mathtt{n}!\langle v_1, v_2, v_3\rangle$ and $\mathtt{n}?\langle x, y, z\rangle$, we have that the condition on the transition label is *false*, because function $rec(\_, \_, \_, \_)$ returns $\langle x', v_2, v_3\rangle$ and $gval(\langle x', v_2, v_3\rangle) = \emptyset$.

4.4.5 Concluding remarks

Symbolic semantics and symbolic bisimulation were first introduced in [105] by Hennessy and Lin on value-passing process algebras. The symbolic approach has been then applied to π-calculus in [180] by Sangiorgi and in [36] by Boreale and De Nicola. Victor has adopted a similar approach in [198] to efficiently characterise hyperequivalence for the fusion calculus. A more recent work on a symbolic semantics for a fusion-based calculus is [48] by Buscemi and Montanari. A revisited symbolic technique for π-calculus has been recently proposed in [33] by Bonchi and Montanari.

We believe that the alternative symbolic operational semantics defined in this section can pave the way for the development of efficient model and equivalence checkers for COWS. In fact, the model checking approach introduced in Section 4.2 does not support a fully compositional verification methodology. It allows to analyse systems of services 'as a whole', but does not enable analysis of services in isolation (e.g. a provider service without a proper client). The symbolic operational semantics should permit to overcome this limitation that is somewhat related to the original semantics of COWS which, although based on an LTS, follows a reduction style. Furthermore, the symbolic operational semantics can be used to improve efficiency of checking the equivalences introduced in Section 4.3. This, of course, requires defining alternative characterizations of the equivalences on top of the symbolic transition system. We plan to pursue these lines of research in the near future, and in particular to implement the operational semantics and equivalence and model checkers on top of it.

Chapter 5

On the expressiveness of COWS

In the previous two chapters, we showed that COWS can model different and typical aspects of SOC, and presented some methods and tools to analyse COWS terms. In this chapter, we want to investigate the expressiveness of COWS.

On the one hand, we provide some encodings of other formal languages for SOC: the three orchestration languages Orc, SCC and WS-CALCULUS, the process calculus Lπ, and the modelling language SRML. Translations from higher level languages, besides showing the descriptive power of COWS, pave the way for using the reasoning mechanisms and verification techniques that are being made available for COWS (see Chapter 4 for an account) to verify properties of applications designed with different languages.

On the other hand, we present two COWS variants: a timed extension of COWS, called C$\clock$WS, and a dialect of COWS for concurrent constraint programming, which permits modelling the phases of dynamic service publication, discovery and negotiation of SOC applications. In fact, since it is not known to what extent timed computation can be reduced to untimed forms of computation [197], we show how timed activities can be easily incorporated into COWS. Instead, since COWS definition abstracts from a few sets of objects (e.g., the set of expressions that can occur within terms of the calculus), we appropriately specialise these parameters of the language so that services can specify and conclude Service Level Agreements by generating and composing constraints dynamically. This way, we obtain a linguistic formalism capable of modelling all the phases of the life cycle of SOC applications.

Structure of the chapter. The rest of the chapter is organized as follows. Section 5.1 presents the encodings in COWS of Orc, SCC, WS-CALCULUS, Lπ and SRML. Section 5.2 introduces C$\clock$WS and the variant of COWS for concurrent constraint programming. Each section reports a comparison with related work.

5.1 Encoding other formal languages for SOC

We present here the encodings in COWS of three orchestration languages: Orc, SCC and WS-CALCULUS. The first language is based on the functional paradigm and has already

proved to be capable of expressing the most common workflow patterns; the second one is a language for SOC centered on the explicit modelling of services' interaction sessions and their dynamic creation; the last one exploits message correlation and turned out to be suitable to model in a quite direct way the semantics of whole WS-BPEL [138, 129]. We end the section with the encoding in COWS of Lπ, the variant of π-calculus closest to COWS, and the implementation in COWS of the modelling language SRML.

We prove that there is a formal correspondence, based on the operational semantics (in the style of [158]), between Orc expressions and the COWS services resulting from their encoding. Similar results could be also proved for the other encodings. However, here we aim only at showing the descriptive power of COWS, without losing ourselves in detailed and intricate proofs. We leave as a future work the task of developing a formal account of its expressiveness.

5.1.1 Encoding Orc

Orc [153] is a recently proposed task orchestration language with applications in workflow, business process management, and web service orchestration. We will show that its encoding in COWS enjoys a property of operational correspondence. This is an important witness of COWS's expressiveness because it is known that Orc can express the most common workflow patterns identified in [196]. Orc syntax is:

$$(\textit{Expressions})\ \ f, g \ ::= \ \mathbf{0} \ \mid \ S(w) \ \mid \ E(w) \ \mid \ f > x > g \ \mid \ f \mid g \ \mid \ g \ \textbf{where}\ x :\in f$$

$$(\textit{Parameters})\ \ w \ \ ::= \ x \ \mid \ v$$

where S ranges over *site* names, E over *expression* names, x over variables, and v over values. Each expression name E has a unique declaration of the form $E(x) \triangleq f$. Expressions can be composed by means of sequential composition $\cdot > x > \cdot$, symmetric parallel composition $\cdot \mid \cdot$, and asymmetric parallel composition $\cdot$ **where** $x :\in \cdot$ starting from the elementary expressions $\mathbf{0}$, that is treated as a site that is never called, $S(w)$ (site call) and $E(w)$ (expression call). The variable x is *bound* in g for the expressions $f > x > g$ and g **where** $x :\in f$. We use $fv(f)$ to denote the set of variables that are not bound (i.e. which occur *free*) in f.

Evaluation of expressions may call a number of sites and returns a (possibly empty) stream of values. So, in summary, an Orc expression can be either a site call, an expression call or a composition of expressions according to one of the three basic orchestration patterns.

Site call: a site call can have the form $S(w)$, where the site name is known statically, and w is the parameter of the call. If w is a variable, then it must be instantiated before the call is made.

Expression call: an expression call has the form $E(w)$ and executes the expression defined by $E(x) \triangleq f$ after having replaced x by w. Here w is passed by reference. Note that expression definitions can be recursive.

$$S(v) \overset{!v'}{\hookrightarrow} \mathbf{0} \quad \textit{(SiteCall)} \qquad \frac{E(x) \triangleq f}{E(w) \overset{\tau}{\hookrightarrow} f \cdot \{x \mapsto w\}} \quad \textit{(Def)}$$

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g} \quad \textit{(Sym1)} \qquad \frac{g \overset{l}{\hookrightarrow} g'}{f \mid g \overset{l}{\hookrightarrow} f \mid g'} \quad \textit{(Sym2)}$$

$$\frac{f \overset{\tau}{\hookrightarrow} f'}{f > x > g \overset{\tau}{\hookrightarrow} f' > x > g} \quad \textit{(Seq1)} \qquad \frac{f \overset{!v}{\hookrightarrow} f'}{f > x > g \overset{\tau}{\hookrightarrow} (f' > x > g) \mid g \cdot \{x \mapsto v\}} \quad \textit{(Seq2)}$$

$$\frac{g \overset{l}{\hookrightarrow} g'}{g \textbf{ where } x :\in f \overset{l}{\hookrightarrow} g' \textbf{ where } x :\in f} \quad \textit{(Asym1)}$$

$$\frac{f \overset{\tau}{\hookrightarrow} f'}{g \textbf{ where } x :\in f \overset{\tau}{\hookrightarrow} g \textbf{ where } x :\in f'} \quad \textit{(Asym2)}$$

$$\frac{f \overset{!v}{\hookrightarrow} f'}{g \textbf{ where } x :\in f \overset{\tau}{\hookrightarrow} g \cdot \{x \mapsto v\}} \quad \textit{(Asym3)}$$

Table 5.1: Orc asynchronous operational semantics

Symmetric parallel composition: the composition $f \mid g$ executes both f and g concurrently, assuming that there is no interaction between them. It publishes the interleaving of the two streams of values published by f and g, in temporal order.

Sequential composition: the composition $f > x > g$ executes f, and, for each value v returned by f, it executes an instance of g with v assigned to x. It publishes the interleaving (in temporal order) of the streams of values published by the different instances of g.

Asymmetric parallel composition: the composition g **where** $x :\in f$ starts in parallel both f and the parts of g that do not need x. When f publishes the first value, let say v, it is killed and v is assigned to x. The composition publishes the stream obtained from g (instantiated with v).

The asynchronous operational semantics of Orc is given by the labelled transition relation $\overset{l}{\hookrightarrow}$ defined in Table 5.1, where label τ indicates an internal event while label $!v$ indicates the value v resulting from evaluating an expression. A site call can progress only when the actual parameter is a value v (rule *(SiteCall)*); it elicits one response v'. While site calls use a call-by-value mechanism, expression calls use a call-by-name mechanism

$$\langle\!\langle \mathbf{0} \rangle\!\rangle_{\mathtt{r}} = \mathbf{0} \qquad \langle\!\langle S(w) \rangle\!\rangle_{\mathtt{r}} = \mathtt{S}!\langle w, \mathtt{r} \rangle \qquad \langle\!\langle E(w) \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}'](\mathtt{E}!\langle \mathtt{r}, \mathtt{r}' \rangle \mid [z]\, \mathtt{r}'?\langle z \rangle.\, z!\langle w \rangle)$$

$$\langle\!\langle f > x > g \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}_f](\langle\!\langle f \rangle\!\rangle_{\mathtt{r}_f} \mid *[x]\, \mathtt{r}_f?\langle x \rangle.\, \langle\!\langle g \rangle\!\rangle_{\mathtt{r}}) \qquad \langle\!\langle f \mid g \rangle\!\rangle_{\mathtt{r}} = \langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}}$$

$$\langle\!\langle g \textbf{ where } x :\in f \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}_f, x]\ (\ \langle\!\langle g \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\ \langle\!\langle f \rangle\!\rangle_{\mathtt{r}_f} \mid \mathtt{r}_f?\langle x \rangle.\, \textbf{kill}(k)\ )\ )$$

Table 5.2: Orc encoding

(rule *(Def)*), namely the actual parameter replaces the formal one and then the corresponding expression is evaluated. Symmetric parallel composition $f \mid g$ consists of concurrent evaluations of f and g (rules *(Sym1)* and *(Sym2)*). Sequential composition $f > x > g$ activates a concurrent copy of g with x replaced by v, for each value v returned by f (rules *(Seq1)* and *(Seq2)*). Asymmetric parallel composition g **where** $x :\in f$ prunes threads selectively. It starts in parallel both f and the part of g that does not need x (rules *(Asym1)* and *(Asym2)*). The first value returned by f is assigned to x and the continuation of f and all its descendants are then terminated (rule *(Asym3)*).

Notably, the presented operational semantics slightly simplifies that described in [153], in a way that does not misrepresent the properties of the encoding. Indeed, in the original semantics, a site call involves three steps: invocation of the site, response from the site, and publication of the result. Here, instead, a site call is performed in one step, that corresponds to the immediate publication of the result.

The encoding of Orc expressions in COWS exploits function $\langle\!\langle \cdot \rangle\!\rangle_{\mathtt{r}}$ shown in Table 5.2. The function is defined by induction on the syntax of expressions and is parameterized by the endpoint $\mathtt{r}$ used to return the result of expressions evaluation. Thus, a site call is rendered as an invoke activity that sends a pair made of the parameter of the invocation and the endpoint for the reply along the endpoint $\mathtt{S}$ corresponding to site name S. Expression call is rendered similarly, but we need two invoke activities: $\mathtt{E}!\langle \mathtt{r}, \mathtt{r}' \rangle$ activates a new instance of the body of the declaration, while $z!\langle w \rangle$ sends the value of the actual parameter (when this value will be available) to the created instance, by means of a private endpoint stored in z received from the encoding of the corresponding expression declaration along the private endpoint $\mathtt{r}'$ previously sent. Sequential composition is encoded as the parallel composition of the two components sharing a delimited endpoint, where a new instance of the component on the right is created every time that on the left returns a value along the shared endpoint. Symmetric parallel composition is encoded as parallel composition, where the values produced by the two components are sent along the same return endpoint. Finally, asymmetric parallel composition is encoded in terms of parallel composition in such a way that, whenever the encoding of f returns its first value, this is passed to the encoding of g (since f and g share the variable x) and a kill activity is enabled. Due to its eager semantics, the kill will terminate what remains of the term corresponding to the encoding of f.

Moreover, for each site S, we define the service:

$$* [x, y] \, \mathtt{S}?\langle x, y\rangle.y!\langle \epsilon_x^S \rangle \tag{5.1}$$

that receives along the endpoint $\mathtt{S}$ a value (stored in x) and an endpoint (stored in y) to be used to send back the result, and returns the evaluation of ϵ_x^S, an unspecified expression corresponding to S and depending on x.

Similarly, for each expression declaration $E(x) \triangleq f$ we define the service:

$$* [y, z] \, \mathtt{E}?\langle y, z\rangle.[\mathtt{r}] \, (z!\langle \mathtt{r}\rangle \mid [x] \, (\mathtt{r}?\langle x\rangle \mid \langle\!\langle f \rangle\!\rangle_y) \,) \tag{5.2}$$

Here, the received value (stored in x) is processed by the encoding of the body of the declaration, that is activated as soon as the expression is called.

Finally, the encoding of an Orc expression f, written $[\![f]\!]_{\mathtt{r}}$, is the parallel composition of $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}}$, of a service of the form (5.1) or (5.2) for each site or expression called in f, in any of the expressions called in f, and so on recursively.

We can prove that there is a formal correspondence, based on the operational semantics, between Orc expressions and the COWS services resulting from their encoding. To simplify the proof, we found it convenient to extend the syntax of Orc expressions with $f \cdot \{x \mapsto y\}$, that behaves as the expression obtained from f by replacing all free occurrences of x with y. Correspondingly, we add the following rule to those defining the operational semantics:

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \cdot \{x \mapsto y\} \overset{l}{\hookrightarrow} f' \cdot \{x \mapsto y\}} \; \textit{(Sub)}$$

Next proposition, that can be easily proved by induction on the syntax of expressions, states that there is an operational correspondence between the extended semantics and the original one.

Proposition 5.1.1 *If $y \notin fv(f)$, then $f \overset{l}{\hookrightarrow} f'$ iff $f \cdot \{x \mapsto y\} \overset{l}{\hookrightarrow} f' \cdot \{x \mapsto y\}$.*

Expression $f \cdot \{x \mapsto y\}$ is encoded as the parallel composition of the encoding of f with a receive activity $\mathtt{r}'?\langle x\rangle$, that initializes the shared variable x, and with an invoke activity $\mathtt{r}'!\langle y\rangle$, that forwards the value of variable y (when it will be available) along the private endpoint $\mathtt{r}'$. Formally, it is defined as

$$\langle\!\langle f \cdot \{x \mapsto y\} \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}'] \, (\mathtt{r}'!\langle y\rangle \mid [x] \, (\mathtt{r}'?\langle x\rangle \mid \langle\!\langle f \rangle\!\rangle_{\mathtt{r}}) \,)$$

The *operational correspondence* between Orc expressions and the COWS services resulting from their encoding can be characterized by two propositions, which we call *completeness* and *soundness*. The former states that all possible executions of an Orc expression can be simulated by its encoding, while the latter states that the initial step of a COWS term resulting from an encoding can be simulated by the corresponding Orc

expression so that the continuation of the encoding can evolve in the encoding of the expression continuation. By letting $s \stackrel{\alpha}{\Longrightarrow} s'$ to mean that there exist two services, s_1 and s_2, such that s_1 is a reduct of s, $s_1 \stackrel{\alpha}{\longrightarrow} s_2$ and s' is a reduct of s_2, the two properties can be stated as follows.

Theorem 5.1.1 (Completeness) *Given an Orc expression f and an endpoint $\mathtt{r}$, $f \stackrel{l}{\hookrightarrow} f'$ implies $[\![f]\!]_{\mathtt{r}} \equiv (\!|f'|\!)_{\mathtt{r}} \mid s \stackrel{\alpha}{\Longrightarrow} (\!|f'|\!)_{\mathtt{r}} \mid s$, where $\alpha = \mathtt{r} \lhd \langle v \rangle$ if $l = !v$, and $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$ if $l = \tau$.*

Proof (sketch). By induction on the length of the inference of $f \stackrel{l}{\hookrightarrow} f'$, with a case analysis on the last rule used in the derivation. □

Lemma 5.1.1 *Given an Orc expression f and an endpoint $\mathtt{r}$, $[\![f]\!]_{\mathtt{r}} \xrightarrow{\mathtt{r} \lhd \langle v \rangle}\!\!\!\!\rightarrow$.*

Proof. By a straightforward induction on the definition of the encoding. □

Theorem 5.1.2 (Soundness) *Given an Orc expression f and an endpoint $\mathtt{r}$, $[\![f]\!]_{\mathtt{r}} \equiv (\!|f|\!)_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s'$ implies that there exists an Orc expression f' such that $f \stackrel{l}{\hookrightarrow} f'$ and $s' \stackrel{\alpha}{\Longrightarrow} (\!|f'|\!)_{\mathtt{r}} \mid s$, where $\alpha = \mathtt{r} \lhd \langle v \rangle$ if $l = !v$, and $\alpha = \mathtt{n}'\,\emptyset\,\ell'\,\bar{v}'$ if $l = \tau$.*

Proof (sketch). By induction on the definition of the encoding $(\!|f|\!)_{\mathtt{r}}$. □

As usual, for the sake of readability, we have only outlined here the techniques used in the proofs. We refer the interested reader to Appendix B.4 for a full account.

5.1.2 Encoding SCC

SCC (*Service Centered Calculus* [34]) is a formalism aiming to serve as a basis for programming and composing services. SCC integrates complementary aspects borrowed from well-known process calculi, such as π-calculus (names handling primitives), Orc (pipelining and pruning of activities), web-π, cJoin, and Sagas (long running transactions and compensations). A key feature of SCC is the session handling mechanism, that allows for the definition of structured interaction protocols. A *session* permits bi-directional communication between two parties (service- and client-side). Transparently for the programmers, sessions are opened by service invocations, which spawn fresh session parties (locally to each partner). Moreover, sessions can be nested and values can be returned outside sessions. Finally, SCC sessions can be closed by the involved parties, providing a mechanism for process interruption and service cancellation and update.

The syntax of SCC, given in Table 5.3, is parameterized by a countable set of names, ranged over by a, b, ..., x, y, ... Basically, in SCC processes are defined as parallel compositions of service definitions and service invocations. *Service definitions* take the

P, Q, T	::=		(processes)
		$\mathbf{0}$	(nil)
	|	$a.P$	(concretion)
	|	$(x)P$	(abstraction)
	|	return $a.P$	(return value)
	|	$a \Rightarrow (x)P : (y)T$	(service definition)
	|	$a\{(x)P\} \Leftarrow_h Q$	(service invocation)
	|	$a \triangleright_h P$	(session)
	|	$P \mid Q$	(parallel composition)
	|	$(\nu a)P$	(new name)

Table 5.3: SCC syntax

form $a \Rightarrow (x)P : (y)T$, where a is the service name, x is a formal parameter, P is the actual implementation of the service, and $(y)T$ is the protocol of the termination handler service that will be instantiated on the server-side in service invocation. *Service invocations* are written as $a\{(x)P\} \Leftarrow_h Q$: each new value v produced by the client Q will trigger a new invocation of service a; for each invocation, an instance of the process P, with x bound to the actual invocation value v, implements the client-side protocol for interacting with the new instance of a. Name h identifies the termination handler service on the client-side. A service invocation causes activation of a new *session*: a pair of dual fresh names, r and $\tilde{r}$, identifies the two sides of the session. A session side has the form $r \triangleright_h P$ and the first time the protocol P invokes the termination handler service identified by h (i.e. that of the other side), the session is closed. Notably, in order to program the session termination, the calculus has a special name close, that is replaced by the name of the termination handler instantiated on the other side at invocation time. Within a session, client and server protocols can communicate whenever a *concretion* is available on one side and an *abstraction* is ready on the other side, i.e. abstractions and concretions model input and output, respectively. A value can be returned outside the current session (just one level up) by means of the primitive return .

To illustrate an example of interaction protocol in SCC, consider the (simplified) service definition $succ \Rightarrow (x)\, x + 1$ that models a service that, received an integer, returns its successor. A (simplified) client for this service can be written as $succ\,\{\,(x)\,(y)\,\mathsf{return}\ y\,\} \Leftarrow 5$. After the service invocation, we have that x is bound to the argument 5, the client waits for a value from the server *succ* and the received value is substituted for y and hence returned as the result of the service invocation. As a consequence of the interaction between the client and the server *succ*, the following session is triggered:

$$(\nu r)(r \triangleright 5 + 1 \mid \tilde{r} \triangleright (y)\,\mathsf{return}\ y)$$

The value 6 is computed on the service-side and then received at the client side, that reduces first to $\tilde{r} \triangleright \mathsf{return}\ 6$ and then to $6 \mid \tilde{r} \triangleright \mathbf{0}$. Values returned outside the session to the

enclosing environment can be used to invoke other services. For instance, the following client invokes the service *succ* and then prints the obtained result:

$$print\,\{(z)\,\mathbf{0}\} \Leftarrow (succ\,\{\,(x)\,(y)\,\mathsf{return}\ y\,\} \Leftarrow 5)$$

To take into account unexpected events, the above client can be extended so to exploit a suitable service *fault* that can handle printer failures. Thus, we have the following parallel composition:

$$print\{(z)\mathbf{0}\} \Leftarrow_{fault} (succ\{(x)(y)\mathsf{return}\ y\} \Leftarrow 5) \mid fault \Rightarrow (code)\,ErrorHandling$$

where *ErrorHandling* is a service protocol able to manage printer errors according to their identifier code. Here, we suppose that when invoked by the client, a service-side session of the form $r \triangleright_{fault} P[fault/\mathsf{close}]$ is created, where P is the printer protocol and *fault* is substituted for close. In case of printer failure the protocol P should invoke the service close (instantiated to *fault*), with an error code *err* as a parameter. As effect of this invocation, the whole service-side session r is destroyed and the invocation will instantiate an error recovery session that executes $ErrorHandling[err/code]$.

Since COWS is a calculus with asynchronous communication while SCC relies on synchronous communication, then for the sake of the presentation, we define the encoding $[\![\_]\!]^K_{\mathtt{in,out,ret}}$, from SCC to 'synchronous COWS' and exploit the auxiliary encoding $\langle\!\langle\_\rangle\!\rangle$, from the synchronous version of COWS to the original one. The auxiliary encoding acts as an homomorphism except for the communication activities:

$$\begin{array}{rcl} \langle\!\langle u \bullet u'!\langle \epsilon_1, \ldots \epsilon_n\rangle.s\rangle\!\rangle & = & [\mathtt{r}]\,(u \bullet u'!\langle \epsilon_1, \ldots \epsilon_n, \mathtt{r}\rangle \mid \mathtt{r}?\langle\rangle.\langle\!\langle s\rangle\!\rangle) \\ \langle\!\langle p \bullet o?\langle w_1, \ldots, w_n\rangle.s\rangle\!\rangle & = & [x]\,(p \bullet o?\langle w_1, \ldots, w_n, x\rangle.(x!\langle\rangle \mid \langle\!\langle s\rangle\!\rangle)\,) \end{array}$$

The encoding $[\![\_]\!]^K_{\mathtt{in,out,ret}}$, shown in Table 5.4, is inspired by that from the close-free fragment of SCC to π-calculus [34]. The encoding is parametric on three endpoints used to receive values from (in), send values to (out), and return values to the enclosing session (ret). Moreover, it exploits an auxiliary function K, that maps SCC names used to identify termination handler services (ranged over by h) to COWS killer labels (ranged over by k).

We comment on salient points. Concretion and return actions are rendered as outputs on the endpoints out and ret respectively, while abstractions are rendered as inputs on in. The three endpoints are set by the encoding of sessions that, to permit closing a session, also introduces a delimitation for a killer label (associated to the session name by updating the function K). To model bi-directional sessions we associate a pair of endpoints $\mathtt{a}_1$ and $\mathtt{a}_2$ to each session name a. Parallel composition and restriction operators are mapped homomorphically. In the encoding of restriction, the extra condition $a \notin dom(K)$ avoids name conflicts between the encoded term and the function K, and can always be satisfied by first α-converting the a in $(va)P$. To model service definitions and invocations we exploit a distinguished endpoint scc; each service definition waits invocations

(concretion) $[\![a.P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = \mathtt{out}!\langle a\rangle.[\![P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}}$

(abstraction) $[\![(x)P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = [x]\,\mathtt{in}?\langle x\rangle.[\![P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}}$

(return value) $[\![\mathsf{return}\ a.P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = \mathtt{ret}!\langle a\rangle.[\![P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}}$

(session) $[\![a \triangleright_h P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = [k]\,[\![P]\!]^{K,\{h\mapsto k\}}_{\mathtt{a}_1,\mathtt{a}_2,\mathtt{out}}$

$[\![\tilde{a} \triangleright_h P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = [k]\,[\![P]\!]^{K,\{h\mapsto k\}}_{\mathtt{a}_2,\mathtt{a}_1,\mathtt{out}}$

(new name) $[\![(\nu a)P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = [a]\,[\![P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}}$ if $a \notin dom(K)$

(parallel comp.) $[\![P \mid Q]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = [\![P]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} \mid [\![Q]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}}$

(service definition)

$$[\![a \Rightarrow (x)P : (z)T]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = *\,[x_1,x_2,y_1,y_2,x]\,\mathtt{scc}?\langle a,x_1,x_2,y_1,y_2,x\rangle.\ [k]\,(\,[\![P[y_1/\mathsf{close}]]\!]^{K,\{y_1\mapsto k\}}_{x_2,x_1,\mathtt{out}} \mid *\,[z]\,\mathtt{th}?\langle y_2,z\rangle.[\![T[y_1/\mathsf{close}]]\!]^{K,\{y_1\mapsto k\}}_{\_,\_,\mathtt{out}})$$

(service invocation)

$$[\![a\{(x)P\} \Leftarrow_h Q]\!]^K_{\mathtt{in},\mathtt{out},\mathtt{ret}} = \begin{cases} [\mathtt{r}]\,(\,[\![Q]\!]^K_{\mathtt{in},\mathtt{r},\mathtt{ret}} \mid *\,[x]\,\mathtt{r}?\langle x\rangle.\ [h',\mathtt{i},\mathtt{o}]\,\mathtt{scc}!\langle a,\mathtt{i},\mathtt{o},h,h',x\rangle.\ [k]\,[\![P[h'/\mathsf{close}]]\!]^{K,\{h'\mapsto k\}}_{\mathtt{i},\mathtt{o},\mathtt{out}}) & \text{if } a \notin dom(K) \\ [\mathtt{r}]\,(\,[\![Q]\!]^K_{\mathtt{in},\mathtt{r},\mathtt{ret}} \mid *\,[x]\,\mathtt{r}?\langle x\rangle.\ \mathtt{th}!\langle a,x\rangle.\mathbf{kill}(K(a))\,) & \text{if } a \in dom(K) \end{cases}$$

Table 5.4: SCC encoding

on this endpoint, and request messages are routed to the correct service by means of their first field, that is the name of the invoked service and acts as a correlation value. Similarly, we exploits a distinguished endpoint `th` to model termination handler services. A service definition, when it is invoked (along `scc`), instantiates the service protocol and the termination handler service, by using the received data. Name close, in the service protocol and in the termination handler, is replaced by a killer label corresponding to the client termination handler name. Finally, for service invocation we differentiates two cases: service invocation and termination handler invocation. In both cases, outputs on the fresh endpoint `r` where process Q produces the parameters for service invocation are intercepted, and each value v triggers an invocation of the service a. In the former case ($a \notin dom(K)$), the invoked endpoint is `scc`, and the transmitted data are as follows: the service name a (for correlation purpose), two fresh endpoints `i` and `o` for bi-directional session communication, two termination handler names h and h' (the former is installed

n	::=	$a :: C$	(nodes)
C	::=	$*s \mid m \gg s$	(components)
	$\mid$	$\langle a, o, \bar{u} \rangle \mid C \mid C$	
m	::=	$\emptyset \mid \{p = u\} \mid m \cup m$	(correl. constraints)
s	::=		(services)
		$\mathbf{0}$	(null)
	$\mid$	$\mathbf{exit}$	(exit)
	$\mid$	$\mathbf{ass}\,(\bar{w}, \bar{\epsilon})$	(assign)
	$\mid$	$\mathbf{inv}\,(r, o, \bar{w})$	(invoke)
	$\mid$	$\mathbf{rec}\,(r, o, \bar{w})$	(receive)
	$\mid$	$\mathbf{if}\ (\epsilon)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{s\}$	(switch)
	$\mid$	$s; s$	(sequence)
	$\mid$	$s \mid s$	(flow)
	$\mid$	$\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$	(pick)
	$\mid$	$A(\bar{w})$	(call)

Table 5.5: WS-CALCULUS syntax

at client-side, while the latter will be installed at server-side), and the produced value v. After the invocation, the client protocol P is instantiated. In case of invocation of a termination handler service ($a \in dom(K)$), the invoked endpoint is `th`, and after this invocation, the corresponding session is closed by means of the kill activity.

5.1.3 Encoding WS-CALCULUS

WS-CALCULUS [130] is a process language that formalizes the semantics of an expressive subset of WS-BPEL, with special concern for modeling the interactions among web services, be them WS-BPEL processes or not, in a network context. The *syntax* of (an *untyped* variant of) WS-CALCULUS, given in Table 5.5, is parameterized with respect to the following syntactic sets: *properties* (sorts of late bound constants storing some relevant values within service instances, ranged over by p), *values* (basic values and addresses, ranged over by u), *partner links* (variables storing addresses used to identify service partners within an interaction), *operation parameters* (basic variables, partner links and properties, ranged over by w), and *service identifiers* (ranged over by A). Notationally, we will use a to range over *addresses* and r to range over addresses and partner links.

WS-CALCULUS permits to model the interactions among web service instances in a network context. A *network* of services is a finite set of nodes. *Nodes*, written as $a :: C$, are uniquely identified by an address a and host components C. *Components* C may be service specifications, instances or requests. The behavioural specification of a service s is written $*s$, while $m \gg s'$ represents a service instance that behaves according to s'

(nodes)	$\langle\!\langle a :: C \rangle\!\rangle$	$=$	$\langle\!\langle C \rangle\!\rangle_a$
(components)	$\langle\!\langle * s \rangle\!\rangle_a$	$=$	$* \, [k_a, V(s)] \, \langle\!\langle s \rangle\!\rangle_a$
	$\langle\!\langle \{\bar{p} = \bar{u}\} \gg s \rangle\!\rangle_a$	$=$	$[k_a, V(s)] \, \langle\!\langle s \cdot \{\bar{p} \mapsto \bar{u}\} \rangle\!\rangle_a$
	$\langle\!\langle \langle a', o, \bar{u} \rangle \rangle\!\rangle_a$	$=$	$\{\!\!\{ a \bullet o ! \langle a', \bar{u} \rangle \}\!\!\}$
(null)	$\langle\!\langle \mathbf{0} \rangle\!\rangle_a$	$=$	$\mathbf{0}$
(exit)	$\langle\!\langle \mathbf{exit} \rangle\!\rangle_a$	$=$	$\mathbf{kill}(k_a)$
(assign)	$\langle\!\langle \mathbf{ass}\,(\bar{w}, \bar{\epsilon}) \rangle\!\rangle_a$	$=$	$[\bar{w} = \bar{\epsilon}]$
(invoke)	$\langle\!\langle \mathbf{inv}\,(r, o, \bar{w}) \rangle\!\rangle_a$	$=$	$r \bullet o ! \langle a, \bar{w} \rangle$
(receive)	$\langle\!\langle \mathbf{rec}\,(r, o, \bar{w}) \rangle\!\rangle_a$	$=$	$a \bullet o ? \langle r, \bar{w} \rangle$
(call/def)	$\langle\!\langle A(\bar{w}) \rangle\!\rangle_a$	$=$	$[\mathtt{A}] \, (\, \mathtt{A} ! \langle \bar{w} \rangle \mid [\bar{x}] \, \mathtt{A} ? \langle \bar{x} \rangle . \langle\!\langle s \rangle\!\rangle_a \,)$ if $A(\bar{x}) \stackrel{def}{=} s$

Table 5.6: ws-calculus encoding (an excerpt)

and whose properties evaluate according to the (possibly empty) set m of correlation constraints. A *correlation constraint* is a pair, written $p = u$, recording the value u assigned to the property p. Properties are used to store values that are important to identify service instances. A service request $\langle a, o, \bar{u} \rangle$ represents an operation invocation that must still be processed and contains the invoker address a, the operation name o and the data $\bar{u}$ for operation execution.

Services are structured activities built from *basic activities*, i.e. instance forced termination **exit**, assignment **ass** (_ , _), service invocation **inv** (_ , _ , _) and service request processing **rec** (_ , _ , _), by exploiting operators for conditional choice **if** (_) **then** {_} **else** {_} (*switch*), sequential composition _ ; _ (*sequence*), parallel composition _ | _ (*flow*), external choice $\sum_{i \in I}$ **rec** (_ , _ , _) ; _ (*pick*) and service call $A(\bar{w})$ (of course, we assume that every service identifier A has a unique definition of the form $A(\bar{x}) \stackrel{def}{=} s$).

The encoding from ws-calculus to COWS, denoted by $\langle\!\langle \cdot \rangle\!\rangle$, is defined inductively on the syntax of ws-calculus and is shown in Table 5.6. We only show the relevant cases; for example, the encodings of structural activities are quite standard and, hence, omitted. The encoding of a service net is the parallel composition of the encodings of its nodes. The encoding of a node is given in term of the encoding of the hosted components parameterized by the address of the node. Both variables and properties occurring in service components are encoded as COWS variables.

The auxiliary parameterized encoding $\langle\!\langle \cdot \rangle\!\rangle_a$ is defined inductively over the syntax of services. The parameter a is used both as the partner name to which a given request must be sent and as a reference for the killer label k_a used to identify each service instance. Service specifications are encoded as COWS persistent services, by exploiting the replication operator. The fact that variables are global to service instances is rendered through the

$\langle\!\langle a \rangle\!\rangle_{S \cup \{a\}} = x_a \bullet y_a$	$\langle\!\langle a \rangle\!\rangle_S = p_a \bullet o_a \quad \text{if } a \notin S$
$\langle\!\langle \mathbf{0} \rangle\!\rangle_S = \mathbf{0}$	$\langle\!\langle a(b).P \rangle\!\rangle_S = [\langle\!\langle b \rangle\!\rangle_{S'}]\, \langle\!\langle a \rangle\!\rangle_{S'} ? \langle\!\langle b \rangle\!\rangle_{S'} . \langle\!\langle P \rangle\!\rangle_{S'} \quad S' = S \cup \{b\}$
$\langle\!\langle (\nu a)P \rangle\!\rangle_S = [\langle\!\langle a \rangle\!\rangle_S]\, \langle\!\langle P \rangle\!\rangle_S$	$\langle\!\langle \bar{a}b \rangle\!\rangle_S = \langle\!\langle a \rangle\!\rangle_S \, ! \langle\!\langle b \rangle\!\rangle_S$
$\langle\!\langle P_1 \mid P_2 \rangle\!\rangle_S = \langle\!\langle P_1 \rangle\!\rangle_S \mid \langle\!\langle P_2 \rangle\!\rangle_S$	$\langle\!\langle !a(b).P \rangle\!\rangle_S = * \langle\!\langle a(b).P \rangle\!\rangle_S$

Table 5.7: Lπ encoding

outermost delimitation $[k_a, V(s)]$, where $V(s)$ is the set of free variables and properties of s. Partner links used to identify service partners within an interaction are translated by exploiting the address of the hosting node. The effect of executing **exit** inside a service instance hosted at a is achieved by forcing termination of the COWS term resulting from the encoding the instance and identified by the label k_a. Basic activity **ass** $(\bar{w}, \bar{\epsilon})$ is encoded by $[\bar{w} = \bar{\epsilon}]$, which is the term (3.3) introduced in Section 3.2.1.3. Finally, the case of service call is handled along the lines of the encoding of the Orc site call (presented in Section 5.1.1).

5.1.4 Encoding localised π-calculus

Localised π-calculus (Lπ [147]) is the variant of π-calculus closest to COWS. In fact, all Lπ constructs have a direct counterpart in COWS and is indeed possible to define an encoding that enjoys operational correspondence. More precisely, the syntax of Lπ processes is

$$P ::= \mathbf{0} \quad | \quad a(b).P \quad | \quad \bar{a}b \quad | \quad P \mid P \quad | \quad (\nu a)P \quad | \quad !a(b).P$$

with the constraint that in processes $a(b).P$ and $!a(b).P$ name b may not occur free in P in input position. For simplicity sake, we define the encoding only for Lπ processes such that their bound names are all distinct and different from the free ones (but it can be easily extended to deal with all processes). The crux of the encoding is mapping each Lπ channel name in a COWS endpoint, that is composed of variables if the channel name is bound by an input prefix (because in Lπ the name is used as a placeholder and COWS distinguishes between names and variables), and of names otherwise. The actual encoding function $\langle\!\langle \cdot \rangle\!\rangle_S$, that is parameterized by a set of names S, is defined by induction on the syntax of Lπ processes by the clauses in Table 5.7 (where the endpoint $p_a \bullet o_a$ is sometimes used in place of the tuple $\langle p_a, o_a \rangle$). The encoding of process P is given by service $\langle\!\langle P \rangle\!\rangle_S$ with $S = \emptyset$; as the encoding proceeds, S is used to record the names that have been freed and were initially bound by an input prefix.

5.1.5 Encoding SRML

SRML (Sensoria *Reference Modelling Language*, [90]) provides primitives for modelling composite services and activities whose business logic involves the orchestration of inter-

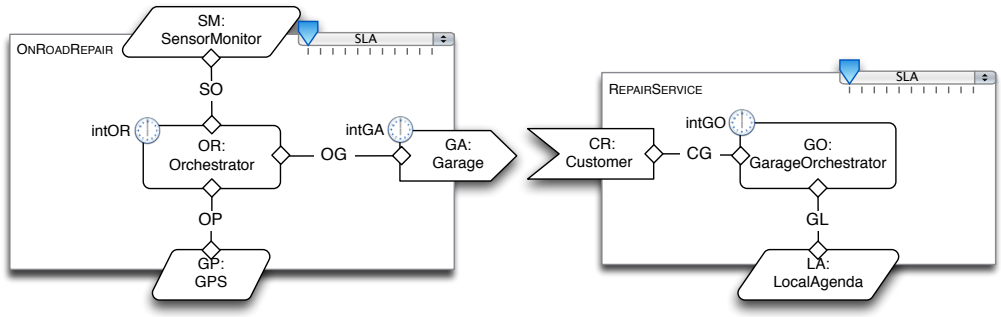

Figure 5.1: Activity module OnRoadRepair and service module RepairService

actions among more elementary components and the invocation of services provided by external parties. SRML is inspired by SCA (*Service Component Architecture*, [17]) and is independent of the languages and platforms that are currently being provided for web [6] (or grid [93]) services.

To illustrate and discuss the use of SRML and its methodology, we consider the scenario depicted in Figure 5.1, which is a simplified model of the automotive case study presented in Section 2.4.1. The scenario involves an activity OnRoadRepair that takes place in a software system (embedded in a vehicle) handling engine failures detected by a sensor. When the activity is triggered, the system (1) determines the current location of the car by using a GPS device, and (2) binds to a repair service selected among those offered by nearby garages that can ensure best levels of assistance, including a tow truck if necessary.

In SRML, *modules* are used for specifying (business) activities and services. There are two kinds of modules. *Activity* modules specify applications developed according to requirements provided by a specific business organisation. An example is the activity OnRoadRepair that will have been developed by, or for, the car manufacturer. *Service* modules are developed (by, or for, service providers) to be published in repositories in ways that allow them to be discovered when a request for an external service is published in the run-time environment. An example is the repair service that OnRoadRepair will procure when the engine-failure sensor is activated.

A module is specified in terms of a number of entities and the way they are interconnected. For example, the activity module OnRoadRepair shown in Figure 5.1 (left-hand side) involves the following software entities: SM (the interface to the sensor that triggers the activity), GP (the interface to the GPS system), and OR (the orchestrator that coordinates the interactions with the external services and GP). These entities are interconnected through *wires*, each of which defines an interaction protocol between two entities. Typically, wires deal with the heterogeneity of partners involved in the activity by performing data (or, more generally, semantic) integration, which is useful when, for instance, a car has to travel across different countries. OnRoadRepair relies on an external service for

booking a garage and calling a tow-truck, the discovery of which will be triggered, on-the-fly, according to the conditions detected by the sensor. This dependency is made explicit through the requires interface GA.

As illustrated, every activity module declares interfaces of four possible kinds: (1) one and only one *serves-interface* that binds the activity to the application that triggered its execution (e.g., SM on the left-hand side of Figure 5.1), (2) a number of *uses-interfaces* (possibly none) that bind to persistent components already present in the configuration when the activity is launched (e.g., GP on the left-hand side of Figure 5.1), (3) a number of *component-interfaces* (at least one) that bind to components that are created when the activity is launched (e.g., OR on the left-hand side of Figure 5.1), (4) a number of *requires-interfaces* (possibly none) that bind the activity to services that are procured externally when certain conditions become true (e.g., GA on the left-hand side of Figure 5.1).

Service modules such as RepairService in Figure 5.1 (right-hand side) provide a service to the external environment and can be dynamically discovered and invoked (instead of being launched directly by users). They have one *provides-interface* — CR in the example — instead of a serves-interface. Notice that the workflow of a module is defined collectively by the components in its configuration and the wires that connect them, which facilitates modular development and reuse driven by the structure of the business domain.

SRML also offers primitives for defining internal and external configuration policies. The internal policies (indicated by clocks) define the initialisation and termination conditions of each component and the conditions that trigger the discovery process of each external service. For instance, intGA in Figure 5.1 is the condition that triggers the discovery of GA; it is defined in terms of the events that can occur during the execution of OnRoadRepair. The external policies (indicated by the rulers) express constraints for Service Level Agreements (SLA).

The graphical notation used to specify the automotive case study has the advantage of being intuitive and facilitating the identification of the relationships among the involved entities. However, it abstracts from a number of details that need to be accounted for when defining an implementation. For this reason, a detailed and 'tractable' textual notation is also defined for SRML, the Backus-Naur Form syntax of which is defined in Appendix C.1. In Appendix C.2, we present the specification of the automotive case study using this textual notation.

Now, we outline how the elements that compose a SRML configuration can be defined in terms of an orchestrated system in COWS. We illustrate our approach by means of the above scenario.

To make the encoding modular, the SRML configuration modelling the automotive case study is decomposed in a number of areas of concern, numbered one to six in Figure 5.2:

(1) *Creation of an activity or service instance.* Every encoding of a SRML module is intended as a *factory* (1a) that handles the creation of different instances. Each instance of a module has an associated *instance handler* (1b) that implements mes-

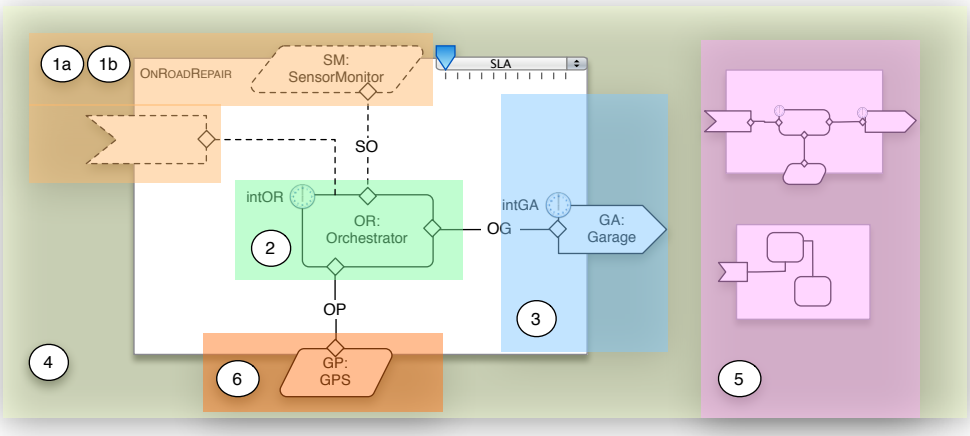

Figure 5.2: Decomposition of OnRoadRepair into areas of concern

sage correlation and maps the interaction/parameter names of the interface to those of the correct components of the module.

(2) *Orchestration.* The orchestration consists of the executable pattern of interactions described by the set of components internal to the SRML module.

(3) *Discovery of a service.* To bind new service components to those in the instance that triggered the discovery, we need what we call a *discovery handler*. From a module's perspective, the information for handling the process of discovery of each of its requires interfaces includes (1) a specification of the required syntactic/behavioural properties (i.e., the business protocol), (2) a specification of the SLA constraints given by the external policies and (3) the condition that triggers the discovery process (i.e., the trigger condition associated with a specific requires-interface in SRML). The discovery handler of a module includes a *requires handler* for each requires-interface of the module. A requires handler implements the mapping of names and parameters of a specific requires-interface to those of the components of the discovered module as established by the wires.

(4) *Middleware.* It consists of those functionalities that support the execution of SRML configurations. Among other things, the middleware enables the discovery and binding processes by relying on a *broker* — a discovery and reasoner entity that selects the most suitable service that matches a given requires-interface among those stored in a *repository*. The middleware also includes a *matchmaking agent* supporting the matching of functional descriptions and a *constraint solver* supporting the negotiation of Quality of Service properties. This is where COWS offers a layer of abstraction that is still above that of a dedicated middleware, thus allowing us

to 'parametrise' the encoding and remain independent of specific technologies. For instance, web service architectures currently provide only very limited brokerage facilities via the technology UDDI [6].

(5) *Environment.* It consists of the activities and services published in some repository.

(6) *Bottom layer.* It consists of the set of persistent entities, which typically already exist when a service instance is created and which may be shared among different instances (e.g., GP of type GPS in OnRoadRepair).

According to this architecture, the COWS representation of a service module is

$$Module^{(1,2,3)} \mid Middleware^{4} \mid Environment^{5} \mid BottomLayer^{6}$$

where $Module^{(1,2,3)}$ is of the form:

$$Factory^{1a}.(InstanceHandler^{1b} \mid Orchestration^{2} \mid DiscoveryHandler^{3})$$

The superscripts establish a correspondence between the terms and the parts of a SRML configuration illustrated in Figure 5.2. One advantage of this architecture is that it permits an incremental development of the different aspects of the encoding.

In Appendix C.3 we give a flavour of the implementation of SRML in COWS through the automotive case study previously presented. A more complete account of the encoding can be found in [30].

5.1.6 Concluding remarks

We have defined encodings from a few languages for SOC such as Orc, SCC, ws-calculus, Lπ and SRML to COWS, and we have formalized the properties enjoyed by the first encoding. This is an important evidence of COWS's expressiveness, since such languages address different aspects (at different levels of abstraction) of currently available SOC technologies.

For what concern the encoding of the modelling language SRML, we give a special emphasis to the architecture of the encoding rather than a detailed account. In fact, we consider this to be the main interest of our work in the sense that it reveals general aspects of what it means to encode (i.e. implement) a business modelling language over a calculus of services. Indeed, our encoding is such that the structure of the COWS terms that implement SRML modules reflects the architecture of the configuration management process that is promoted through SRML. More precisely, we partition the encoding into areas of concern that derive from the declarative semantics of SRML [91], which has the advantage of permitting a modular and incremental development of the encoding. The need to easily support message correlation, together with implementation of shared states and forced termination of (parts of) services, makes COWS a very reasonable choice as target language of the encoding with respect to the many other calculi for SOC proposed in the literature (among which we want to mention [127, 104, 57, 35, 45, 199]).

5.2 COWS's variants

In this section, we present two variants of COWS. The first one extends COWS with timed orchestration constructs; this way we obtain a language, that we call C⏲WS, capable of completely formalizing the semantics of WS-BPEL. The second one permits comfortably modelling Quality of Service requirement specifications and Service Level Agreement achievements, and the phases of dynamic service publication, discovery and negotiation. This can be achieved by smoothly incorporating in COWS constraints and operations on them.

5.2.1 Timed extensions of COWS

This extension is obtained by considering an analogous of WS-BPEL's *wait* activity which causes execution of the invoking service to be suspended until the time interval specified as an argument has elapsed. For the sake of presentation, we postpone the introduction of attribute *until*, that causes suspension of the invoking service until the absolute time reaches the value specified as an argument.

5.2.1.1 Extending COWS with the 'wait' activity

We assume that the set of values now includes a set of positive numbers (ranged over by $\delta, \delta', \ldots$), used to represent *time intervals*. The syntax of COWS is extended as follows:

$$g \;::=\; \ldots \;\mid\; ⏲_{\epsilon}.s$$

Basically, guards are extended with the *wait activity* $⏲_{\epsilon}$, that specifies the time interval, whose value is given by evaluation of ϵ, the executing service has to wait for. Consequently, the choice construct can now be guarded both by message reception and timeout expiration, like WS-BPEL *pick* activity. We assume that evaluation of expressions and execution of basic activities, except for $⏲_{\epsilon}$, are instantaneous (i.e. do not consume time units) and time elapses between them.

The operational semantics of the extended language is defined in terms of the labelled transition relation $\xrightarrow{\hat{\alpha}}$, where $\hat{\alpha}$ stands for α or δ (that models time elapsing), obtained by adding the rules shown in Table 5.8 to those in Table 3.12. Let us briefly comment on the new rules. Time can elapse while waiting on receive/invoke activities, rules *(rec$_{elaps}$)* and *(inv$_{elaps}$)*. When time elapses, but the timeout is still not expired, the argument of wait activities $⏲_{\_}$ is updated (rule *(wait$_{elaps}$)*). Time elapsing cannot make a choice within a pick activity (rule *(pick)*), while the occurrence of a timeout can. Indeed, it is signalled by label †, thus is a computation step, generated by rule *(wait$_{tout}$)* and used by rule *(choice)* to discard the alternative branches. Time elapses synchronously for all services running in parallel: this is modelled by rule *(par$_{sync}$)* and by the remaining rules for empty activity (rule *(nil$_{elaps}$)*), replication (rule *(rep$_{elaps}$)*), wait activity (rule *(wait$_{err}$)*), protection (rule

$$\mathbf{0} \xrightarrow{\delta} \mathbf{0} \ (nil_{elaps}) \qquad * s \xrightarrow{\delta} * s \ (rep_{elaps}) \qquad \mathtt{n}?\bar{w}.s \xrightarrow{\delta} \mathtt{n}?\bar{w}.s \ (rec_{elaps})$$

$$\mathtt{u}!\bar{\epsilon} \xrightarrow{\delta} \mathtt{u}!\bar{\epsilon} \ (inv_{elaps}) \qquad \textcircled{\scriptsize 0}.s \xrightarrow{\dagger} s \ (wait_{tout}) \qquad \frac{\delta \leqslant [\![\epsilon]\!]}{\textcircled{\scriptsize ϵ}.s \xrightarrow{\delta} \textcircled{\scriptsize $[\![\epsilon-\delta]\!]$}.s} \ (wait_{elaps})$$

$$\frac{[\![\epsilon]\!] \neq \delta'}{\textcircled{\scriptsize ϵ}.s \xrightarrow{\delta} \textcircled{\scriptsize ϵ}.s} \ (wait_{err}) \qquad \frac{s \xrightarrow{\delta} s'}{\{\!|s|\!\} \xrightarrow{\delta} \{\!|s'|\!\}} \ (prot_{elaps}) \qquad \frac{g_1 \xrightarrow{\delta} g_1' \qquad g_2 \xrightarrow{\delta} g_2'}{g_1 + g_2 \xrightarrow{\delta} g_1' + g_2'} \ (pick)$$

$$\frac{s_1 \xrightarrow{\delta} s_1' \qquad s_2 \xrightarrow{\delta} s_2'}{s_1 \mid s_2 \xrightarrow{\delta} s_1' \mid s_2'} \ (par_{sync}) \qquad \frac{s \xrightarrow{\delta} s'}{[e]\, s \xrightarrow{\delta} [e]\, s'} \ (scope_{elaps})$$

Table 5.8: Synchronous timed COWS operational semantics (additional rules)

(prot$_{elaps}$)) and delimitation (rule *(scope$_{elaps}$)*). In particular, rule *(wait$_{err}$)* enables time passing for the wait activity also when the expression ϵ used as an argument does not return a positive number; in this case the argument of the wait is left unchanged. Note that, in agreement with its eager semantics, the kill activity does not allow time to pass. Computations can now also include transitions labelled by δ corresponding to time elapsing.

Since time elapses synchronously for all services in parallel, we can think of as all services run on a same service *engine* and share the same clock. By making it explicit the notion of service engine and of deployment of services on engines, it is possible to describe more realistic scenarios. Therefore, we extend the language syntax with the syntactic category of (*service*) *engines* defined as follows:

$$\mathbb{E} \ ::= \ 0 \ \mid \ \{s\} \ \mid \ [n]\,\mathbb{E} \ \mid \ \mathbb{E} \mid \mathbb{E}$$

Each engine $\{s\}$ has its own clock (whose value does not matter and, hence, is not made explicit), that is not synchronized with the clock of other parallel engines (namely, time progresses asynchronously among different engines). Besides, (private) names can be shared among engines, while variables and killer labels cannot. In the sequel, we will only consider *well-formed* engine compositions, i.e. engine compositions where partners used in endpoints of receive activities within different service engines are pairwise distinct. The underlying rationale is that each service has its own partner names and that the service and all its instances run within the same engine.

To define the semantics, we first extend the structural congruence of Section 3.2 with the abelian monoid laws for engines parallel composition and with the following laws:

$$\{s\} \equiv \{s'\} \quad \text{if } s \equiv s' \qquad \{\mathbf{0}\} \equiv 0 \qquad \{[n]\,s\} \equiv [n]\,\{s\} \qquad [n]\,0 \equiv 0$$

$$\frac{s \xrightarrow{\hat{\alpha}} s' \qquad \hat{\alpha} \in \{\delta, \dagger, \mathtt{n}\,\emptyset\,\ell\,\bar{v}\}}{\{s\} \longrightarrow \{s'\}}\ (loc) \qquad \frac{\mathbb{E} \longrightarrow \mathbb{E}'}{[n]\,\mathbb{E} \longrightarrow [n]\,\mathbb{E}'}\ (res)$$

$$\frac{\mathbb{E} \equiv \mathbb{E}' \qquad \mathbb{E}' \longrightarrow \mathbb{F}' \qquad \mathbb{F}' \equiv \mathbb{F}}{\mathbb{E} \longrightarrow \mathbb{F}}\ (str_E) \qquad \frac{\mathbb{E} \longrightarrow \mathbb{E}'}{\mathbb{E} \mid \mathbb{F} \longrightarrow \mathbb{E}' \mid \mathbb{F}}\ (par_{async})$$

$$\frac{s_1 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_1' \qquad s_2 \xrightarrow{\mathtt{n} \rhd \bar{w}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \mathrm{fv}(\bar{w}) = \bar{x} \qquad \mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, |\sigma|)}{\{s_1\} \mid \{[\bar{x}]\, s_2\} \longrightarrow \{s_1'\} \mid \{s_2' \cdot \sigma\}}\ (com_E)$$

Table 5.9: Asynchronous timed COWS operational semantics (additional rules)

$$[n]\,[m]\,\mathbb{E} \;\equiv\; [m]\,[n]\,\mathbb{E} \qquad\qquad \mathbb{E} \mid [n]\,\mathbb{F} \;\equiv\; [n]\,(\mathbb{E} \mid \mathbb{F}) \quad \text{if } n \notin \mathrm{fe}(\mathbb{E})$$

The first law lifts to engines the structural congruence defined on services, the second law transforms an engine with empty activities into an empty engine, while the third law permits to extrude a private name outside an engine. The remaining laws are standard.

Secondly, we define a reduction relation $\longrightarrow$ among engines through the rules shown in Table 5.9. Rule *(loc)* models occurrence of a computation step within an engine, while rule *(res)* deals with private names. Rule (str_E) says that structurally congruent engines have the same behaviour, while rule (par_{async}) says that time elapses asynchronously between different engines (indeed, $\mathbb{F}$ and, then, the clocks of its engines remain unchanged after the transition). Rule (com_E), where $\mathrm{fv}(\bar{w})$ are the free variables of $\bar{w}$, enables interaction between services executing within different engines. It combines the effects of rules $(del_{com\,2})$ and (com_2) in Table 3.12. Indeed, since the delimitations $[\bar{x}]$ for the input variables are singled out, the communication effect can be immediately applied to the continuation s_2' of the service performing the receive. The last premise ensures that, in case of multiple start activities, the message is routed to the correlated service instance rather than triggering a new instantiation.

Notably, computations from a given parallel composition of engines are sequences of (connected) reductions. Communication can take place *intra-engine*, by means of rule (com_2), or *inter-engine*, by means of rule (com_E). In both cases, since we are only considering well-formed compositions of engines, checks for receive conflicts are confined to services running within a single engine, the one performing the receive, differently from the language without explicit engines, where checks involve the whole composition of services. Notice that, to communicate a private name between engines, first it is necessary to exploit the structural congruence for extruding the name outside the sending engine and to extend its scope to the receiving engine, then the communication can take place, by applying rules (com_E), *(res)* and (str_E).

5.2.1.2 Extending COWS with the 'wait until' activity

We introduce the attribute *until* that modifies the semantics of wait activity so that the invoking service is suspended until the absolute time reaches the value resulting by the evaluation of the argument of the *wait until*, denoted by $\clockop_{U\,\epsilon}$. The set of values now includes also a set of *time values*, ranged over by t, t', ..., that are used to explicitly indicate the value of engines' clock, now denoted by $\{s\}_t$.

The operational semantics changes accordingly. In particular, the laws for structural congruence over engines become as follows:

$$\{s\}_t \;\equiv\; \{s'\}_t \quad \text{if } s \equiv s' \qquad\qquad \{[n]\,s\}_t \;\equiv\; [n]\,\{s\}_t$$

The semantics of services is defined in terms of configurations of the form $t \succ s$, for taking into account the absolute time t of the engine where the service s run, and of labelled transitions between configurations $t \succ s \xrightarrow{\hat{\alpha}} t' \succ s'$. The rules in Tables 3.12 and 5.8 are then tailored for using configurations. For example, the rules for the receive activity become as follows:

$$t \succ \mathtt{n}?\bar{w}.s \xrightarrow{\mathtt{n} \rhd \bar{w}} t \succ s \;\; (rec) \qquad\qquad t \succ \mathtt{n}?\bar{w}.s \xrightarrow{\delta} t+\delta \succ \mathtt{n}?\bar{w}.s \;\; (rec_{elaps})$$

Additionally, we have the following rules for the wait until activity:

$$\frac{t+\delta < [\![\epsilon]\!]}{t \succ \clockop_{U\,\epsilon}.s \xrightarrow{\delta} t+\delta \succ \clockop_{U\,\epsilon}.s}\;(waitUntil_1) \qquad \frac{[\![\epsilon - t]\!] = \delta}{t \succ \clockop_{U\,\epsilon}.s \xrightarrow{\delta} [\![\epsilon]\!] \succ \clockop_{0}.s}\;(waitUntil_2)$$

$$\frac{[\![\epsilon]\!] \neq t'}{t \succ \clockop_{U\,\epsilon}.s \xrightarrow{\delta} t+\delta \succ \clockop_{U\,\epsilon}.s}\;(waitUntil_{err})$$

which permit time elapsing until the clock of the engine reaches the argument of $\clockop_{U\,\epsilon}$. Finally, the semantics of engines' composition is tailored by replacing rules *(loc)* and (com_E) with the following ones:

$$\frac{t \succ s \xrightarrow{\delta} t+\delta \succ s'}{\{s\}_t \longrightarrow \{s'\}_{t+\delta}}\;(loc_{elaps}) \qquad \frac{t \succ s \xrightarrow{\alpha} t \succ s' \quad \alpha \in \{\dagger, \mathtt{n}\,\emptyset\,\ell\,\bar{v}\}}{\{s\}_t \longrightarrow \{s'\}_t}\;(loc_{act})$$

$$\frac{\begin{array}{c} t_1 \succ s_1 \xrightarrow{\mathtt{n} \lhd \bar{v}} t_1 \succ s_1' \qquad t_2 \succ s_2 \xrightarrow{\mathtt{n} \rhd \bar{w}} t_2 \succ s_2' \\ \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \mathrm{fv}(\bar{w}) = \bar{x} \qquad \mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, |\sigma|) \end{array}}{\{s_1\}_{t_1} \mid \{\,[\bar{x}]\,s_2\}_{t_2} \longrightarrow \{s_1'\}_{t_1} \mid \{s_2' \cdot \sigma\}_{t_2}}\;(com_E)$$

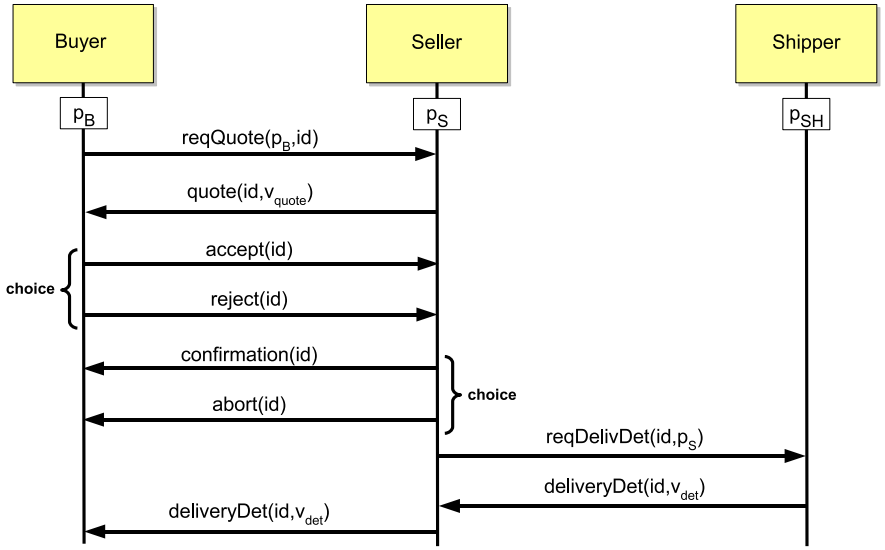

Figure 5.3: Graphical representation of the Buyer/Seller/Shipper protocol

5.2.1.3 Examples

We end this section with some examples of application of the extended framework. The first example provides a sort of clock service, while the last two are use-cases inspired by [58, 59]. Moreover, in Section 5.2.2.3 timed activities are exploited to model a variant of the automotive case study presented in Section 2.4.1.

A clock service. By using the timed activity, we can define a sort of *clock service* that is set to send a message "*tick*" along $\mathtt{m}$ every 10 time units.

$$[\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle\,.\,\oplus_{10}\,.\,(\,\mathtt{m}!\langle\text{“}tick\text{”}\rangle \mid \mathtt{n}!\langle\rangle\,)\,)$$

Notice that, however, because invoke activities are executed lazily (in fact, communication in COWS is asynchronous), it is only guaranteed that *at least* (and *not exactly*) 10 time units elapse between two consecutive emissions of message "*tick*".

A Buyer/Seller/Shipper protocol. We illustrate a simple business protocol for purchasing a fixed good. The protocol, graphically represented in Figure 5.3, involves a buyer, a seller and a shipper. Firstly, *Buyer* asks *Seller* to offer a quote, then, after the *Seller*'s reply, *Buyer* answers with either an acceptance or a rejection message (it sends the latter when the quote is bigger than a certain amount). In case of acceptance, *Seller* sends a confirmation to *Buyer* and asks *Shipper* to provide delivery details. Finally, *Seller* forwards the received delivery information to *Buyer*. Moreover, after *Seller* presents a quote, if *Buyer* does not reply in 30 time units, then *Seller* will abort the transaction. In the end, the whole system is

$$\{\mathit{Buyer}\} \mid \{\mathit{Seller}\} \mid \{\mathit{Shipper}\}$$

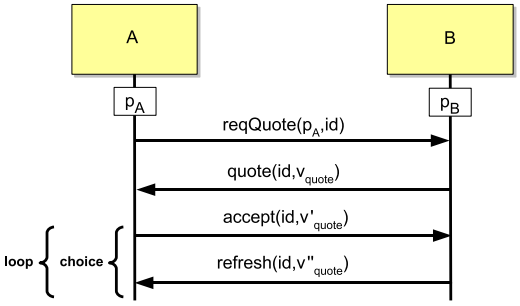

Figure 5.4: Graphical representation of the Investment Bank interaction pattern

where

$$
\begin{aligned}
\mathit{Buyer} \triangleq\ & [id]\,(\, p_S \bullet o_{reqQuote}!\langle p_B, id\rangle \\
& \quad \mid [x_{quote}]\, p_B \bullet o_{quote}?\langle id, x_{quote}\rangle\,. \\
& \quad\quad [k]\,(\, \textbf{if}\ (x_{quote} \leq 1000) \\
& \quad\quad\quad \textbf{then}\ \{\, p_S \bullet o_{accept}!\langle id\rangle \\
& \quad\quad\quad\quad \mid p_B \bullet o_{confirmation}?\langle id\rangle\,. \\
& \quad\quad\quad\quad\ [x_{det}]\, p_B \bullet o_{deliveryDet}?\langle id, x_{det}\rangle\,\} \\
& \quad\quad\quad \textbf{else}\ \{\, p_S \bullet o_{reject}!\langle id\rangle\,\} \\
& \quad\quad \mid p_B \bullet o_{abort}?\langle id\rangle\,.\,\textbf{kill}(k)\,)\,)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Seller} \triangleq\ & *\,[x_B, x_{id}]\, p_S \bullet o_{reqQuote}?\langle x_B, x_{id}\rangle\,. \\
& \quad (\, x_B \bullet o_{quote}!\langle x_{id}, v_{quote}\rangle \\
& \quad\ \mid p_S \bullet o_{accept}?\langle x_{id}\rangle\,. \\
& \quad\quad (\, x_B \bullet o_{confirmation}!\langle x_{id}\rangle \\
& \quad\quad\ \mid p_{SH} \bullet o_{reqDelivDet}!\langle x_{id}, p_S\rangle \\
& \quad\quad\ \mid [x_{det}]\, p_S \bullet o_{deliveryDet}?\langle x_{id}, x_{det}\rangle\,. \\
& \quad\quad\quad\ x_B \bullet o_{deliveryDet}!\langle x_{id}, x_{det}\rangle\,) \\
& \quad + p_S \bullet o_{reject}?\langle x_{id}\rangle \\
& \quad + \ \mathrm{clock}_{30}\,.\, x_B \bullet o_{abort}!\langle x_{id}\rangle\,)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Shipper} \triangleq\ & *\,[x_{id}, x_S]\, p_{SH} \bullet o_{reqDelivDet}?\langle x_{id}, x_S\rangle\,. \\
& \quad [x_{det}]\,[x_{det} = \mathit{computeDelivDet}(x_S)]\,.\, x_S \bullet o_{deliveryDet}!\langle x_{id}, x_{det}\rangle
\end{aligned}
$$

Function *computeDelivDet*(_) computes the delivery details associated with a seller. Notably, if *Buyer* receives an *abort* message from *Seller*, then it immediately halts its other activities, by means of the killing activity.

Investment Bank interaction pattern. We describe a typical interaction pattern in Investment Bank and other businesses, graphically represented in Figure 5.4. We consider two participants, *A* and *B*. *A* starts by requiring a quote to *B*, that answers with an initial quote. Then, *B* enters a loop, sending a new quote every 5 time units until *A* accepts a quote. Of course, in order to receive new quotes, also *A* cycles until it sends the quote

acceptance message to B. Services A and B are modelled as follows:

$$
\begin{aligned}
A \triangleq\ & p_B \bullet o_{reqQuote}!\langle p_A, id\rangle \\
& \mid [x_{quote}]\, p_A \bullet o_{quote}?\langle id, x_{quote}\rangle\,. \\
& \qquad [\mathtt{n}]\,(\,\mathtt{n}!\langle x_{quote}\rangle \\
& \qquad\quad \mid *\,[x]\,\mathtt{n}?\langle x\rangle\,. \\
& \qquad\qquad [x_{new}]\,(\,\oplus_{rand()}\,.\,p_B \bullet o_{accept}!\langle id, x\rangle \\
& \qquad\qquad\qquad + \; p_A \bullet o_{refresh}?\langle id, x_{new}\rangle\,.\,\mathtt{n}!\langle x_{new}\rangle\,)\,)
\end{aligned}
$$

$$
\begin{aligned}
B \triangleq\ & *\,[x_A, x_{id}]\, p_B \bullet o_{reqQuote}?\langle x_A, x_{id}\rangle\,. \\
& \qquad (\, x_A \bullet o_{quote}!\langle x_{id}, v_{quote}\rangle \\
& \qquad\quad \mid [\mathtt{n}]\,(\,\mathtt{n}!\langle v_{quote}\rangle \\
& \qquad\qquad \mid *\,[x]\,\mathtt{n}?\langle x\rangle\,. \\
& \qquad\qquad\quad [x_{quote}]\,(\,p_B \bullet o_{accept}?\langle x_{id}, x_{quote}\rangle \\
& \qquad\qquad\qquad + \; \oplus_5\,.\,(\, x_A \bullet o_{refresh}!\langle x_{id}, newQuote(x)\rangle \\
& \qquad\qquad\qquad\quad \mid \mathtt{n}!\langle newQuote(x)\rangle\,)\,)\,)\,)
\end{aligned}
$$

Function *newQuote*(_), given the last quote sent from B to A, computes and returns a new quote. Notably, in both services, the iterative behaviour is modelled by means of a private endpoint (i.e. n) and the replication operator. At each iteration, A waits a randomly chosen period of time, whose value is returned by function *rand*(), before replying to B. If this time interval is longer than 5 time units, a receive on operation $o_{refresh}$ triggers a new iteration.

Now, consider the system $A \mid B$. If the participant A does not accept the current quote in 5 time units, then a new quote is produced by the participant B, because its timeout has certainly expired. Instead, if we consider the system $\{A\} \mid \{B\}$, the clock of B can be slower than that of A, thus the production of a new quote is not ensured.

5.2.1.4 Concluding remarks

We have introduced C⊙WS, an extension of COWS that permits modeling timed activities and is, hence, capable of completely formalizing the semantics of WS-BPEL. We have first considered a language where all services are implicitly allocated on a same engine. Then, we have presented an extension with explicit notions of service engine and of deployment of services on engines.

For modelling time and timeouts, we have drawn again our inspiration from the rich literature on timed process calculi (see, e.g., [71, 159] for a survey). Thus, in C⊙WS, basic actions are *durationless*, i.e. instantaneous, and the passing of time is modelled by using explicit actions, like in TCCS [154]. Moreover, actions execution is *lazy*, i.e. can be delayed arbitrary long in favour of passing of time, like in lTCCS [155]. Finally, since many distributed systems offer only weak guarantees on the upper bound of inter-location clock drift [18], passing of time is modelled synchronously for services deployed on a same 'service engine', and asynchronously otherwise.

As a further work, we want to develop type systems and behavioural equivalences capable of dealing also with time related aspects. Pragmatically, they could provide a means to express and guarantee time-based QoS properties of services (such as, e.g., time to reply to service requests), that should be published in service contracts. We plan also to investigate an alternative operational semantics for C◷WS aiming at avoiding infinite branching due to time elapsing of 'non-maximal' intervals of time. This way, e.g., the term $◷_{10}.s_1 \mid ◷_6.s_1$ should evolve only to $◷_4.s_1 \mid ◷_0.s_2$, i.e. transitions to $◷_{10-6+\delta}.s_1 \mid ◷_\delta.s_2$ with $\delta > 0$ should be blocked.

5.2.2 Service publication, discovery and negotiation with COWS

In SOC, services can play essentially three different roles: the provider, the requestor and the registry. Providers offer functionalities and publish machine-readable service descriptions on registries to enable automated discover and invocation by requestors. In addition to the function that the service performs, service descriptions should also include non-functional properties, such as e.g., response time, availability, reliability, security, and performance, that jointly represent the *quality of the service* (QoS). Some of these properties could depend on the current run-time configuration of the system (e.g. the maximum allowed bandwidth might depend on the actual load of the server), thus a *dynamic discovery* process is often needed to find a provider that meets the requestors' requirements. Moreover, since services are often developed and run by different organizations, a key issue of the discovery process is to define a flexible *negotiation* mechanism that allows two or more parties to reach a joint agreement about cost and quality of a service, *prior to service execution*. This mechanism ranges from simple forms where a two-phase negotiation is sufficient (one of the two parties exposes a contract template that the other party can fill in with values in a given range) to more sophisticated forms where the parties use complex strategies and interact repeatedly. For example, if the involved parties fail to reach an agreement, their strategies can weaken the requirements and retry, or just give up the negotiation.

The outcome of the negotiation phase is a *Service Level Agreement* (SLA), i.e. a contract among the involved parties (service requestor and provider and, possibly, some third parties) that sets out both type and bounds on various performance metrics of the service to be provided, and the remedial actions to be performed if these are not met. For example, an SLA among a Web hosting provider and its customers may specify the annual cost of the service and the guaranteed bandwidth that will be provided, and the penalties to be imposed if the service fails to fulfill the guaranteed bandwidth. After the agreement has been achieved, trustworthy measurement services can possibly be used by each party to dynamically monitor that the contract is respected by the other parties.

The expansion of web services has caused the development of several new languages and technologies, among which we mention those for supporting the phases of discovery, negotiation, agreement and monitoring, like e.g. WSLA [140, 119] and WS-Agreement [8], that permit specifying and managing SLAs, WS-Negotiation [115], that permits im-

plementing automated negotiation, and [189, 186], that exploit the ontology languages DAML-S and OWL-S to enable semantic matching of service capabilities.

To provide formal foundations to current (web) services technologies, in Chapter 3 we have introduced COWS. We demonstrate now that COWS can model all the phases of the life cycle of service-oriented applications, such as publication, dynamic discovery, negotiation, deployment and execution. We are not affirming that whoever programs service-oriented applications should use COWS as the sole language. First of all, forcing to use only one language would be unrealistic and in neat contrast with the 'open-endedness' of the SOC paradigm. Moreover, COWS is a lower level modelling language rather than a full-fledged programming language. We are instead putting forward that COWS can be a common and convenient basis to enable analysis of service-oriented applications, e.g. by means of translation from higher level languages. The possibility of analysing COWS specifications by using the techniques and tools presented in Chapter 4 is certainly an important added value of using COWS for modelling services.

Technically, we exploit the fact that COWS language definition abstracts from a few sets of objects (e.g., the set of expressions that can occur within terms of the calculus) and appropriately specialize these parameters of the language so that services can specify and conclude SLAs. We follow the approach put forward in cc-pi [47], a language that combines basic features of name-passing calculi with concurrent constraint programming [184]. Specifically, we show that constraints and operations on them can be smoothly incorporated in COWS, and propose a disciplined way to model multisets of constraints and manipulate them through appropriate interaction protocols. This way, SLA requirements are expressed as constraints that can be dynamically generated and composed, and that can be used by the involved parties both for service publication and discovery (on the Web), and for the SLA negotiation process. Consistency of the set of constraints resulting from negotiation means that the agreement has been reached.

5.2.2.1 Using COWS for concurrent constraint programming

We now tailor COWS for specifying Service Level Agreements. We take advantage of the fact that its syntax and operational semantics are parametrically defined with respect to the set of *values*, the syntax of *expressions* that operate on values and, therefore, the definition of the *pattern-matching* function. We show that, by specializing these parameters, we can obtain a dialect that properly integrates the principle of 'computing with partial information', or constraints, that is at the basis of the concurrent constraint programming paradigm [184].

We first provide some insights into the constraint system used. In COWS, a *constraint* is a relation among a specified set of variables which gives some information on the set of possible values that these variables may assume. Such information is usually not complete since a constraint may be satisfied by several assignments of values to the variables. For

example, we can employ constraints such as

$$\mathsf{cost} \geqslant 350 \qquad \mathsf{cost} = \mathsf{bw} \cdot 0.05 \qquad \mathsf{z} = \frac{1}{1 + |\mathsf{x} - \mathsf{y}|}$$

In practice, we do not take a definite standing on which of the many kind of constraints to use. From time to time, the appropriate kind of constraints to work with should be chosen depending on what one intends to model.

Formally a constraint c is represented as a function $c : (V \to D) \to \{\mathbf{true}, \mathbf{false}\}$, where V is the set of constraint variables (that, as explained in the sequel, is included in the set of COWS names), and D is the domain of interpretation of V, i.e. the domain of values that the variables may assume. If we let $\eta : V \to D$ be an assignment of domain elements to variables, then a constraint is a function that, given an assignment η, returns a truth value indicating if the constraint is satisfied by η. For instance, the assignment $\{\mathsf{cost} \mapsto 500\}$ satisfies the first constraint, while $\{\mathsf{cost} \mapsto 500, \mathsf{bw} \mapsto 8000\}$ does not satisfy the second constraint, that is, instead, satisfied by $\{\mathsf{cost} \mapsto 400, \mathsf{bw} \mapsto 8000\}$. An assignment that satisfies a constraint is called a *solution*.

The constraints we have presented are called *crisp* in the literature, because they can only be satisfied or violated. In fact, we can also use more general constraints called *soft constraints* [97, 28]. These constraints, given an assignment for the variables, return an element of an arbitrary constraint semiring (*c-semiring*, [27]), namely a partially ordered set of 'preference' values equipped with two suitable operations for combination ($\times$) and comparison ($+$) of (tuples of) values and constraints. Formally, a c-semiring is an algebraic structure $\langle A, +, \times, 0, 1\rangle$ such that: A is a set and $0, 1 \in A$; $+$ is a binary operation on A that is commutative, associative, idempotent, 0 is its unit element and 1 is its absorbing element; $\times$ is a binary operation on A that is commutative, associative, distributes over $+$, 1 is its unit element and 0 is its absorbing element. Operation $+$ induces a partial order $\leq$ on A defined by $a \leq b$ iff $a + b = b$, which means that a is more constrained than b. The minimal element is thus 0 and the maximal 1. For example, crisp constraints can be understood as soft constraints on the c-semiring $\langle\{\mathbf{true}, \mathbf{false}\}, \vee, \wedge, \mathbf{false}, \mathbf{true}\rangle$ of the boolean values.

The COWS dialect we work with in the rest of the section specializes expressions to also include *constraints*, ranged over by c, and *constraint multisets*, ranged over by C, and to be formed by using the following operators.

- *Consistency check:* predicate $isCons(C)$ takes a constraint multiset C and holds true if C is consistent. Formally, $isCons(\{c_1, \ldots, c_n\})$ holds true if there exists an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \neq \mathbf{false}$, i.e. if the combination of all constraints has at least a solution[1]. The predicate $isCons(\_)$ is defined for crisp constraints. However, we can generalize its definition to soft constraints by requiring that it is satisfied if there exists an assignment η such that $c_1\eta \times \ldots \times c_n\eta \neq 0$.

[1] We do not consider here the well-studied problem of solving a constraint system. Among the many techniques exploited to this aim, we mention dynamic programming [156, 26] and branch and bound search [204].

$$\mathcal{M}(x, v) = \{x \mapsto v\} \qquad \mathcal{M}(v, v) = \emptyset \qquad \mathcal{M}(\langle\rangle, \langle\rangle) = \emptyset \qquad \frac{\mathcal{M}(a_1, b_1) = \sigma_1 \quad \mathcal{M}(\bar{a}_2, \bar{b}_2) = \sigma_2}{\mathcal{M}((a_1, \bar{a}_2), (b_1, \bar{b}_2)) = \sigma_1 \uplus \sigma_2}$$

$$\frac{\mathit{isCons}(C \uplus \{c\})}{\mathcal{M}(\langle c, x\rangle, C) = \{x \mapsto C\}} \qquad \frac{C \vdash c}{\mathcal{M}(\langle c^{\vdash}, x\rangle, C) = \{x \mapsto C\}}$$

Table 5.10: (Extended) matching rules

- *Entailment check:* predicate $C \vdash c$ takes a constraint multiset C and a constraint c and holds true if c is entailed by C. Formally, $\{c_1, \ldots, c_n\} \vdash c$ holds true if for all assignments η holds that $c_1\eta \wedge \ldots \wedge c_n\eta \leq_B c\eta$, where $\leq_B$ is the partial ordering over booleans, defined by $b_1 \leq_B b_2$ iff $b_1 \vee b_2 = b_2$. Also this predicate can be generalized to soft constraints by requiring that $\{c_1, \ldots, c_n\} \vdash c$ holds true if there exists an assignment η such that $c_1\eta \times \ldots \times c_n\eta \leq c\eta$.

- *Retraction:* operation $C - c$ takes a constraint multiset C and a constraint c and returns the multiset $C \backslash \{c\}$ if $c \in C$, otherwise returns C.

- *Multiset union:* binary operator $\uplus$ is the standard union operator between multisets.

Since constraints and constraint multisets are expressions, they need to be evaluated. The (expression) evaluation function $[\![\_]\!]$ acts on constraints and constraint multisets as the identity, except for constraints containing COWS variables, for which the function is undefined. Therefore, evaluated constraints and constraint multisets are values that can be communicated by means of synchronization of invoke and receive activities and can replace variables by means of application of substitutions to terms.

To efficiently implement the primitives of the concurrent constraint programming paradigm, we tailor the rules in Table 3.3 (Section 3.2.1) defining the pattern-matching function $\mathcal{M}(\_, \_)$ to deal with constraints and operations on them, as shown in Table 5.10. We assume that tuples can be arbitrarily nested and can be constructed using a concatenation operator defined as $\langle a_1, \ldots, a_n\rangle : \langle b_1, \ldots, b_m\rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$. To single out an element of a tuple, we will write $(\bar{a}, c, \bar{b})$ to denote the tuple $\langle a_1, \ldots, a_n, c, b_1, \ldots, b_m\rangle$, where $\bar{a}$ or $\bar{b}$ might not be present. The original matching rules (reported in the upper part of Table 5.10) are still valid and state that variables match any value (thus, e.g., $\mathcal{M}(x, C) = \{x \mapsto C\}$), two values match only if they are identical, and two tuples match if they have the same number of fields and corresponding fields do match. The new rules (shown in the lower part of the table) allow a two-field tuple to match a single value in two specific cases: a tuple $\langle c, x\rangle$ and a multiset of constraints C do match if $C \uplus \{c\}$ is consistent, while a tuple $\langle c^{\vdash}, x\rangle$ and a multiset of constraints C do match if c is entailed by C; in both cases, the substitution $\{x \mapsto C\}$ is returned. Notably, by applying the operator $\_^{\vdash}$ to a constraint one can require an entailment check instead of a consistency check.

The concurrent constraint computing model is based on a shared store of constraints

that provides partial information about possible values that variables can assume. In COWS the store of constraints is represented by the following service:

$$store_C \triangleq [\mathrm{n}]\,(\,\mathrm{n}!\langle C\rangle \mid *\,[x]\,\mathrm{n}?\langle x\rangle.\,(\,p_s \bullet o_{get}!\langle x\rangle \mid [y]\,p_s \bullet o_{set}?\langle y\rangle.\,\mathrm{n}!\langle y\rangle\,)\,)$$

where p_s is a distinguished partner, o_{get} and o_{set} are distinguished operations. Other services can interact with the store service in mutual exclusion, by acquiring the lock (and, at the same time, the stored value) with a receive along $p_s \bullet o_{get}$ and by releasing the lock (providing the new stored value) with an invoke along $p_s \bullet o_{set}$. Notably, local stores of constraints can be simply modelled by restricting the scope of the partner name p_s.

The store is composed in parallel with the other services, which can act on it by performing operations for adding/removing constraints to/from the store (tell and retract, respectively), and for checking entailment/consistency of a constraint by/with the store (ask and check, respectively). These four operations can be rendered in COWS as follows:

$$\langle\!\langle \mathsf{tell}\ c.s \rangle\!\rangle = [\mathrm{n}]\,(\,\mathrm{n}!\langle c\rangle \mid [y]\,\mathrm{n}?\langle y\rangle.\,[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.\,(\{\!|\,p_s \bullet o_{set}!\langle x \uplus \{y\}\rangle\,|\!\} \mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \mathsf{ask}\ c.s \rangle\!\rangle = [\mathrm{n}]\,(\,\mathrm{n}!\langle c^{\vdash}\rangle \mid [y]\,\mathrm{n}?\langle y\rangle.\,[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.(\{\!|\,p_s \bullet o_{set}!\langle x\rangle\,|\!\} \mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \mathsf{check}\ c.s \rangle\!\rangle = [\mathrm{n}]\,(\,\mathrm{n}!\langle c\rangle \mid [y]\,\mathrm{n}?\langle y\rangle.\,[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.\,(\{\!|\,p_s \bullet o_{set}!\langle x\rangle\,|\!\} \mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \mathsf{retract}\ c.s \rangle\!\rangle = [\mathrm{n}]\,(\,\mathrm{n}!\langle c\rangle \mid [y]\,\mathrm{n}?\langle y\rangle.\,[x]\,p_s \bullet o_{get}?\langle x\rangle.\,(\{\!|\,p_s \bullet o_{set}!\langle x - y\rangle\,|\!\} \mid \langle\!\langle s\rangle\!\rangle)\,)$$

where n is fresh. Essentially, each operation is a term that first takes the store of constraints (thus acquiring the lock so that other services cannot concurrently interact with the store) and then returns the (possibly) modified store (thus releasing the lock). Since the invoke activities $\mathrm{n}!\langle c\rangle$ and $\mathrm{n}!\langle c^{\vdash}\rangle$ can be performed only if $[\![c]\!]$ is defined, i.e. if c does not contain COWS variables, the store can only contain evaluated constraints. Availability of the store is guaranteed by the fact that, once the store and the lock have been acquired, the activities reintroducing the store and releasing the lock are protected from the effect of kill activities. This disciplined use of the store permits to preserve its consistency. Notably, the matching rules in the lower part of Table 5.10 are essential for faithfully modelling the semantics of the original operations.

While tell and ask are the classical concurrent constraint programming primitives, operations check and retract are borrowed from [47]. In particular, operation retract is debatable since its adoption prevents the store of constraints to be 'monotonically' refined. In fact, in concurrent constraint programming a computation step does not change the value of a variable, but may rule out certain values that were previously possible; therefore, the set of possible values for the variable is contained in the set of possible values at any prior step. This monotonic evolution of the store during computations permits to define the result of a computation as the least upper bound of all the stores occurring along the computation and provides concurrent constraint languages with a simple denotational semantics in which programs are identified to closure operators on the semi-lattice of constraints [183]. Therefore, if one wants to exploit some of the properties of concurrent constraint programming that require monotonicity, he must consider the fragment of

COWS without retract. On the other hand, in the context of dynamic service discovery and negotiation, the use of operation retract enables modelling many frequent situations where it is necessary to remove a constraint from the store for, e.g., weakening a request.

To avoid interference between communication and operations on the store, we do not allow constraints in the store to contain variables, thus they cannot change due to application of substitutions generated by communication. Indeed, suppose constraints in the store may contain variables and consider the following example:

$$[x]\,(\,store_{\emptyset} \mid \mathsf{tell}(x \leq 5).\,(\mathtt{n}!\langle 6\rangle \mid \mathtt{n}?\langle x\rangle)\,)$$

After action tell has added the constraint $x \leq 5$ to the store, communication along the endpoint n can modify the constraint in $6 \leq 5$. This way, the communication can make the store inconsistent. This means that the write-once variables of COWS are not suitable for modelling constraint variables.

Therefore, as we stated before, we do not allow constraints in the store to contain variables. Instead, they can use specific names, that we call *constraint variables* and, for the sake of presentation, write as $\mathsf{x}, \mathsf{y}, \ldots$ (i.e. in the sans serif style). Indeed, names are not affected by expression evaluation (i.e. $[\![\mathsf{x}]\!] = \mathsf{x}$) and by substitution application (i.e. $\mathsf{x} \cdot \sigma = \mathsf{x}$). Moreover, names can be delimited, thus allowing us to model *local constraints*. In the sequel, we will use $\mathrm{cv}(t)$ to denote the set of constraint variables occurring in a term t. Notice however that constraints occurring as arguments of operations may contain variables so that we can specify constraints that will be dynamically determined. For example, we can write tell ($\mathsf{cost} \geqslant x_{min\_cost}$).$s$; of course, since $[\![\mathsf{cost} \geqslant x_{min\_cost}]\!]$ is undefined, this operation is blocked until variable x_{min_cost} is substituted by a value.

5.2.2.2 Communication protocols for constraints generation

Besides ask, tell, retract and check, inter-service communication can be used to implement many protocols allowing two parties to generate new constraints. For instance, in [47], service synchronization works like two global ask and tell constructs: as a result of the synchronization between the output $\bar{x}\langle y\rangle$ and the input $x\langle y'\rangle$ the new constraint $y = y'$ is added to the store. Therefore, synchronization allows local constraints (i.e. constraints with restricted names) to interact, thus establishing an SLA between the two parties, and (possibly) to become globally available. Differently, COWS does not allow communication to directly generate new constraints: e.g., an invoke $p \bullet o!\langle \mathsf{x}\rangle$ and a receive $p \bullet o?\langle \mathsf{y}\rangle$ cannot synchronize, because $\mathcal{M}(\mathsf{y}, \mathsf{x})$ does not hold. In the rest of this section, we present three example protocols that permit establishing new constraints. For the sake of readability, in the protocols we will use the conditional choice construct **if** (ϵ) **then** $\{s_1\}$ **else** $\{s_2\}$ introduced in Section 3.2.1.3 (encoding (3.4)).

A simple protocol. To create constraints of the form $\mathsf{x} = \mathsf{y}$, where each of x and y is initially local to only one party, we can use the standard COWS communication mechanism

together with operation tell. For example, the following term

$$store_C \mid p \bullet o!\langle \mathsf{x} \rangle \mid [z]\, p \bullet o?\langle z \rangle.\, \mathsf{tell}\,(z = \mathsf{y}).\, s \tag{5.3}$$

for z fresh in s, adds to the store the constraint $\mathsf{x} = \mathsf{y}$, if it is consistent with C.

Indeed, the communication along endpoint $p \cdot o$ takes place before the consistency check (performed by operation tell), and the term evolves into

$$store_C \mid \mathsf{tell}\,(\mathsf{x} = \mathsf{y}).\, s$$

Now, if $\mathsf{x} = \mathsf{y}$ is not consistent with the store, the receive and invoke activities along $p \bullet o$ are definitively consumed and the execution of term s is blocked.

This protocol is simple and divergence-free, but it may introduce deadlocked states in the terms. This fact has a relevant impact on the specification of protocols for negotiation, particularly when there are more parties that provide (or require) the same service. For example, consider the following term

$$store_C \mid p \bullet o!\langle \mathsf{x} \rangle \mid [z]\, p \bullet o?\langle z \rangle.\, \mathsf{tell}\,(z = \mathsf{y}).\, s \mid [z']\, p \bullet o?\langle z' \rangle.\, \mathsf{tell}\,(z' = \mathsf{w}).\, s'$$

where another receive activity is put in parallel with term (5.3). Now, if $\mathsf{x} = \mathsf{y}$ is not consistent with C, then the term can non-deterministically evolve in a stuck state by performing the receive $p \bullet o?\langle z \rangle$, although $\mathsf{x} = \mathsf{w}$ might be consistent with C.

A divergent protocol. To overtake the previous problems, the following more refined protocol restores the communication activities if the constraints generated when communication takes place are not consistent with the current store. To simplify the encoding, we assume that a single communication cannot produce both substitutions and new constraints. The extended communication activities can be rendered as follows:

$$\langle\!\langle p \bullet o!\bar{\epsilon} \rangle\!\rangle \;=\; [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.[\mathtt{m}]\,(\,p \bullet o!(\mathtt{m}, \bar{\epsilon}) \mid \mathtt{m}?\langle\rangle.\mathtt{n}!\langle\rangle\,)\,)$$

$$\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle \;=\; \begin{cases} [x]\, p \bullet o?(x, \bar{w}).\langle\!\langle s \rangle\!\rangle & \text{if } \mathrm{cv}(\bar{w}) = \emptyset \\ s' & \text{if } \bar{w} = \langle \mathsf{x}_1, \ldots, \mathsf{x}_n \rangle \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\begin{aligned} s' \;\triangleq\; & [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.[x, x_1, \ldots, x_n]\, p \bullet o?\langle x, x_1, \ldots, x_n \rangle.[y]\, p_s \bullet o_{get}?\langle y \rangle. \\ & \quad [\mathtt{r}]\,(\,\{\!|\,\mathbf{if}\ (\mathit{isCons}(y \uplus \{\mathsf{x}_1 = x_1, \ldots, \mathsf{x}_n = x_n\})\,) \\ & \qquad \mathbf{then}\ \{\, p_s \bullet o_{set}!\langle y \uplus \{\mathsf{x}_1 = x_1, \ldots, \mathsf{x}_n = x_n\} \rangle \mid \mathtt{r}!\langle\rangle \,\} \\ & \qquad \mathbf{else}\ \{\, p_s \bullet o_{set}!\langle y \rangle \mid x!\langle\rangle \mid \mathtt{n}!\langle\rangle \,\}\,|\!\!\} \\ & \quad \mid \mathtt{r}?\langle\rangle.\langle\!\langle s \rangle\!\rangle\,)\,) \end{aligned}$$

for $\mathtt{n}$, $\mathtt{m}$ and $\mathtt{r}$ fresh in $\bar{\epsilon}$, $\bar{w}$ and s. An invoke activity is encoded as a term that performs the same invoke with, as an additional argument, a private endpoint $\mathtt{m}$ where, if communication fails, it waits for an acknowledgement that triggers the restart of the term. A receive activity with a tuple, as an argument, without constraint variables is encoded as a term

that performs the same receive with, as an additional argument, a dummy variable (i.e. x), that stores the endpoint for the acknowledgement (that, however, is not sent in this case). A receive activity with, as an argument, a tuple of constraint variables is encoded as a term that performs a receive with a tuple of COWS variables which store the endpoint for the acknowledgement and the received data (i.e. constraint variables or values). After the receive, the term takes the current store of constraints and checks its consistency with the constraints that would be generated by taking place of the communication. In the positive case, it updates the store with the new constraints and triggers the (encoding of the) continuation term s by a signal along the endpoint $\mathtt{r}$. Otherwise, it leaves unchanged the store, sends an ack back to the corresponding invoking term, and restarts its execution. As in the encodings of concurrent constraint programming primitives, in order to guarantee the release of the lock on the shared store of constraints, the activities following the acquisition of the lock are protected.

The encoded receives can be terms like s', hence they cannot be used as guards of a choice. Therefore, to implement a choice between encoded receives, they must be put in parallel and synchronized by using a lock (as in [158]).

Communication can generate constraints expressing equalities between names (alike fusions of [47]) or equalities between names and values. For example, the invoke $p \bullet o!\langle \mathsf{x} \rangle$ and the receive $p \bullet o?\langle \mathsf{y} \rangle$ can synchronize and add the constraint $\mathsf{x} = \mathsf{y}$ to the store, if consistency is preserved, otherwise the synchronization is forbidden. Similarly, the receive above can synchronize with the invoke $p \bullet o!\langle v \rangle$ and generate the constraint $\mathsf{y} = v$. Let us now consider the following term

$$store_C \mid \langle\!\langle p \bullet o!\langle \mathsf{x} \rangle \rangle\!\rangle \mid \langle\!\langle p \bullet o?\langle \mathsf{y} \rangle.s \rangle\!\rangle \mid \langle\!\langle p \bullet o?\langle \mathsf{w} \rangle.s' \rangle\!\rangle$$

and assume that $\mathsf{x} = \mathsf{w}$ is consistent with the store while $\mathsf{x} = \mathsf{y}$ is not. In this case, by performing the receive $p \bullet o?\langle \mathsf{y} \rangle$ the term will come back to the initial state, while by performing the receive $p \bullet o?\langle \mathsf{w} \rangle$ it becomes $store_{C \uplus \{\mathsf{x}=\mathsf{w}\}} \mid \langle\!\langle p \bullet o?\langle \mathsf{y} \rangle.s \rangle\!\rangle \mid \langle\!\langle s' \rangle\!\rangle$ where, for simplicity sake, we omit the stuck terms produced by the encoding.

Of course, since the protocol can diverge (i.e. an invoke can synchronize infinitely often with the same receive without modifying the store), a *fairness assumption* is essential to guarantee progress properties: if an invoke can synchronize with many receives and at least one synchronization produces consistent constraints, then eventually this synchronization will succeed.

A divergence-free protocol. To get rid of divergence, we could add the following pattern-matching rule

$$\frac{|\bar{\mathsf{x}}|=|\bar{x}|=|\bar{v}| \qquad isCons(C \uplus \{\bar{\mathsf{x}} = \bar{v}\})}{\mathcal{M}((\bar{\mathsf{x}} : \bar{x}, y), (\bar{v}, C)) = \{\bar{x} \mapsto \bar{v}, y \mapsto C\}}$$

(notice that, since the tuples $(\bar{\mathsf{x}} : \bar{x}, y)$ and $(\bar{v}, C)$ have different length, the rule does not interfere with the other ones) and encode communication activities as follows:

$$\langle\!\langle p \bullet o!\bar{\epsilon} \rangle\!\rangle = [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.\,[\mathtt{m}]\,(\,[x]\,p_s \bullet o_{get}?\langle x\rangle. \\ (\,p \bullet o!\langle(\bar{\epsilon}, x), \mathtt{m}\rangle \mid \{\!|\, p_s \bullet o_{set}!\langle x\rangle \,|\!\} \mid \mathtt{m}?\langle\rangle.\,\mathtt{n}!\langle\rangle\,)\,)\,)$$

$$\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle = \begin{cases} [z, x]\, p \bullet o?\langle(\bar{w}, z), x\rangle.\, \langle\!\langle s \rangle\!\rangle & \text{if } \mathrm{cv}(\bar{w}) = \emptyset \\ s' & \text{if } \bar{w} = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle \\ undef & \text{otherwise} \end{cases}$$

$$s' \triangleq [\mathtt{n}]\,(\,\mathtt{n}!\langle\rangle \mid *\,\mathtt{n}?\langle\rangle.[x_1, \ldots, x_n, z, x]\, p \bullet o?\langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_n, x_1, \ldots, x_n, z\rangle, x\rangle. \\ [y]\, p_s \bullet o_{get}?\langle y\rangle. \\ [\mathtt{r}]\,(\,\{\!|\, \textbf{if } (y == z) \\ \textbf{then } \{\, p_s \bullet o_{set}!\langle y \uplus \{\mathtt{x}_1 = x_1, \ldots, \mathtt{x}_n = x_n\}\rangle \mid \mathtt{r}!\langle\rangle \,\} \\ \textbf{else } \{\, p_s \bullet o_{set}!\langle y\rangle \mid x!\langle\rangle \mid \mathtt{n}!\langle\rangle \,\} \,|\!\|\, \\ \mid \mathtt{r}?\langle\rangle.\,\langle\!\langle s \rangle\!\rangle\,)\,)$$

for $\mathtt{n}$, $\mathtt{m}$ and $\mathtt{r}$ fresh in $\bar{\epsilon}$, $\bar{w}$ and s. Essentially, the encoding of the invoke reads the store and releases it, invoke and receive activities synchronize (i.e. the new constraints are consistent with the store), the encoding of the receive reads the store and, if the value is unchanged, adds the new constraints, otherwise it restarts the terms. The encoding is divergence-free in the sense that whenever the terms are restarted the value of the store of constraints differs from that in the previous execution, namely the terms *cannot stutter* on the same store. Of course, more robust protocols (that, e.g., avoid also starvation) could be defined. However, they should rely on synchronization among more than two entities at the same time (as it is permitted, e.g., by the join input of the Join-calculus [94]), that goes against our choice to reconcile expressiveness and implementability.

5.2.2.3 Examples

We show two examples that illustrate how our framework can be used to model both automatic service discovery mechanisms and negotiation mechanisms with the aim of achieving service level agreements. The former is a specification of a web hosting scenario, that is, in a simplified form, one of the typical SOC scenarios where SLA among organizations are largely employed. The latter one integrates publication, discovery and negotiation to the specification of the automotive case study presented in Section 2.4.1 and specified in COWS in Section 3.4.1.

A Web hosting scenario. We consider a scenario, inspired by [47], that involves a client C that needs a web hosting service, and some service providers P_1, P_2, ..., that offer different Web hosting solutions, varying in cost and bandwidth. In order to be invoked by clients, provider services need to be discovered. The dynamic service discovery mechanism relies on a (single) registry R that, similarly to an UDDI registry, allows providers to publish their service description, as a WSDL document, and clients to discover published services by performing search queries. Suppose that providers obtain their bandwidth resources from third parties T_1, T_2, ..., and that each provider can cover only a delimited

geographical area. The whole system results from the parallel composition of C, R, $store_{\emptyset}$ and all provider and third party services. In the following COWS specification of this scenario, we implicitly rely on the first communication protocol presented in Section 5.2.2.2.

The client service C is defined as

$$\begin{aligned} \mathrm{C} \triangleq\ & p_{\mathrm{R}} \bullet o_{search}!\langle \text{“}web\ hosting\text{”}, p_{\mathrm{C}}, (\mathsf{zip} = 10012)\rangle \\ & \mid [x]\, p_{\mathrm{C}} \bullet o_{corr}?\langle x\rangle.\, [k] \ *\ [x_{\mathrm{P}}]\, p_{\mathrm{C}} \bullet o_{resp}?\langle x_{\mathrm{P}}\rangle. \\ & \qquad\qquad (\, x_{\mathrm{P}} \bullet o_{req}!\langle p_{\mathrm{C}}\rangle \mid [z]\, p_{\mathrm{C}} \bullet o_{startNeg}?\langle z\rangle.\, \mathrm{C}^{neg}\,) \end{aligned}$$

The endpoint $p_{\mathrm{R}} \bullet o_{search}$ is used to perform a query on the registry and transmit the client's partner name p_{C}. Besides the string identifying the kind of required service, a query contains a constraint identifying the location of the client[2]. The registry will reply by sending along $p_{\mathrm{C}} \bullet o_{corr}$ a private name used to send a stop signal to the registry, and along $p_{\mathrm{C}} \bullet o_{resp}$ all partner names corresponding to providers that satisfy the query. For each of them, an instance of the client is created that starts a negotiation phase, implemented by term C^{neg}. We assume that once a negotiation succeeds, C^{neg} forces termination of all the other parallel instances (by performing **kill**(k)) and sends a signal to the registry to stop the database querying (by performing $\{\!|p_{\mathrm{R}} \bullet o_{stop}!\langle x\rangle|\!\}$). Client C issues one request at a time; in case of concurrent queries, correlation can be exploited to relate a query and its answers. The more general case of multiple clients can still be dealt with by using correlation or by relying on the (very reasonable) assumption that clients' partner names are pairwise distinct.

A service provider is defined alike the following term P

$$\begin{aligned} \mathrm{P} \triangleq\ & p_{\mathrm{R}} \bullet o_{pub}!\langle \text{“}web\ hosting\text{”}, p_{\mathrm{P}}, ((\mathsf{zip} \geqslant 10000) \wedge (\mathsf{zip} \leqslant 14905))\rangle \\ & \mid *\,[x_{\mathrm{C}}]\, p_{\mathrm{P}} \bullet o_{req}?\langle x_{\mathrm{C}}\rangle.\, [p]\,(\, x_{\mathrm{C}} \bullet o_{startNeg}!\langle p\rangle \mid p_{\mathrm{T}} \bullet o_{startNeg}!\langle p\rangle \\ & \qquad\qquad \mid [x_{\mathrm{T}}]\, p \bullet o_{third}?\langle x_{\mathrm{T}}\rangle.\, \mathrm{P}^{neg}\,) \end{aligned}$$

The endpoint $p_{\mathrm{R}} \bullet o_{pub}$ is used for invoking the registry service and transmitting the description of the provided service. This description consists of a string identifying the kind of provided service, the provider's partner name p_{P}, and a constraint that defines the area covered by the provider[3] which, of course, may differ from provider to provider. Notably, zip is a global constraint variable. Each request sent by a client, say C, triggers a new negotiation phase. Specifically, when a request arrives along the endpoint $p_{\mathrm{P}} \bullet o_{req}$, the provider creates a new instance that generates a new partner name p, that defines a private endpoint used to receive from C and from the considered third party, say T. Indeed, the provider instance sends p to C and T. After that, T replies with another private partner name (stored in x_{T}), that allows the instance of P to interact with the correct instance of T, and the provider instance continues as P^{neg}.

The registry service R is defined as

[2] A client position is expressed by a zip code. In the example, the client C is located at the Computer Science Department of the New York University.

[3] A geographical area is defined by a set of United States Postal Service zip codes. In the example, the provider P covers the whole State of New York.

$$\begin{aligned}
\mathrm{R} \triangleq\ & [\mathtt{n}]\,(\ *[x_{type}, x_{\mathrm{P}}, x_c]\, p_{\mathrm{R}} \bullet o_{pub}?\langle x_{type}, x_{\mathrm{P}}, x_c\rangle.\, \mathtt{n}!\langle x_{type}, x_{\mathrm{P}}, x_c\rangle \\
& \quad \mid *[x_{type}, x_{\mathrm{C}}, x'_c]\, p_{\mathrm{R}} \bullet o_{search}?\langle x_{type}, x_{\mathrm{C}}, x'_c\rangle. \\
& \qquad\qquad [id]\,(\ x_{\mathrm{C}} \bullet o_{corr}!\langle id\rangle \ \mid\ [p_s]\,(\ store_{\emptyset} \mid \mathsf{tell}\, x'_c.\, \mathrm{R}'\ )\ )\ ) \\
\mathrm{R}' \triangleq\ & [k]\,(\ *[x_{\mathrm{P}}, x_c]\, \mathtt{n}?\langle x_{type}, x_{\mathrm{P}}, x_c\rangle. \\
& \qquad\qquad (p_{\mathrm{R}} \bullet o_{pub}!\langle x_{type}, x_{\mathrm{P}}, x_c\rangle \mid \mathsf{check}\, x_c.\, x_{\mathrm{C}} \bullet o_{resp}!\langle x_{\mathrm{P}}\rangle) \\
& \quad \mid\ p_{\mathrm{R}} \bullet o_{stop}?\langle id\rangle.\, \mathbf{kill}(k)\ )
\end{aligned}$$

For each publication request received along the endpoint $p_{\mathrm{R}} \bullet o_{pub}$ from a provider service, the registry service emits a tuple containing the service description along the private endpoint $\mathtt{n}$. The parallel composition of these outputs represents the database of the registry service. When a client request is received along $p_{\mathrm{R}} \bullet o_{search}$, R replies by sending a new correlation identifier id, that will be used to correlate stop signals sent from the client along $p_{\mathrm{R}} \bullet o_{stop}$, and initializes a new local store by adding the constraint within the query message. Then, it cyclically reads a tuple (whose first field is the string specified by the client) from the internal database, checks if the provider constraints are consistent with the store and, in case of success, sends the provider's partner name to the client. Notably, reading a tuple in the database, in this case, consists of an input along $\mathtt{n}$ followed by an output along $p_{\mathrm{R}} \bullet o_{pub}$; this way we are guaranteed that, after being consumed, the tuple is correctly added to the database. The termination of the loop is triggered by the receiving of a signal along $p_{\mathrm{R}} \bullet o_{stop}$. It is worth noticing that database tuples are non-deterministically chosen, thus the same provider name can be sent many times. This could be avoided by refining the specification, e.g. by tagging each tuple with an identifier (stored in an additional field), that permits reading the tuples in an orderly way.

Finally, the third party T which P relies on is defined as

$$\mathrm{T} \triangleq\ *[x_{\mathrm{P}}]\, p_{\mathrm{T}} \bullet o_{startNeg}?\langle x_{\mathrm{P}}\rangle.\, [p']\,(\ x_{\mathrm{P}} \bullet o_{third}!\langle p'\rangle \mid \mathrm{T}^{neg}\ )$$

Once the discovery phase terminates, the client, the selected provider and the corresponding third party initiate the negotiation phase, in order to sign an SLA contract before the execution of the service. Notably, the success of the negotiation also depends on the resources provided by the third party.

Each party specifies its SLA requirements or guarantees: the client C imposes that 600 Euro is the maximum cost it is willing to pay for the service; the provider P indicates the minimum annual cost of 350 Euro for the service and the cost per unit of bandwidth $\mathsf{cost} = \mathsf{bw} \cdot 0.05$; and the third party T fixes the maximum bandwidth that it can supply at a rate of 10·000 Mbit/s. Thus, we have

$$\begin{aligned}
\mathrm{C}^{neg} \triangleq\ & [\mathsf{bw}', \mathsf{cost}']\, \mathsf{tell}\ (\mathsf{cost}' \leqslant 600). \\
& (\ z \bullet o_{sync}!\langle \mathsf{bw}', \mathsf{cost}'\rangle \mid p_{\mathrm{C}} \bullet o_{sign}?\langle\rangle.\, (\ z \bullet o_{ackC}!\langle\rangle \mid \mathrm{C}'\ )\ ) \\
\mathrm{C}' \triangleq\ & [x', y]\, p_{\mathrm{C}} \bullet o_{fix}?\langle x', y\rangle. \\
& \mathsf{check}\ ((x' = \mathsf{bw}') \wedge (y = \mathsf{cost}')).(\mathbf{kill}(k) \mid \{\!|\, p_{\mathrm{R}} \bullet o_{stop}!\langle x\rangle \,|\!\}\,)
\end{aligned}$$

$$\begin{aligned}
\mathrm{P}^{neg} \triangleq\ & [\mathsf{bw}, \mathsf{cost}]\, \mathsf{tell}\,(\,(\mathsf{cost} \geqslant 350) \wedge (\mathsf{cost} = \mathsf{bw} \cdot 0.05)\,)\,. \\
& p \bullet o_{sync}?\langle \mathsf{bw}, \mathsf{cost} \rangle. \\
& (x_{\mathrm{T}} \bullet o_{sync}!\langle \mathsf{bw} \rangle \mid p \bullet o_{ackT}?\langle\rangle.\,(x_{\mathrm{C}} \bullet o_{sign}!\langle\rangle \mid p \bullet o_{ackC}?\langle\rangle.\,\mathrm{P}'\,)\,)
\end{aligned}$$

$$\begin{aligned}
\mathrm{P}' \triangleq\ & p_{\mathrm{P}} \bullet o_{reqMetrics}!\langle p \rangle \mid [x_{bw}]\, p \bullet o_{metrics}?\langle x_{bw} \rangle\,. \\
& (\,x_{\mathrm{C}} \bullet o_{fix}!\langle x_{bw}, x_{bw} \cdot 0.05 \rangle \mid \mathsf{check}\,((x_{bw} \cdot 0.05 = \mathsf{cost}) \wedge (x_{bw} = \mathsf{bw}))\,)
\end{aligned}$$

$$\mathrm{T}^{neg} \triangleq [\mathsf{bw}'']\, \mathsf{tell}\,(\mathsf{bw}'' \leqslant 10{\cdot}000).\ p' \bullet o_{sync}?\langle \mathsf{bw}'' \rangle.\ x_{\mathrm{P}} \bullet o_{ackT}!\langle\rangle$$

Each party starts by adding its local constraints (i.e. constraints with restricted constraint variables) to the shared global store by performing an operation tell. Then, for sharing the local constraints, all parties synchronize each other by invoking operation o_{sync} (that each party provides). Finally, since all constraints are consistent, by communicating along $p_{\mathrm{C}} \bullet o_{sign}$ and $p \bullet o_{ackC}$, C and P sign the following contract:

$$\begin{aligned}
(\mathsf{cost} \geqslant 350) \wedge (\mathsf{cost} = \mathsf{bw} \cdot 0.05) \wedge (\mathsf{cost}' \leqslant 600) \wedge (\mathsf{bw}'' \leqslant 10{\cdot}000) \\
\wedge (\mathsf{bw} = \mathsf{bw}') \wedge (\mathsf{bw} = \mathsf{bw}'') \wedge (\mathsf{cost} = \mathsf{cost}')
\end{aligned}$$

Once the contract is signed, P invokes an internal service (along the endpoint $p_{\mathrm{P}} \bullet o_{reqMetrics}$) to obtain a run-time measurement of the bandwidth effectively supplied to the client. For simplicity sake, P's subservice performing the measurement is not explicitly represented. Then, P fixes the bandwidth, communicates it to client C (along the endpoint $p_{\mathrm{C}} \bullet o_{fix}$), and, by performing some operations check, the two parties validate the signed contract with respect to the value fixed by the provider. Afterwards, during the execution, the client service could use again operation check in a similar way to verify compliance with the SLA defined at negotiation time, by exploiting run-time data provided by some trustworthy measurement service. Finally, in case contract validation succeeds, C stops all its other instances that are concurrently performing negotiation phases, by means of a kill activity, and notifies the registry that it does not need further query results, by communicating along $p_{\mathrm{R}} \bullet o_{stop}$. Notably, COWS's prioritized semantics guarantees that only one instance of C signs a contract with the provider.

Automatic discovery and negotiation in the automotive case study. Initially, each on road service (such as e.g. garages, tow trucks, ...) has to publish its service description on a service registry. For example, assume that a garage service description consists of: a string identifying the kind of provided service, the provider's partner name, and a constraint that defines the garage location. Now, by assuming that the registry provides the operation o_{pub} by means of the partner name p_{reg}, a garage service can request the publication of its description as follows:

$$p_{reg} \bullet o_{pub}!\langle \text{“}garage\text{”}, p_{garage}, \mathsf{gps} = (4348.1143N, 1114.7206E) \rangle$$

where gps is a constraint variable.

The service registry is similar to that presented in the previous example and can be defined as

$$[o_{DB}]\,(\,*\,[x_{type}, x_p, x_c]\; p_{reg} \bullet o_{pub}?\langle x_{type}, x_p, x_c\rangle.p_{reg} \bullet o_{DB}!\langle x_{type}, x_p, x_c\rangle \;\mid\; R^{search}\,)$$

For each publication request received along the endpoint $p_{reg} \bullet o_{pub}$ from a provider service, the registry service outputs a service description along the private endpoint $p_{reg} \bullet o_{DB}$. The parallel composition of all these outputs represents the database of the registry. The subservice R^{search}, serving the searching requests, is defined as

$$\begin{aligned} R^{search} \;\triangleq\; & *\,[x_{type}, x_{client}, x_c, o_{addToList}, o_{askList}] \\ & p_{reg} \bullet o_{search}?\langle x_{type}, x_{client}, x_c\rangle.\,[p_s]\,(\,store_\emptyset \mid \mathsf{tell}\, x_c.\, R' \mid List\,) \end{aligned}$$

$$\begin{aligned} R' \;\triangleq\; & [k]\,(\,*\,[x_p, x_{const}]\; p_{reg} \bullet o_{DB}?\langle x_{type}, x_p, x_{const}\rangle. \\ & \qquad (\,\{\!|\, p_{reg} \bullet o_{pub}!\langle x_{type}, x_p, x_{const}\rangle \,|\!\}\; \mid \mathsf{check}\, x_{const}.\; p_{reg} \bullet o_{addToList}!\langle x_p\rangle\,) \\ & \quad \mid\; \oplus_\delta.\,(\,\mathbf{kill}(k) \mid \{\!|\, [x_{list}]\; p_{reg} \bullet o_{askList}?\langle x_{list}\rangle.\; x_{client} \bullet o_{resp}!\langle x_{list}\rangle \,|\!\}\,)\,) \end{aligned}$$

When a searching request is received along $p_{reg} \bullet o_{search}$, the registry service initializes a new local store (delimitation $[p_s]$ makes $store_\emptyset$ inaccessible outside of service R^{search}) by adding the constraint within the query message. Then, it cyclically reads a description (whose first field is the string specified by the client) from the internal database, checks if the provider constraints are consistent with the store and, in case of success, adds the provider's partner name to a list (by exploiting an internal service *List*, that provides operations $o_{addToList}$ and $o_{askList}$). After δ time units from the initialization of the local store[4], the loop is terminated by executing a kill activity and the current list of providers for service type x_{type} is sent to the client. Notably, reading a description in the database, in this case, consists of an input along $p_{reg} \bullet o_{DB}$ followed by an output along $p_{reg} \bullet o_{pub}$; this way we are guaranteed that, after being consumed, the description is correctly added to the database. It is worth noticing that service descriptions are non-deterministically retrieved, thus the same provider can occur in the returned list many times, similarly to the registry service presented in the previous example. Moreover, since our notion of time does not rely on the so-called 'maximal progress assumption'[5], i.e. communication does not prevent execution of timed transitions, there is no guarantee that any service at all is retrieved.

After the user's car breaks down and *Orchestrator* is triggered, the service *Discovery* of the in-vehicle platform will receive from *Orchestrator* a request containing the GPS data of the car, that it stores in x_{loc}, and a string identifying the kind of the required services (see the specification in Section 3.4.1). By exploiting the latter information, it will know that it has to search a garage, a tow truck and a rental car service. For example, the component taking care of discovering a garage service can be

$$p_{reg} \bullet o_{search}!\langle \text{“}garage\text{”}, p_{car}, dist(x_{loc}, \mathsf{gps}) < 20\,\rangle \;\mid\; [x_{garageList}]\; p_{car} \bullet o_{resp}?\langle x_{garageList}\rangle$$

where the constraint $dist(x_{loc}, \mathsf{gps}) < 20$ means that the required garages must be less than 20 km far from the stranded car's actual location.

[4] The timeout is modelled by means of the timed construct $\oplus_\delta.s$ introduced in Section 5.2.1.

[5] We refer to Section 5.2.1 for further details about the considered notion of time.

Once the discovery phase terminates and *Reasoner* communicates the best garage service to *Orchestrator*, the latter and the selected garage engage in a negotiation phase in order to sign an SLA. First, *Orchestrator* invokes the operation $o_{orderGar}$ provided by the selected garage (see *OrderGarageAndTowTruck*); then, it starts the negotiation by performing an operation tell that adds *Orchestrator*'s local constraints (i.e. constraints with restricted constraint variables) to the shared global store; finally, it synchronizes with the garage service, by invoking o_{sync}, for sharing its local constraints with it.

$$
\begin{array}{l}
[\mathsf{cost}, \mathsf{duration}] \\
\mathsf{tell}\,((\mathsf{cost} < 1500 \wedge \mathsf{duration} < 48) \vee (\mathsf{cost} < 800 \wedge \mathsf{duration} \geqslant 48))\,. \\
(\, x_{garage} \bullet o_{sync}!\langle \mathsf{cost}, \mathsf{duration} \rangle \\
\quad \mid p_{car} \bullet o_{garageOK}?\langle x_{gps}, x_{garageInfo} \rangle . \cdots + p_{car} \bullet o_{garageFail}?\langle\rangle . \cdots)
\end{array}
$$

In our example, the constraints state that for a repair in less than two days the driver is disposed to spend up to 1500 Euros, otherwise he is ready to spend less than 800 Euros.

After the synchronization with *Orchestrator*, the selected garage service tries to impose its first-rate constraint $c = ((\mathsf{cost'} > 2000 \wedge 6 < \mathsf{duration'} < 24) \vee (\mathsf{cost'} > 1500 \wedge \mathsf{duration'} \geqslant 24))$ and, if it fails to reach an agreement within δ' time units, weakens the requirements and retries with the constraint $c' = ((\mathsf{cost'} > 1700 \wedge 6 < \mathsf{duration'} < 24) \vee (\mathsf{cost'} > 1200 \wedge \mathsf{duration'} \geqslant 24))$. Both constraints are specifically generated by the garage service for the occurred engine failure, by exploiting the transmitted diagnostic data. After δ'' time units, if also the second attempt fails, it gives up the negotiation. This negotiation task is modelled as follows:

$$
\begin{array}{l}
[x_{cost}, x_{duration}, \mathsf{cost'}, \mathsf{duration'}] \\
p_{garage} \bullet o_{sync}?\langle x_{cost}, x_{duration} \rangle . \mathsf{tell}\,(x_{cost} = \mathsf{cost'} \wedge x_{duration} = \mathsf{duration'})\,. \\
\quad (\,\mathsf{tell}\; c\,.\, x_{cust} \bullet o_{garageOK}!\langle garageGPS, garageInfo \rangle \\
\quad\quad + \oplus_{\delta'} . (\,\mathsf{tell}\; c'\,.\, x_{cust} \bullet o_{garageOK}!\langle garageGPS, garageInfo \rangle \\
\quad\quad\quad + \oplus_{\delta''} . x_{cust} \bullet o_{garageFail}!\langle\rangle\,)\,)
\end{array}
$$

Notably, operations tell cannot be used as guards for the choice operator. Thus, a term like $\mathsf{tell}\; c\,.\, s \;+\; \oplus_e . s'$ should be considered as an abbreviation for

$$[p, q, o]\,(\,\mathsf{check}\; c\,.\,(p \bullet o!\langle\rangle \mid q \bullet o?\langle\rangle . \mathsf{tell}\; c\,.\, s) \;\mid\; \oplus_e . s' \;+\; p \bullet o?\langle\rangle . q \bullet o!\langle\rangle\,)$$

Intuitively, if the constraint c is consistent with the store, the timer can be stopped (i.e. communication along $p \bullet o$ makes a choice and removes the wait activity); afterward, the constraint can be added to the store, provided that other interactions that took place in the meantime do not lead to inconsistency. Otherwise, if the timeout expires, the constraint cannot be added to the store.

5.2.2.4 Other concurrent constraint programming constructs

In the previous sections, for the sake of presentation, only four operations have been defined to interact with the store of constraints. We want now to show that variants of these operations or other concurrent constraint programming constructs can be easily implemented in COWS, to model some peculiar aspects of discovery and negotiation processes.

Non-blocking operations. The operations tell c, check c and ask c are blocking operations, i.e. if the constraint c is not consistent with/entailed by the current store, the operations, and their continuation, are suspended until the constraint is consistent/entailed. Nevertheless, non-blocking variants of these operations can be defined. For example, by adding the following pattern-matching rule

$$\frac{\neg\, isCons(C \uplus \{c\})}{\mathcal{M}(\langle c^{\neg}, x\rangle, C) = \{x \mapsto C\}}$$

the non-blocking operation tell $c\,\{s_1\}\{s_2\}$ – that adds c to the store and continues as s_1, if c is consistent with the store, or otherwise continues as s_2 – can be rendered as follows:

$$\begin{aligned}\langle\!\langle \mathsf{tell}\ c\,\{s_1\}\{s_2\}\rangle\!\rangle \;=\; & [\mathtt{n}]\,(\,\mathtt{n}!\langle c, c^{\neg}\rangle \\ & \mid [y_1, y_2]\,\mathtt{n}?\langle y_1, y_2\rangle. \\ & \quad [x]\,(\,p_s \bullet o_{get}?\langle\langle y_1, x\rangle\rangle.(\{\!|\, p_s \bullet o_{set}!\langle x \uplus \{y_1\}\rangle \,|\!\} \mid \langle\!\langle s_1\rangle\!\rangle) \\ & \qquad +\; p_s \bullet o_{get}?\langle\langle y_2, x\rangle\rangle.(\{\!|\, p_s \bullet o_{set}!\langle x\rangle \,|\!\} \mid \langle\!\langle s_2\rangle\!\rangle)\,)\end{aligned}$$

This operation can be used, for example, to model a party of a negotiation that, in case its first-rate constraint is too strong to reach an agreement, weakens the requirements and retries with another constraint. For example, the term

$$\mathsf{tell}\ c_{strong}\,\{s_{strongSuccess}\}\{\,\mathsf{tell}\ c_{weak}\,\{s_{weakSuccess}\}\{s_{quit}\}\,\}$$

continues as $s_{strongSuccess}$ (resp. $s_{weakSuccess}$) if constraint c_{strong} (resp. c_{weak}) is consistent with the current store; if both attempts fail, it gives up the negotiation and continues as s_{quit}.

Getting the (best) solutions. During the negotiation phase, one is usually interested in satisfaction or violation of constraints. However, when the involved parties reach an agreement, one could be interested to obtain (one of) the best solution of the resulting multiset of constraints.

To achieve this aim, we introduce a function $getSol(C)$ that takes a constraint multiset C and, if C is consistent, returns a solution. Formally, in case of crisp constraints, $getSol(\{c_1, \ldots, c_n\})$ returns an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \neq \mathbf{false}$. Instead, in case of soft constraints, it returns one of the optimal solutions, i.e. an assignment η such that $c_1\eta \times \ldots \times c_n\eta \neq 0$ and $c_1\eta' \times \ldots \times c_n\eta' \leq c_1\eta \times \ldots \times c_n\eta$ for any η'. Like for consistency and entailment predicates, we do not consider here the problem of solving a constraint multiset and refer the interested reader to the literature (see e.g. [156, 26, 204]).

We also add the following rule to those defining the pattern-matching function:

$$\frac{getSol(C) = \eta \qquad \eta\,|_{\bar{\mathsf{x}}} = \bar{v}}{\mathcal{M}(\langle \bar{\mathsf{x}}, \bar{x}, y\rangle, C) = \{\bar{x} \mapsto \bar{v}, y \mapsto C\}}$$

Here, $\_\,|_{\bar{\mathsf{x}}}$ is a projection function that, given an assignment η, returns the tuple of values associated by η to the constraint variables $\bar{\mathsf{x}}$. Therefore, the construct $\mathsf{getSol}(\bar{\mathsf{x}}, \bar{x}).s$ – that

gets (one of) the best solution of the current store of constraints, assigns to $\bar{x}$ the values associated to $\bar{\mathsf{x}}$ and continues as s – can be rendered in COWS as follows:

$$\langle\!\langle \mathsf{getSol}(\bar{\mathsf{x}}, \bar{x}).s \rangle\!\rangle \;=\; [y]\, p_s \bullet o_{get}?\langle \bar{\mathsf{x}}, \bar{x}, y\rangle.\, (\{\!|\, p_s \bullet o_{set}!\langle y\rangle \,|\!\} \mid \langle\!\langle s \rangle\!\rangle)$$

Notably, if a variable within $\bar{x}$ is replaced before the execution of $\mathsf{getSol}(\bar{\mathsf{x}}, \bar{x})$, the pattern-matching rule above cannot be applied and, thus, the operation is stuck forever. However, this unwanted behaviour can be easily prevented by properly delimiting the variables, as in, e.g., the term $[\bar{x}]\,\mathsf{getSol}(\bar{\mathsf{x}}, \bar{x}).s$.

We now illustrate the semantics of the operation getSol by means of some examples. Suppose that the following crisp constraints are the result of a negotiation between a client and a provider:

$$c_{client} \;=\; \mathsf{cost} \leqslant 600 \qquad\qquad c_{provider} \;=\; \mathsf{cost} \geqslant 150$$

Then, any assignment that maps cost to a value between 150 and 600 is an effective solution. Thus, in this case, execution of $\mathsf{getSol}(\mathsf{cost}, x)$ has the effect of substituting x with a value between 150 and 600. As another example, consider the soft constraints

$$c'_{client} \;=\; \lfloor 600/\mathsf{cost} \rfloor \qquad\qquad c'_{provider} \;=\; \lfloor \mathsf{cost}/150 \rfloor$$

defined over the domain of interpretation [100..800] for the variable cost and returning values within the c-semiring $\langle [0..6], max, min, 0, 6\rangle$. Constraints c'_{client} and $c'_{provider}$ associate to each assignment for the variable cost an element of the c-semiring, which represents a grade of preference. For example, from the client point of view, the assignment $\{\mathsf{cost} \mapsto 500\}$ has grade of preference 1, while the assignment $\{\mathsf{cost} \mapsto 300\}$ has grade of preference 2, because (of course) the client prefers to save money. Instead, from the provider point of view, the greater the values of cost are the higher the grade of preference is. Moreover, c'_{client} states that values greater than 600 are not acceptable for the client, because the corresponding grade is 0; similarly, $c'_{provider}$ states that values lesser than 150 are not acceptable for the provider. In this case, the operation $\mathsf{getSol}(\mathsf{cost}, x)$ has the effect of substituting x with one of the best solutions of the constraint system, i.e. an assignment that produces the maximal grade of preference. For instance, the assignment $\{\mathsf{cost} \mapsto 300\}$ is one of the best solutions, indeed $min(c'_{client} \cdot \{\mathsf{cost} \mapsto 300\}, c'_{provider} \cdot \{\mathsf{cost} \mapsto 300\}) = min(2, 2) = 2$ and one can prove that 2 is the highest grade of preference for the combination of the two constraints.

Of course, more complex variants of the operation getSol could be implemented, in order to get all the (best) solutions or all the solutions with a grade better than a certain threshold.

5.2.2.5 Concluding remarks

By focussing on QoS requirement specifications and SLA achievements, we have demonstrated that COWS is a suitable formalism for modelling publication, discovery, negotiation, deployment and execution of service-oriented applications. Specifically, we have

shown that constraints and operations on them can be smoothly incorporated in COWS, and proposed a disciplined way to model multisets of constraints and manipulate them through appropriate interaction protocols. The novelty of our proposal is that all the above different key aspects of SOC are dealt with in an homogeneous and direct way by using a single linguistic formalism that already provides a number of analytical tools and techniques (see Chapter 4 and [172, 173]).

We end by touching upon more strictly related work. Most of the proposals in the literature result from the extension of some well-known process calculus with constructs to describe QoS requirements. This is, for example, the case of cc-pi [47], a calculus that generalises the explicit name 'fusions' of the pi-F calculus [206] to 'named constraints', namely constraints defined on enriched c-semiring structures. Rather than on fusions of names, COWS relies on substitutions of variables with values and can thus express also soft constraints by exploiting the simpler notion of c-semiring. Moreover, COWS permits defining local stores of constraints while cc-pi processes necessarily share one global store. A similar approach to SLAs negotiation is proposed in [10], although it is based on fuzzy sets instead of constraints and relies on three different languages, one for client requests, one for provider descriptions and one for contracts creation and revocation. SLA compliance has been also the focus of $\mathcal{K}$oS [72] and Kaos [73], two calculi designed for modelling network aware applications with located services and mobility. In both cases, QoS parameters are associated to connections and nodes of nets, and operations have a QoS value; the operational semantics ensures that systems evolve according to SLAs. All the mentioned proposals aim at specifying and concluding SLAs, while COWS permits also modelling other service-oriented aspects, such as e.g. service publication, discovery and orchestration, fault and compensation handling, service instances and interactions.

Integrations of the concurrent constraint paradigm with process calculi have also been used to define foundational formalisms for computer music languages. This is the case of the π^+-calculus [82], an extension of the (polyadic) π-calculus with constraint agents that can interact with a store of constraints by performing 'tell' and 'ask' actions. Differently from COWS, the store of constraints is not a term of the calculus, indeed the operational semantics of π^+-calculus is defined over configurations consisting of pairs of an agent and a store, and local stores are not supported.

A different approach to QoS is adopted in [172], where a stochastic extension of COWS is presented to enable quantitative reasoning about service behaviours. Specifically, COWS syntax and semantics are enriched along the lines of Markovian extensions of process calculi [101], and then probabilistic verification is carried on by using the PRISM probabilistic model checker.

There are also some other works that, differently from COWS, exploit static service discovery mechanisms. For example, [14] introduces λ^{req}, an extension of the λ-calculus with primitive constructs for call-by-contract invocation. In particular, an automatic machinery, based on a type system and a model-checking technique, constructs a viable plan for the execution of services belonging to a given orchestration. Non-functional aspects

are also included and enforced by means of a runtime security monitor. In [139], user's requests and compositions of web services are statically modelled via constraints. Finally, the calculi of contracts of [39] represent a more abstract approach for statically checking compliance between the client requirements and the service functionalities. A contract defines the possible flows of interactions of a service, but does not take into account non-functional properties and, thus, cannot be used for specifying and negotiating SLAs.

Chapter 6

Concluding remarks and future work

This thesis attempts to provide a formal account of the SOC paradigm and related technologies. To sum up, the thesis contains three main contributions:

1. we have introduced COWS, a formalism for specifying and combining services, while modelling their dynamic behaviour;

2. we have presented some methods and tools developed to analyse COWS terms: a type system to check confidentiality properties, a bisimulation-based observational semantics, a logical verification methodology to express and check functional properties, and a symbolic characterisation of the operational semantics;

3. we have discussed the descriptive power of COWS, by showing some encodings of other formal languages and by presenting two COWS dialects that permit modelling timed activities and dynamic service publication, discovery and negotiation.

We have exploited many examples and two large case studies, from automotive and financial domains, to illustrate the COWS's approach for specifying and analysing SOC applications.

As a future work, we plan to continue our programme to lay rigorous methodological foundations for specification and validation of SOC middlewares and applications by pursuing the following promising lines of research:

Development of further analysis techniques. We intend to tailor the studies on session and behavioural types developed for the π-calculus (see e.g. [112, 207, 57, 3, 124, 121, 117, 122, 114]) to COWS. In fact, these type disciplines, in the case of services, could permit to express and enforce many relevant policies for, e.g., regulating resources usage, constraining the sequences of messages accepted by services, ensuring service interoperability and compositionality, guaranteeing absence of deadlock in service composition, checking that interaction obeys a given protocol.

Moreover, we aim at identifying appropriate sets of sound and general equational laws that can facilitate the task of analysing SOC systems through the semantic theories introduced in Section 4.3. We also plan to develop efficient symbolic characterisations of the labelled bisimilarities over the symbolic operational semantics for COWS introduced in Section 4.4.

We are considering developing a formal account of COWS's expressiveness, by proving the goodness of the encodings proposed in this thesis and by studying the expressive power of prioritized constructs and primitives for dealing with termination.

We plan also to formalize the relationships between COWS and UML4SOA [141], the high-level formalism used for specifying the case studies presented in Chapter 2. In particular, we intend to define a 'compositional' translation from UML4SOA to COWS (see [12] for a preliminary work in this direction) and develop an automatic translator using Java-based technologies. The output should be COWS specifications that can be accepted in input by the CMC model checker; this way, it should be possible to easily check properties of graphical specifications of services by using COWS as an intermediate step.

Finally, we would like to apply our analysis techniques to verify properties of security protocols for web service conversation, such as WS-SecureConversation [162] and WS-Security [161].

Development of prototype implementations. Besides the foundational aspects, we believe that prototype implementations of COWS could be also important to assess its practical usability and to minimize the gap between theory and practice. The implementation of a language based on a process calculus typically consists of a run-time system (a sort of abstract machine) implemented in a high level language like Java, and of a compiler that, given a program written in the programming language based on the calculus, produces code that uses the run-time system above. Consider as an example the experimental language X-KLAIM, based on the calculus KLAIM [74], designed for modelling network aware applications with located services and mobility. Its implementation relies on KLAVA [20, 22], a Java package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs in Java programs that use KLAVA. We plan to follow a similar approach, by also exploiting a generic Java framework called IMC (*Implementing Mobile Calculi*, [19]) that provides recurrent mechanisms for network applications and, hence, can be used as a kind of middleware for the implementation of different process calculi. In fact, in [21], IMC has been successfully used to implement in Java the service-oriented calculus CaSPiS. Among other implementations of service-oriented calculi, we want to mention: JOLIE [157], an interpreter written in Java for a programming language designed for web service orchestration and based on SOCK [104]; JSCL (*Java Signal Core Layer*, [88, 89]), a coordination middleware for services based on the event notification paradigm of Signal Calculus

[87]; and PiDuce [64], a distributed run-time environment devised for experimenting web services technologies that implements a variant of asynchronous π-calculus extended with native XML values, datatypes and patterns.
Moreover, we intend to implement the alternative symbolic operational semantics defined in Section 4.4, by following the approach underlying CMC [192]. More specifically, we aim at realizing an interpreter capable to derive all computations originating from a COWS term in an automated way. Afterwards, building upon this tool, we would implement a modular framework composed of translators from high-level languages (e.g. WS-BPEL, UML4SOA, SRML) to COWS and of a model checker that overcomes the compositionality limitations of the approach presented in Section 4.2.

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of 27th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 104–115. ACM Press, 2001.

[2] J. Abreu, L. Bocchi, J. Fiadeiro, and A. Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In *Proc. of 27th International Conference on Formal Methods for Networked and Distributed Systems (FORTE)*, volume 4574 of *Lecture Notes in Computer Science*, pages 358–373. Springer, 2007.

[3] L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 642–658. Springer, 2008.

[4] Active Endpoints. ActiveBPEL 5.0.2, May 2008. Available at `http://www.activevos.com/community-open-source.php`.

[5] M. Alessandrini. Finance case study - Scenario descriptions, 2007. Sensoria report.

[6] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2004.

[7] R.M. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous pi-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[8] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Grid Resource Allocation Agreement Protocol (GRAAP) WG, 2007. Available at `http://www.ogf.org`.

[9] Apache Software Foundation. Apache ODE 1.2, July 2008. Available at `http://ode.apache.org`.

[10] D. Bacciu, A. Botta, and H. Melgratti. A fuzzy approach for negotiating quality of services. In *Proc. of 2nd International Symposium on Trustworthy Global Computing (TGC)*, volume 4661 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2006.

[11] F. Banti, A. Lapadula, R. Pugliese, and F. Tiezzi. Specification and analysis of SOC systems using COWS: A finance case study. In *Proc. of 4th International Workshop on Automated Specification and Verification of Web Systems (WWV'08)*, volume 235 of *Electronic Notes in Theoretical Computer Science*, pages 71–105. Elsevier, 2009.

[12] F. Banti, A. Lapadula, R. Pugliese, and F. Tiezzi. Towards a Framework for the Verification of UML Models of Services. Technical report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2009. Available at `http://rap.dsi.unifi.it/cows`.

[13] M. Bartoletti, P. Degano, and G. Ferrari. Plans for Service Composition. In *Proc. of 6th International Workshop on Issues in the Theory os Security (WITS)*, volume 4691 of *Lecture Notes in Computer Science*. Springer, 2006.

[14] M. Bartoletti, P. Degano, and G. Ferrari. Security Issues in Service Composition. In *Proc. of 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2006.

[15] M. Bartoletti, P. Degano, and G. Ferrari. Types and Effects for Secure Service Orchestration. In *Proc. of 19th Computer Security Foundations Workshop (CSFW)*, pages 35–40. IEEE Computer Society Press, 2006.

[16] J. Bauer, F. Nielson, H.R. Nielson, and H. Pilegaard. Relational Analysis of Correlation. In *Proc. of 15th International Static Analysis Symposium*, volume 5079 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2008.

[17] M. Beisiegel, H. Blohm, D. Booz, J. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff. Service Component Architecture. Building Systems using a Service Oriented Architecture. Technical report, SCA Consortium, 2005. Available at `http://www.oracle.com/technology/tech/webservices/standards/sca/pdf/SCA_White_Paper1_09.pdf`.

[18] M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous pi-Calculi. In *Proc. of 15th International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2004.

[19] L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *Proc. of 5th International Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume 3543 of *Lecture Notes in Computer Science*, pages 181–193. Springer, 2005.

[20] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-Klaim. In *Proc. of 7th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115. IEEE Computer Society Press, 1998.

[21] L. Bettini, R. De Nicola, M. Lacoste, and M. Loreti. Implementing Session Centered Calculi. In *Proc. of 10th international conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2008.

[22] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[23] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proc. of 10th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 388–397. IEEE Computer Society Press, 1995.

[24] G. Bhat, R. Cleaveland, and G. Lüttgen. A Practical Approach to Implementing Real-Time Semantics. *Annals of Software Engineering*, 7:127–155, 1999.

[25] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[26] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*. Lecture Notes in Computer Science. Springer, 2004.

[27] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[28] S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Transactions on Computational Logic*, 7(3):563–589, 2006.

[29] L. Bocchi, A. Fantechi, L. Gönczy, and N. Koch. Prototype language for service modelling: SOA Ontology in structured natural language, 2006. Sensoria deliverable D1.1a.

[30] L. Bocchi, J.L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From Architectural to Behavioural Specification of Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows/`.

[31] L. Bocchi, J.L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From Architectural to Behavioural Specification of Services. In *Proc. of 6th International*

Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), Electronic Notes in Theoretical Computer Science. Elsevier, 2009. To appear.

[32] L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. In *Proc. of 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[33] F. Bonchi and U. Montanari. Symbolic Semantics Revisited. In *Proc. of 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 4962 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2008.

[34] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *Proc. of 3rd International Workshop on Web Services and Formal Methods (WS-FM))*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[35] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *Proc. of 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2008.

[36] M. Boreale and R. De Nicola. A Symbolic Semantics for the pi-Calculus. *Information and Computation*, 126(1):34–52, 1996.

[37] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, Sophia Antipolis, 1991.

[38] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.2, W3C recommendation, 24 june 2003. Available at `http://www.w3.org/TR/SOAP/`.

[39] M. Bravetti and G. Zavattaro. Contract Based Multi-party Service Composition. In *Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN)*, volume 4767 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.

[40] M. Bravetti and G. Zavattaro. A Theory for Strong Service Compliance. In *Proc. of 9th International Conference on Coordination Models and Languages (COORDINATION)*, volume 4467 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2007.

[41] M. Bravetti and G. Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.

[42] P. Brémond-Grégoire, S.B. Davidson, and I. Lee. CCSR92: Calculus for Communicating Shared Resources with Dynamic Priorities. In *Proc. of 1st North American Process Algebra Workshop (NAPAW)*, Workshops in Computing, pages 65–85. Springer, 1993.

[43] A. Brown, S. Johnston, and K. Kelly. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Technical report, Rational Software Corporation, 2003.

[44] R. Bruni, M. Butler, C. Ferreira, T. Hoare, H.C. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proc. of 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.

[45] R. Bruni, I. Lanese, H.C. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In *Proc. of 10th international conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[46] R. Bruni, H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of 32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 209–220. ACM Press, 2005.

[47] M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.

[48] M. Buscemi and U. Montanari. Open Bisimulation for the concurrent Constraint Pi-Calculus. In *Proc. of 17th European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2008.

[49] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration: A Synergic Approach for System Design. In *Proc. of 3rd International Conference on Service-Oriented Computing (ICSOC)*, volume 3826 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2005.

[50] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of 8th international conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.

[51] M.J. Butler and C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In *Proc. of 6th international conference on Coordination Models and Languages (COORDINATION)*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.

[52] M.J. Butler, C.A.R. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2005.

[53] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). In *Proc. of 13th International Conference on Concurrency Theory (CONCUR)*, volume 2421 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2002.

[54] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[55] J. Camilleri and G. Winskel. CCS with Priority Choice. *Information and Computation*, 116(1):26–37, 1995.

[56] M. Carbone, K. Honda, and N. Yoshida. A Calculus of Global Interaction based on Session Types. In *Proc. of 2nd International Workshop on Developments on Computational Models(DCM)*, volume 171 of *Electronic Notes in Theoretical Computer Science*, pages 127–151. Elsevier, 2007.

[57] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[58] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. Structured Global Programming for Communication Behaviour. To appear as W3C working note, 2006.

[59] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. Technical report, W3C, 2006.

[60] M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[61] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[62] L. Cardelli, G. Ghelli, and A.D. Gordon. Types for the Ambient Calculus. *Information and Computation*, 177(2):160–194, 2002.

[63] S. Carpineti and C. Laneve. A Basic Contract Language for Web Services. In *Proc. of 15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2006.

[64] S. Carpineti, C. Laneve, and L. Padovani. PiDuce - a project for experimenting Web services technologies. Submitted to Science of Computer Programming. Available at `http://www.cs.unibo.it/PiDuce/`, 2006.

[65] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. General session types. Unpublished. Available at `http://www.sti.uniurb.it/padovani/publications.html`, 2008.

[66] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001. Available at `http://www.w3.org/TR/wsdl/`.

[67] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[68] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[69] R. Cleaveland, G. Lüttgen, and V. Natarajan. Priorities in process algebra. *Handbook of Process Algebra, chapter 12*, pages 391–424, 2001.

[70] Microsoft Corporation. Microsoft biztalk server. Web site: `http://www.microsoft.com/biztalk/`.

[71] F. Corradini, D. D'Ortenzio, and P. Inverardi. On the Relationships among four Timed Process Algebras. *Fundamenta Informaticae*, 38(4):377–395, 1999.

[72] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Formal Basis for Reasoning on Programmable QoS. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 436–479. Springer, 2003.

[73] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Process Calculus for QoS-Aware Applications. In *Proc. of 7th international conference on Coordination Models and Languages (COORDINATION)*, volume 3454 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2005.

[74] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.

[75] R. De Nicola, D. Gorla, and R. Pugliese. Confining Data and Processes in Global Computing Applications. *Science of Computer Programming*, 63:57–87, 2006.

[76] R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. *Information and Computation*, 205(10):1491–1525, 2007.

[77] R. De Nicola and M. Hennessy. Testing Equivalence for Processes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560. Springer, 1983.

[78] R. De Nicola, J.P. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.

[79] R. De Nicola and M. Loreti. A modal logic for mobile agents. *ACM Transactions on Computational Logic*, 5(1):79–128, 2004.

[80] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[81] R. De Nicola and F.W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proc. of the Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[82] J.F. Díaz, C. Rueda, and F.D. Valencia. π^+-calculus: A Calculus for Concurrent Processes with Constraints. *CLEI electronic journal*, 1(2), 1998.

[83] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *Proc. of Fundamental Approaches to Software Engineering (FASE)*, volume 4961 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2008.

[84] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A Logical Verification Methodology for Service-Oriented Computing. Technical report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows`.

[85] H. Fecher. A Real-Time Process Algebra with Open Intervals and Maximal Progress. *Nordic Journal of Computing*, 8(3):346–365, 2001.

[86] J. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-The-Fly Verification Techniques for the Generation of test Suites. In *Proc. of 8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1996.

[87] G. Ferrari, R. Guanciale, and D. Strollo. Event based service coordination over dynamic and heterogeneous networks. In *Proc. of 4th International Conference on Service Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2006.

[88] G. Ferrari, R. Guanciale, and D. Strollo. JSCL: A middleware for service coordination. In *Proc. of 26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE)*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2006.

[89] G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto. Event-Based Service Coordination. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 2008.

[90] J. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. In *Proc. of 3rd International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.

[91] J.L. Fiadeiro, A. Lopes, and L. Bocchi. Semantics of Service-Oriented System Configuration. Technical report, University of Leicester, 2008. Available at `http://www.cs.le.ac.uk/people/jfiadeiro/`.

[92] L.C. Filipe, I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Behavioural Theory at Work: Program Transformations in a Service-Centred Calculus. In *Proc. of 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2008.

[93] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.

[94] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. of 23rd Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385. ACM Press, 1996.

[95] C. Fournet and G. Gonthier. A Hierarchy of Equivalences for Asynchronous Calculi. In *Proc. of 25th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 844–855. Springer, 1998.

[96] A. Francalanza and M. Hennessy. A Theory of System Fault Tolerance. In *Proc. of 9th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2006.

[97] E.C. Freuder and R.J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[98] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the ACM Special Interest Group on Management of Data Annual Conference (SIGMOD)*, pages 249–259. ACM Press, 1987.

[99] P. Gardner, C. Laneve, and L. Wischik. Linear Forwarders. In *Proc. of 14th International Conference on Concurrency Theory (CONCUR)*, volume 2761 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 2003.

[100] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[101] S.T. Gilmore and M. Tribastone. Evaluating the Scalability of a Web Service-Based Distributed e-Learning and Course Management system. In *Proc. of 3rd International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4184 of *Lecture Notes in Computer Science*, pages 214–226. Springer, 2006.

[102] H. Goguen. Typed operational semantics. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 1995.

[103] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core. Technical report, W3C, May 2006. W3C Recommendation.

[104] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proc. of 4th International Conference on Service Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[105] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[106] M. Hennessy and X. Liu. A Modal Logic for Message Passing Processes. *Acta Informatica*, 32(4):375–393, 1995.

[107] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[108] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

[109] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proc. of 3rd International Conference on Business Process Management (BPM)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2005.

[110] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[111] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proc. of 5th European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[112] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of 7th European Symposium on Programming (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[113] K. Honda and N. Yoshida. On Reduction-Based Process Semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[114] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM Press, 2008.

[115] P.C.K. Hung, H. Li, and J. Jeng. WS-Negotiation: An Overview of Research Issues. In *Proc. of 37th Hawaii International Conference on System Sciences (HICSS)*, volume 01. IEEE Computer Society Press, 2004.

[116] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[117] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[118] N. Kavantzas, D. Burdett, and G. Ritzinger. Web Services Choreography Description Language version 1.0. Technical report, W3C, 2004. Available at `http://www.w3.org/TR/ws-cdl-10/`.

[119] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management, Special Issue on E-Business Management*, 11(1), 2003.

[120] Z.D. Kirli. Confined mobile functions. In *Proc. of 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 283–294. IEEE Computer Society Press, 2001.

[121] N. Kobayashi. Type Systems for Concurrent Programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2003.

[122] N. Kobayashi, K. Suenaga, and L. Wischik. Resource Usage Analysis for the π-calculus. In *Proc. of 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2006.

[123] N. Koch. Automotive case study: UML specification of on road assistance scenario, 2007. Sensoria report.

[124] I. Lanese, F. Martins, A. Ravara, and V.T. Vasconcelos. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *Proc, of 5th International Conference on Software Engineering and Formal Methods (SEFM)*, pages 305–314. IEEE Computer Society Press, 2007.

[125] C. Laneve and L. Padovani. Smooth Orchestrators. In *Proc. of 9th International Conference Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2006.

[126] C. Laneve and L. Padovani. The *must* Preorder Revisited. In *Proc. of 18th International Conference on Concurrency Theory (CONCUR)*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.

[127] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proc. of 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[128] C. Laneve and G. Zavattaro. web-pi at Work. In *Prof. of 1st International Symposium on Trustworthy Global Computing (TGC)*, volume 3705 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2005.

[129] A. Lapadula. *A Formal Account of Web Services Orchestration*. PhD Thesis in Computer Science, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows`.

[130] A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *Proc. of 8th international conference on Coordination Models and Languages (COORDINATION)*, volume 4038 of *Lecture Notes in Computer Science*, pages 145–163. Springer, 2006.

[131] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[132] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN)*, volume 4767 of *Lecture Notes in Computer Science*, pages 223–239. Springer, 2007.

[133] A. Lapadula, R. Pugliese, and F. Tiezzi. C⊙WS: A timed service-oriented calculus. In *Proc. of 4th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 4711 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2007.

[134] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows`.

[135] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *Proc. of 10th international conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008.

[136] A. Lapadula, R. Pugliese, and F. Tiezzi. Service discovery and negotiation with COWS. In *Proc. of 3rd International Workshop on Automated Specification and Verification of Web Systems (WWV)*, volume 200 of *Electronic Notes in Theoretical Computer Science*, pages 133–154. Elsevier, 2008.

[137] A. Lapadula, R. Pugliese, and F. Tiezzi. Specifying and Analysing SOC Applications with COWS. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 701–720. Springer, 2008.

[138] A. Lapadula, R. Pugliese, and F. Tiezzi. Towards modelling WS-BPEL using ws-calculus. Technical report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows`.

[139] A. Lazovik, M. Aiello, and R. Gennari. Encoding Requests to Web Service Compositions as Constraints. In *Proc. of 11th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3709 of *Lecture Notes in Computer Science*, pages 782–786. Springer, 2005.

[140] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck. WSLA Language Specification Version 1.0. Technical report, IBM Corporation, 2003. Available at `http://www.research.ibm.com/wsla`.

[141] P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *SCC*, volume 2, pages 533–536. IEEE Computer Society Press, 2008.

[142] P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *Proc. of 5th International Conference on Services Computing (SCC)*, volume 2, pages 533–536. IEEE Computer Society Press, 2008.

[143] M. Mazzara and I. Lanese. Towards a Unifying Theory for Web Services Composition. In *Proc. of 3rd International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4184 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2006.

[144] M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.

[145] R. Meolic, T. Kapus, and Z. Brezocnik. ACTLW - an Action-based Computation Tree Logic With Unless Operator. *Elsevier Information Sciences*, 178(6):1542–1557, 2008.

[146] L.G. Meredith and S. Bjorg. Contracts and types. *Communications of the ACM*, 46(10):41–47, 2003.

[147] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[148] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[149] R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[150] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[151] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[152] R. Milner and D. Sangiorgi. Barbed Bisimulation. In *Proc. of 19th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

[153] J. Misra and W.R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.

[154] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proc. of 1st International Conference on Concurrency Theory (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer, 1990.

[155] F. Moller and C. Tofts. Relating Processes With Respect to Speed. In *Proc. of 2nd International Conference on Concurrency Theory (CONCUR)*, volume 527 of *Lecture Notes in Computer Science*, pages 424–438. Springer, 1991.

[156] U. Montanari and F. Rossi. Constraint Relaxation may be Perfect. *Artificial Intelligence*, 48(2):143–170, 1991.

[157] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *Proc. of 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord)*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 19–33. Elsevier, 2007.

[158] U. Nestmann and B.C. Pierce. Decoding Choice Encodings. In *Proc. of 7th International Conference on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 1996.

[159] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In *Proc. of 3rd International Workshop on Computer Aided Verification (CAV)*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer, 1991.

[160] F. Nielson, H.R. Nielson, and C.L. Hankin. *Principles of Program Analysis*. Springer, 1999. Second printing, 2005.

[161] OASIS Web Service Security (WSS) TC. Web Services Security. Technical report, OASIS, February 2006. Available at `http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html`.

[162] OASIS Web Services Secure Exchange TC. WS-SecureConversation 1.3. Technical report, OASIS, March 2007. Available at `http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html`.

[163] OASIS WS-Reliability TC. Web Services Reliable Messaging (WS-Reliability) 1.1. Technical report, OASIS, 2004. Available at `http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/`.

[164] OASIS WS-Security TC. Web Services Security (WS-Security). Technical report, OASIS, 2006. Available at `http://docs.oasis-open.org/wss/v1.1/`.

[165] OASIS WS-TX TC . Web Services Transaction (WS-TX). Technical report, OASIS, 2006. Available at `http://docs.oasis-open.org/ws-tx/wscoor/`.

[166] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[167] Oracle. Oracle BPEL Process Manager 10.1.3, December 2007. Available at `http://www.oracle.com/technology/bpel`.

[168] J. Parrow and B. Victor. The update calculus. In *Proc. of 6th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 1349 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1997.

[169] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of 13th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185. IEEE Computer Society Press, 1998.

[170] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[171] Iain Phillips. CCS with priority guards. *Journal of Logic and Algebraic Programming*, 75(1):139–165, 2008.

[172] D. Prandi and P. Quaglia. Stochastic COWS. In *Proc. of 5th International Conference on Service Oriented Computing (ICSOC)*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.

[173] D. Prandi, P. Quaglia, and N. Zannone. Formal analysis of BPMN via a translation into COWS. In *Proc. of 10th international conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2008.

[174] R. Pugliese, F. Tiezzi, and N. Yoshida. On observing dynamic prioritised actions in SOC. In *Proc. of 36th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[175] R. Pugliese, F. Tiezzi, and N. Yoshida. A Symbolic Semantics for a Calculus for Service-Oriented Computing. In *Proc. of 1st Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2009. To appear.

[176] F. Puhlmann. Why do we actually need the pi-calculus for business process management? In *Proc. of 9th International Conference on Business Information Systems(BIS)*, volume 85 of *LNI*, pages 77–89. GI, 2006.

[177] F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. In *Proc. of 3rd International Conference Business Process Management (BPM)*, volume 3649, pages 153–168, 2005.

[178] S. Ross-Talbot and T. Fletcher. Web services choreography description language: Primer (working draft). Technical report, W3C, June 2006.

[179] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPM Center Report, 2006. `http://www.bpmcenter.org`.

[180] D. Sangiorgi. A Theory of Bisimulation for the pi-Calculus. *Acta Informatica*, 33(1):69–97, 1996.

[181] D. Sangiorgi and D. Walker. On Barbed Equivalences in pi-Calculus. In *Proc. of 12th International Conference on Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2001.

[182] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[183] V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. of 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 333–352. ACM Press, 1991.

[184] V.A. Saraswat and M.C. Rinard. Concurrent Constraint Programming. In *Proc. of 17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–245. ACM Press, 1990.

[185] SENSORIA. Software engineering for service-oriented overlay computers. Web site: `http://www.sensoria-ist.eu/`.

[186] N. Srinivasan, M. Paolucci, and K. Sycara. Semantic Web Service Discovery in the OWL-S IDE. In *Proc. of 39th Hawaii International International Conference on Systems Science (HICSS)*, volume 6. IEEE Computer Society Press, 2006.

[187] C. Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.

[188] C. Stirling and D. Walker. Local Model Checking in the Modal μ-calculus. In *Proc. of 3rd International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 354 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 1989.

[189] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of Semantic Web services. *Journal of Web Semantics*, 1(1):27–46, 2003.

[190] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *Proc. of 13th International Workshop on Formal Methods for Industrial Critical Systems(FMICS)*, volume 4916 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2008.

[191] M.H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti. Formal verification of an automotive scenario in service-oriented computing. In *Proc. of 30th International Conference on Software Engineering (ICSE)*, pages 613–622. ACM Press, 2008.

[192] M.H. ter Beek, S. Gnesi, and F. Mazzanti. CMC-UMC: A framework for the verification of abstract service-oriented properties. In *Proc. of 24th Annual ACM Symposium on Applied Computing (SAC)*, 2009. To appear.

[193] S. Thatte. XLANG: Web Services for Business Process Design. Technical report, Microsoft, 2001.

[194] UDDI Spec TC. UDDI Specification Technical Committee Draft. Technical report, OASIS, October 2004. Available at `http://uddi.org/pubs/uddi_v3.htm/`.

[195] F. van Breugel and M. Koshkina. Models and verification of BPEL. Technical report, Department of Computer Science and Engineering, York University, 2006. Available at `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`.

[196] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[197] R.J. van Glabbeek. On Specifying Timeouts. In *Proc. of the Workshop Essays on Algebraic Process Calculi (APC 25)*, volume 162 of *Electronic Notes in Theoretical Computer Science*, pages 173–175. Elsevier, 2006.

[198] B. Victor. Symbolic Characterizations and Algorithms for Hyperequivalence. Technical Report DoCS 98/96, Uppsala University, December 1998.

[199] H.T. Vieira, L. Caires, and J. Costa Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *Proc. of 17th European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[200] M. Viroli. Towards a Formal Foundation to Orchestration Languages. In *Proc. of 1st International Workshop on Web Services and Formal Methods (WSFM)*, volume 105 of *Electronic Notes in Theoretical Computer Science*, pages 51–71. Elsevier, 2004.

[201] J. Vitek and B. Bokowski. Confined Types in Java. *Software: Practice and Experience*, 31(6):507–532, 2001.

[202] M. Wirsing, L. Bocchi, A. Clark, J.L. Fiadeiro, S. Gilmore, M. Hölzl, N. Koch, and R. Pugliese. *SENSORIA: Engineering for Service-Oriented Overlay Computers*. MIT Press, 2009. To appear.

[203] M. Wirsing, A. Clark, S. Gilmore, M. Holzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-Based Development of Service-Oriented Systems. In *Proc. of 26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE)*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2006.

[204] M. Wirsing, G. Denker, C. Talcott, A. Poggio, and L. Briesemeister. A Rewriting Logic Framework for Soft Constraints. In *Proc. of 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 181–197. Elsevier, 2007.

[205] M. Wirsing, M. Hölzl, L. Acciai, F. Banti, A. Clark, A. Fantechi, S. Gilmore, S. Gnesi, L. Gönczy, N. Koch, A. Lapadula, P. Mayer, F. Mazzanti, R. Pugliese, A. Schroeder, F. Tiezzi, M. Tribastone, and D. Varró. SensoriaPatterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In *Proc. of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 17 of *Communications in Computer and Information Science*, pages 170–190. Springer, 2008.

[206] L. Wischik and P. Gardner. Explicit fusions. *Theoretical Computer Science*, 340(3):606–630, 2005.

[207] N. Yoshida and V.T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. In *Proc. of 1st International Workshop on Security and Rewriting Techniques (SecReT)*, volume 171 of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier, 2006.

[208] T. Zhao, J. Palsber, and J. Vitek. Lightweight confinement for featherweight java. In *Proc. of 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)*, pages 135–148. ACM Press, 2003.

Appendix A

CMC specification of the case studies and their properties

We report here the 'machine readable' syntax of CMC and the complete specification of the automotive and finance case studies, together with the SocL formulation of their properties we have checked, written using such syntax.

A.1 Syntax accepted by CMC

The syntax accepted by CMC is presented in Table A.1. *Killer labels* (ranged over by `k`, `k'`, ...) start with lower case letters, and can only be used as argument of kill activities; *variables* (ranged over by `X`, `Y`, ...) start with capital letters; *service identifiers* (ranged over by `A`, `A'`, ...) start with capital letters and each of them has a fixed non-negative arity; *names* (ranged over by `n`, `m`,...,`p`,`p'`,...,`o`,`o'`, ...) start with lower case letters; *values* (ranged over by `v`, `v'`, ...) are either integer numbers, booleans, or names; *identifiers* (ranged over by `u`, `u'`, ...) are either variables or names. The arguments of a receive-guarded choice must be receive activities. The expression operators + and = are defined as follows: if both e_1 and e_2 are evaluated as integer numbers then the evaluation of $e_1 + e_2$ returns the integer number corresponding to their sum, otherwise it returns the name corresponding to their concatenation; if both e_1 and e_2 are evaluated as values then the evaluation of $e_1 = e_2$ returns the boolean `true` if these values are the same value, otherwise it returns the boolean `false`.

The let construct permits to re-use the same 'service code', thus allowing to define services in a modular style; `let` `A`(*fparams*) $=s$ `in` s' `end` behaves like s', where calls to `A` can occur. A service call `A`(*aparams*) occurring in the body s' of a construct `let` `A`(*fparams*) $=s$ `in` s' `end` behaves like the service obtained from s by replacing the formal parameters *fparams* with the corresponding actual parameters *aparams*.

The syntax accepted by CMC for expressing SocL formulae slightly differs from that presented in Section 4.2.1 mainly for what concern the notation to indicate correlation

s	::=		(services)
		`nil`	(empty activity)
	|	`kill(k)`	(kill)
	|	`u.u'!` <*args*>	(invoke)
	|	`p.o?` <*params*> . s	(receive)
	|	$s_1 + \ldots + s_n$	(receive-guarded choice)
	|	$s_1 \mid s_2$	(parallel composition)
	|	$\{ s \}$	(protection)
	|	[n♯] s	(name delimitation)
	|	[k] s	(kill delimitation)
	|	[X] s	(variable delimitation)
	|	$* s$	(replication)
	|	A(*aparams*)	(call)
	|	`let` A(*fparams*) =s `in` s' `end`	(let construct)
e	::=	X | v | $e_1 + e_2$ | $e_1 = e_2$	(expressions)
args	::=	e | *args*, *args*	(invoke arguments)
params	::=	X | v	(receive parameters)
	|	*params*, *params*	
fparams	::=	X | n | k	(formal parameters)
	|	*fparams*, *fparams*	
aparams	::=	X | v | k	(actual parameters)
	|	*aparams*, *aparams*	

Table A.1: CMC syntax

variables. In fact, given a variable `var`, its binding occurrence (i.e. `var` in SocL) is written `$var`, while its free occurrences are written `%var`. Moreover, logical operators $\vee$ and $\neg$ are written `or` and `not`, respectively.

A.2 Automotive case study

The complete specification of the automotive case study written in the CMC syntax is as follows. It is worth noticing that CMC requires that the abstraction rules to apply to a specification be provided together with the specification itself. However, since in the sequel we report a few number of analyses relying on different abstractions of the automotive scenario, for the sake of presentation, we introduce each considered set of abstraction rules together with the associated set of SocL formulae.

```
let
 SensorsMonitor(car) = car.engineFailure!<diagnosticData>


 GpsSystem(car) = * car.reqLoc?<>. car.respLoc!<gpsPos>


 Discovery(car) =
   * [GPS] [TYPE]
     car.findServ?<GPS,TYPE>.
       [nonDet#][choice#]
       (nonDet.choice!<> | nonDet.choice?<>. car.found!<list>
                         + nonDet.choice?<>. car.notFound!<>)


 Reasoner(car) =
   [SERV_LIST]
   car.choose?<SERV_LIST>. car.chosen!<garage1,towTruck2,rentalCar1>


 CardCharge(car,end,undo,k) =
   bank.charge!<car,ccNum,amount,car>
   | { car.chargeFail?<car>.kill(k)
       + car.chargeOK?<car>.
         (end.end!<>
          | car.undo?<cc>. car.undo?<cc>.bank.revoke!<car>) }


 FindServices(car,end,undo,LOC,LIST,k) =
   car.reqLoc!<>
   | car.respLoc?<LOC>.
       (car.findServ!<LOC,servicesType>
        | car.found?<LIST>. end.end!<>
          + car.notFound?<>.
              (kill(k) | { car.undo!<cc> | car.undo!<cc> } ) )


 ChooseAndOrder(car,CAR_DATA,undo,LOC,LIST)  =
   [GPS]
   ( car.choose!<LIST>
     |
     [GARAGE] [TOWTRUCK] [RENTALCAR]
      car.chosen?<GARAGE,TOWTRUCK,RENTALCAR>.
        ( [GAR_INFO]
          -- Garage ordering
          ( GARAGE.orderGar!<car,CAR_DATA>
            |
            car.garageFail?<>.
              (car.undo!<cc>
```

```
              | [ass#][ign#] (ass.ign!<LOC> | ass.ign?<GPS>.nil)
            )
          + car.garageOK?<GPS,GAR_INFO>.
              ( [TOW_INFO]
                -- Tow Truck ordering
                ( TOWTRUCK.orderTow!<car,LOC,GPS>
                  |
                  car.towTruckFail?<>. car.undo!<gar>
                  + car.towTruckOK?<TOW_INFO>.nil
                )
                | car.undo?<gar>.
                   (GARAGE.cancel!<car> | car.undo!<cc> | car.undo!<rc>)
              )
        )
        |
        [RC_INFO]
        -- Rental Car ordering
        ( RENTALCAR.orderRC!<car,GPS>
          |
          car.rentalCarFail?<>. car.undo!<cc>
          + car.rentalCarOK?<RC_INFO>.
              car.undo?<rc>. RENTALCAR.redirect!<car,LOC>
        )
      )
  )


BankInterface(check,checkOK,checkFail) =
  * [CUST] [CC] [AMOUNT] [ID]
    bank.charge?<CUST,CC,AMOUNT,ID>.
      ( bank.check!<ID,CC,AMOUNT>
        | bank.checkFail?<ID>. CUST.chargeFail!<ID>
          + bank.checkOK?<ID>.
              [k] ( CUST.chargeOK!<ID> | bank.revoke?<ID>.kill(k) ) )


CreditRating(check,checkOK,checkFail) =
  * [ID] [CC] [A]
    bank.check?<ID,CC,A>.
      [p#][o#] (p.o!<> | p.o?<>. bank.checkOK!<ID>
                       + p.o?<>. bank.checkFail!<ID>)


Bank =
  [check#] [checkOK#] [checkFail#]
  ( BankInterface(check,checkOK,checkFail)
    | CreditRating(check,checkOK,checkFail) )
```

```
 Garage(garage) =
   * [CUST] [SENSORS_DATA] [checkOK#] [checkFail#]
     garage.orderGar?<CUST,SENSORS_DATA>.
       ( garage.checkOK!<> | garage.checkFail!<>
         | garage.checkFail?<>. CUST.garageFail!<>
           + garage.checkOK?<>.
               [k] ( CUST.garageOK!<garageGPS,garageInfo>
                     |
                     garage.cancel?<CUST>. kill(k) ) )


 TowTruck(towTruck) =
   * [CUST] [CAR_GPS] [GARAGE_GPS] [checkOK#] [checkFail#]
     towTruck.orderTow?<CUST,CAR_GPS,GARAGE_GPS>.
       ( towTruck.checkOK!<> | towTruck.checkFail!<>
         | towTruck.checkFail?<>. CUST.towTruckFail!<>
           + towTruck.checkOK?<>. CUST.towTruckOK!<towTruckInfo> )


 RentalCar(rentalCar) =
   * [CUST] [GPS] [checkOK#] [checkFail#]
     rentalCar.orderRC?<CUST,GPS>.
       ( rentalCar.checkOK!<> | rentalCar.checkFail!<>
         | rentalCar.checkFail?<>. CUST.rentalCarFail!<>
           + rentalCar.checkOK?<>.
               [k] ( CUST.rentalCarOK!<rentalCarInfo>
                     | [NEW_GPS]
                       rentalCar.redirect?<CUST,NEW_GPS>. kill(k)
                   )
       )

in
   [car#]
   ( SensorsMonitor(car) | GpsSystem(car) | Discovery(car) | Reasoner(car)
     |
     -- Orchestrator
     [CAR_DATA]
     ( car.lowOilFailure?<CAR_DATA>.nil
--     + ...other failures...
       + car.engineFailure?<CAR_DATA>.
           [end#] [undo#] [LOC] [LIST]
           ( [k] ( CardCharge(car,end,undo,k)
                   |
                   FindServices(car,end,undo,LOC,LIST,k)
                 )
             |
             end.end?<>.
               end.end?<>.
                  ChooseAndOrder(car,CAR_DATA,undo,LOC,LIST)
```

```
          )
     )
   )
   |
   Bank()
   |
   Garage(garage1) | Garage(garage2)
   |
   TowTruck(towTruck1) | TowTruck(towTruck2)
   |
   RentalCar(rentalCar1) | RentalCar(rentalCar2)
end
```

A.2.1 Verification of the abstract properties from Section 4.2.1

The abstraction rules used for this analysis are the following.

```
Abstractions {
 Action $car.engineFailure  -> request(road_assistance,$car)
 Action $car.towTruckOK     -> responseOk(road_assistance,$car)
 Action $car.rentalCarOK    -> responseOk(road_assistance,$car)
 Action $car.chargeFail     -> responseFail(road_assistance,$car)
 Action $car.notFound       -> responseFail(road_assistance,$car)
 Action $car.garageFail     -> responseFail(road_assistance,$car)
 Action $car.rentalCarFail  -> responseFail(road_assistance,$car)
 Action $car.towTruckFail   -> responseFail(road_assistance,$car)
 State  $car.engineFailure? -> accepting_request(road_assistance)
}
```

The SocL formulae written in the syntax of CMC are as follows.

```
1) -- Available service --
   AG(accepting_request(road_assistance))

   A[accepting_request(road_assistance)
      U {request(road_assistance,$var)} true]


2) -- Parallel service --
   AG [request(road_assistance,$var)]
      E[true {not (responseOk(road_assistance,%var) or
                   responseFail(road_assistance,%var))}
         U accepting_request(road_assistance)]
```

```
3) --Sequential service --
   AG [request(road_assistance,$var)]
      A[not accepting_request(road_assistance) {true}
         U {responseOk(road_assistance,%var) or
            responseFail(road_assistance,%var)} true]


4) -- One-shot service --
   AG [request(road_assistance,$var)]
      AG not accepting_request(road_assistance)

   AF{request(road_assistance,$var)}
     AG not accepting_request(road_assistance)


5) -- Off-line service ---
   AG [request(road_assistance,$var)]
      AF {responseFail(road_assistance,%var)} true



6) -- Cancelable service --
   AG [request(road_assistance,$var)]
      A[accepting_cancel(road_assistance,%var) {true}
         W {responseOk(road_assistance,%var) or
            responseFail(road_assistance,%var)} true]


7) -- Revocable service --
   EF {responseOk(road_assistance,$var)}
      EF (accepting_undo(road_assistance,%var))


8) -- Responsive service --
   AG [request(road_assistance,$var)]
      AF {responseOk(road_assistance,%var) or
          responseFail(road_assistance,%var)} true


9) -- Single-response service --
   AG [request(road_assistance,$var)]
      not EF {responseOk(road_assistance,%var) or
              responseFail(road_assistance,%var)}
             EF {responseOk(road_assistance,%var) or
                 responseFail(road_assistance,%var)} true


10) -- Multiple-response service --
    AG [request(road_assistance,$var)]
```

```
       AF {responseOk(road_assistance,%var) or
           responseFail(road_assistance,%var)}
       AF {responseOk(road_assistance,%var) or
           responseFail(road_assistance,%var)} true


11) -- No-response service --
    AG [request(road_assistance,$var)]
       not EF {responseOk(road_assistance,%var) or
               responseFail(road_assistance,%var)} true


12) -- Reliable service --
    AG [request(road_assistance,$var)]
       AF {responseOk(road_assistance,%var)} true
```

A.2.2 Verification of some request-response properties

The abstraction rules used for this analysis are the following.

```
Abstractions {
 Action $car.engineFailure  -> request(road_assistance,$car)
 Action $car.towTruckOK     -> responseOk(road_assistance,$car,truckGarage)
 Action $car.rentalCarOK    -> responseOk(road_assistance,$car,rentalCar)
 Action $car.chargeFail     -> responseFail(road_assistance,$car,truckGarage)
 Action $car.chargeFail     -> responseFail(road_assistance,$car,rentalCar)
 Action $car.notFound       -> responseFail(road_assistance,$car,truckGarage)
 Action $car.notFound       -> responseFail(road_assistance,$car,rentalCar)
 Action $car.garageFail     -> responseFail(road_assistance,$car,truckGarage)
 Action $car.rentalCarFail  -> responseFail(road_assistance,$car,rentalCar)
 Action $car.towTruckFail   -> responseFail(road_assistance,$car,truckGarage)
 State  $car.engineFailure? -> accepting_request(road_assistance)
}
```

The SocL formulae written in the syntax of CMC are as follows.

```
-- Once requested, the service always provides at least one response
-- about the status of the car renting.
--
AG [request(road_assistance,$var)]
   AF {responseOk(road_assistance,%var,rentalCar) or
       responseFail(road_assistance,%var,rentalCar)}  true


-- Once requested, the service always provides at least one response
```

```
-- about the status of the garage/tow truck ordering.
--
AG [request(road_assistance,$var)]
   AF {responseOk(road_assistance,%var,truckGarage) or
      responseFail(road_assistance,%var,truckGarage)} true


-- A positive response is never followed by a negative one for the same
-- order.
--
AG [responseOk(road_assistance,$var,$order)]
   not EF {responseFail(road_assistance,%var,%order)} true


-- A negative response is never followed by a positive one for the same
-- order.
--
AG [responseFail(road_assistance,$var,$order)]
   not EF {responseOk(road_assistance,%var,%order)} true
```

A.2.3 Analysis of other services of the automotive case study

The abstraction rules used for this analysis are the following.

```
Abstractions {
 Action $car.reqLoc                -> request(gps,$car)
 Action $car.respLoc               -> responseOk(gps,$car)
 State $car.reqLoc?                -> accepting_request(gps)

 Action bank.charge<*,*,*,$id>     -> request(charge,$id)
 Action *.chargeOK<$id>            -> responseOk(charge,$id)
 Action *.chargeFail<$id>          -> responseFail(charge,$id)
 Action bank.revoke<$id>           -> undo(charge,$id)
 State  bank.charge?               -> accepting_request(charge)
 State  bank.revoke?<$id>          -> accepting_undo(charge,$id)

 Action rentalCar1.orderRC<$car,*> -> request(rental_car1,$car)
 Action $car.rentalCarOK!          -> responseOk(rental_car1,$car)
 Action $car.rentalCarFail!        -> responseFail(rental_car1,$car)
 State  rentalCar1.orderRC?        -> accepting_request(rental_car1)
}
```

The SocL formulae written in the syntax of CMC are as follows.

```
-- The service GpsSystem is always available.
```

```
--
AG accepting_request(gps)


-- The service GpsSystem always replies with successful
-- responses, i.e. it is reliable.
--
AG [request(gps,$var)]
   AF {responseOk(gps,$var)} true


-- The service Bank is always available.
--
AG accepting_request(charge)


-- The service Bank, after it has accepted a request,
-- always provides a single (either positive or negative) response.
--
AG [request(charge,$id)]
   AF {responseOk(charge,%id) or responseFail(charge,%id)}
      not EF {responseOk(charge,%id) or responseFail(charge,%id)} true

-- After a successful response to a credit card charge request, the bank
-- accepts undo requests for the successfully completed transaction, i.e.
-- Bank is a strong revocable service.
--
AG [responseOk(charge,$id)]
   A[ accepting_undo(charge,%id) {true}
      W {undo(charge,%id)} true]


-- The service RentalCar_1 is always available.
AG accepting_request(rental_car1)


-- The service RentalCar_1, once a request is accepted, provides
-- a single (either positive or negative) response.
--
AG [request(rental_car1,$customer)]
   AF {responseOk(rental_car1,%customer) or
       responseFail(rental_car1,%customer)}
   not EF {responseOk(rental_car1,%customer) or
           responseFail(rental_car1,%customer)} true
```

A.2.4 Verification of orchestration and compensation properties

The abstraction rules used for this analysis are the following.

```
Abstractions {
 Action $car.engineFailure     -> request(road_assistance,$car)
 Action $car.towTruckOK        -> responseOk(road_assistance,$car,truckGarage)
 Action $car.rentalCarOK       -> responseOk(road_assistance,$car,rentalCar)
 Action $car.chargeFail        -> responseFail(road_assistance,$car,truckGarage)
 Action $car.chargeFail        -> responseFail(road_assistance,$car,rentalCar)
 Action $car.notFound          -> responseFail(road_assistance,$car,truckGarage)
 Action $car.notFound          -> responseFail(road_assistance,$car,rentalCar)
 Action $car.garageFail        -> responseFail(road_assistance,$car,truckGarage)
 Action $car.rentalCarFail     -> responseFail(road_assistance,$car,rentalCar)
 Action $car.towTruckFail      -> responseFail(road_assistance,$car,truckGarage)

 Action bank.charge<*,*,*,$id> -> request(charge,$id)
 Action *.chargeOK<$id>        -> responseOk(charge,$id)
 Action *.chargeFail<$id>      -> responseFail(charge,$id)
 Action bank.revoke<$id>       -> undo(charge,$id)

 Action $car.garageOk          -> responseOk(garage,$car)
 Action *.cancel<$car>         -> undo(garage,$car)
 Action $car.towTruckFail      -> responseFail(towtruck,$car)
}
```

The SocL formulae written in the syntax of CMC are as follows.

```
-- After a successful credit card charge, the rental car will be
-- booked, or the garage and tow truck will be ordered, or the credit
-- chard charge will be revoked.
--
AG [responseOk(charge,$id)]
   AF {responseOk(road_assistance,%id,rentalCar) or
       responseOk(road_assistance,%id,truckGarage) or
       undo(charge,%id)} true


-- It cannot happen that, after the driver's credit card has been
-- charged and some service ordered, the credit card charge is revoked.
--
not EF {responseOk(charge,$id)}
        EF {responseOk(road_assistance,%id,rentalCar) or
            responseOk(road_assistance,%id,truckGarage)}
        EF {undo(charge,%id)} true


-- It cannot happen that, after the credit card has been charged and
-- then revoked, some order succeeds.
--
not EF {responseOk(charge,$id)}
       EF {undo(charge,%id)}
          EF {responseOk(road_assistance,%id,rentalCar) or
```

```
            responseOk(road_assistance,%id,truckGarage)} true


-- After the garage has been booked, if the tow truck service is not
-- available then the garage is revoked.
--
AG [responseOk(garage,$var)]
   AG [responseFail(towtruck,%var)]
      AF {undo(garage,%var)} true
```

A.3 Finance case study

The complete specification of the finance case study written in the syntax of CMC is as follows.

```
let

  Portal(key,authentication,notAuthorized,authorized,createInst) =
    * [USER] [PWD] [CUST] portal.login?<USER,PWD,CUST>.
      (portal.authentication!<USER,PWD>
         | portal.notAuthorized?<USER>. CUST.failedLogin!<key>
         +
         portal.authorized?<USER>.
         [sessionID#] (CUST.logged!<key,sessionID>
           | portal.creditRequest?<sessionID>.
             portal.createInst!<sessionID>
             + portal.bankTransferRequest?<sessionID>. nil
--           + ...other services provided by the credit portal...
       )
      )


 Authentication(authentication,notAuthorized,authorized) =
    * [USER] [PWD] portal.authentication?<USER,PWD>.
        [nonDet#] [choice#](nonDet.choice!<>
          | nonDet.choice?<>. portal.notAuthorized!<USER>
          + nonDet.choice?<>. portal.authorized!<USER>
        )

 Customer(key,username,password,amount,amountRevised)  =
    [k] (portal.login!<username,password,customer>
         | [ID] (customer.failedLogin?<key>.nil
             + customer.logged?<key,ID>.
               ( portal.creditRequest!<ID>
                 |
                -- at any time the customer could require
                -- the cancellation of the credit request processing
                  [exit#] (customer.exit!<> | customer.exit?<>.
```

```
    ({portal.cancel!<ID>} | kill(k)))
   |
   portal.getCreditRequest!<ID,customerData,amount,customer>
   | portal.securities!<ID,securityValues>
   | portal.balance!<ID,balance>
   | customer.balanceNotValid?<ID>.
   -- balance not valid (1st time)
   (portal.balance!<ID,balanceRevised>
     | customer.balanceNotValid?<ID>.
     -- balance not valid (2nd time): terminates
     kill(k)
   )
   |
  [OFFER] [MOTIVATIONS]
    ( -- receives a negative response
      customer.negativeResp?<ID,MOTIVATIONS>. kill(k)
      +
      -- receives an offer
      customer.offer?<ID,OFFER,MOTIVATIONS>.
        [nonDet#] [choice#](nonDet.choice!<>
          | nonDet.choice?<>. (kill(k) | {portal.answer!<ID,yes>})
          + nonDet.choice?<>. (kill(k) | {portal.answer!<ID,no>})
        )
      +
      -- updating required
      customer.update?<ID,MOTIVATIONS>.
        (portal.updAnswer!<ID,yes>
        | portal.updAmount!<ID,yes>
            | portal.newAmount!<ID,amountRevised>
        | portal.updSecurities!<ID,no>
        | [OFFER] [MOTIVATIONS]
          (customer.negativeResp?<ID,MOTIVATIONS>. kill(k)
           +
           customer.offer?<ID,OFFER,MOTIVATIONS>.
             [nonDet#] [choice#](nonDet.choice!<>
               | nonDet.choice?<>. (kill(k)
                                    | {portal.answer!<ID,yes>})
               + nonDet.choice?<>. (kill(k)
                                    | {portal.answer!<ID,no>})
             )
             +
             customer.update?<ID,MOTIVATIONS>. kill(k)
          )
        )
      )
    )
  )
)
```

```
InformationUpload(createInst,reqProcessing)  =
   * [ID] portal.createInst?<ID>.
        [k] [fault#] [abort#]
        (
         [abortFault]
         (
          [CUST_DATA] [SEC_DATA] [FINAL_BALANCE] [AMOUNT] [CUST]
          portal.getCreditRequest?<ID,CUST_DATA,AMOUNT,CUST>.
            [par#] [end#]
            ( -- Activities 1)
              portal.securities?<ID,SEC_DATA>. par.end!<>
              |
              -- Activities 2)
              [repeat#] [loop#]
              ( repeat.loop!<>
                | * repeat.loop?<>.
                    [BALANCE] portal.balance?<ID,BALANCE>.
                       -- invoke validation service
                       (validation.validateBalance!<ID,portal,BALANCE>
                         | portal.validateBalance?<ID,no>.
                         -- notify the customer that balances
                         -- are not valid and cycles
                            ( CUST.balanceNotValid!<ID> | repeat.loop!<> )
                           +
                           portal.validateBalance?<ID,yes>. par.end!<BALANCE>
                       )
              )
              |
              -- Activities 1) and 2) terminates successfully
              par.end?<>. par.end?<FINAL_BALANCE>.
                -- invokes RequestProcessing
                (kill(k) | {portal.reqProcessing!<ID,CUST_DATA,
                                  SEC_DATA,FINAL_BALANCE,AMOUNT,CUST>})
            )
          | portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
         )
         |
         -- fault handler
         fault.abort?<>. nil
       )


InformationUpdate(reqProcessing,reqUpdate) =
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT] [CUST] [MOTIVATIONS]
     portal.reqUpdate?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST,MOTIVATIONS>.
        [k] [fault#] [abort#]
        (
         [abortFault] [NEW_AMOUNT] [NEW_SEC_DATA] [assign#] [ment#] [par#] [end#]
```

```
        ( -- notifies the customer of needing to update the data
         customer.update!<ID,MOTIVATIONS>
         | (portal.updAnswer?<ID,no>. kill(k)
            + portal.updAnswer?<ID,yes>.
              ( ( -- updates the amount
                  portal.updAmount?<ID,no>. (assign.ment!<AMOUNT>
                    | assign.ment?<NEW_AMOUNT>. par.end!<>)
                  + portal.updAmount?<ID,yes>.
                      portal.newAmount?<ID,NEW_AMOUNT>. par.end!<>
                )
                |
                ( -- updates the securities
                  portal.updSecurities?<ID,no>. (assign.ment!<SEC_DATA>
                    | assign.ment?<NEW_SEC_DATA>. par.end!<>)
                  + portal.updSecurities?<ID,yes>.
                      portal.newSecurities?<ID,NEW_SEC_DATA>. par.end!<>
                )
                |
                -- Updating terminated
                par.end?<>. par.end?<>.
                  -- invokes RequestProcessing
                  (kill(k) | {portal.reqProcessing!<ID,CUST_DATA,
                                NEW_SEC_DATA,BALANCE,NEW_AMOUNT,CUST>})
              )
          )
         | portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
        )
        |
        -- fault handler
        fault.abort?<>. nil
       )


RequestProcessing(reqProcessing,reqUpdate,contractProcessing) =
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT] [CUST]
     portal.reqProcessing?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST>.
        [k] [fault#] [abort#] [undo#]
        (
         [abortFault]
         (
          -- adds request to employee task list
          portal.addToETL!<ID,SEC_DATA,BALANCE,AMOUNT>
          | portal.taskAddedToETL?<ID>.
            (
              -- installs the compensation handler
              {portal.undo?<empTaskList>. portal.removeTaskETL!<ID>}
              |
              -- receives evaluation from an employee
              [RATING] [ADDITIONAL_INFO] [DECISION]
```

```
portal.empEvaluation?<ID,RATING,ADDITIONAL_INFO,DECISION>.
 [cond#] [choice#] (
   cond.choice!<DECISION>
   |
   -- 1) negative evaluation
   cond.choice?<no>.
     (kill(k) | {CUST.negativeResp!<ID,ADDITIONAL_INFO>})
   +
   -- 2) ask to update
   cond.choice?<update>. (kill(k)
   | {portal.reqUpdate!<ID,CUST_DATA, SEC_DATA,
                        BALANCE,AMOUNT,CUST,ADDITIONAL_INFO>})
   +
   -- 3) positive evaluation
   cond.choice?<yes>.
     ( -- adds request to supervisor task list
      portal.addToSTL!<ID,SEC_DATA,BALANCE,AMOUNT,ADDITIONAL_INFO>
      | portal.taskAddedToSTL?<ID>.
        ( -- installs the compensation handler
         {portal.undo?<supTaskList>. portal.removeTaskSTL!<ID>}
         |
         -- receives evaluation from a supervisor
         [OFFER] [MOTIVATIONS] [SUP_DECISION]
           portal.supEvaluation?<ID,OFFER,
                                 MOTIVATIONS,
                                 SUP_DECISION>.
             [cond#] [choice#] (
               cond.choice!<SUP_DECISION>
               |
               -- 1) negative evaluation
               cond.choice?<no>.
                 (kill(k) | {CUST.negativeResp!<ID,MOTIVATIONS>})
               +
               -- 2) ask to update
               cond.choice?<update>.
                 (kill(k) | {portal.reqUpdate!<ID,CUST_DATA,
                              SEC_DATA,BALANCE,AMOUNT,
                              CUST,MOTIVATIONS>})
               +
               -- 3) positive evaluation
               cond.choice?<yes>.
                  ( -- sends the unrated offer to the customer
                   CUST.offer!<ID,OFFER,MOTIVATIONS>
                   | -- receives customer's answer
                     (portal.answer?<ID,yes>.
                         (kill(k)
                          | {portal.contractProcessing!<ID,
                              CUST_DATA, SEC_DATA, BALANCE,
                              AMOUNT, CUST, RATING,
```

```
                                                ADDITIONAL_INFO, OFFER,
                                                MOTIVATIONS>}
                                          )
                                       + portal.answer?<ID,no>. kill(k)
                                     )
                                  )
                               )
                            )
                         )
                      )
                   )
                  |
                  -- receive cancellation from customer
                  portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
                 )
                 |
                 -- fault handler
                 fault.abort?<>.
                   -- compensateAll
                   (portal.undo!<empTaskList> | portal.undo!<supTaskList> )
                )


ContractProcessing(contractProcessing)=
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT]
     [CUST] [RATING] [ADDITIONAL_INFO] [OFFER] [MOTIVATIONS]
     portal.contractProcessing?<ID,CUST_DATA,SEC_DATA,BALANCE,
       AMOUNT,CUST,RATING,ADDITIONAL_INFO,OFFER,MOTIVATIONS>.
       -- ... contract processing ...
       nil


ValidationService  =
   * [ID] [BANK][BALANCE]
     validation.validateBalance?<ID,BANK,BALANCE>.
     [nonDet#] [choice#](nonDet.choice!<>
                         | nonDet.choice?<>. BANK.validateBalance!<ID,yes>
                           + nonDet.choice?<>. BANK.validateBalance!<ID,no> )


EmployeeTaskList =
   * [ID] [SEC_DATA] [BALANCE] [AMOUNT]
     portal.addToETL?<ID,SEC_DATA,BALANCE,AMOUNT>.
       ( portal.taskAddedToETL!<ID>
         | [EMP] (portal.askTaskETL?<EMP>.
                    EMP.getTaskETL!<ID,SEC_DATA,BALANCE,AMOUNT>
                  + portal.removeTaskETL?<ID>. nil
                 )
       )
```

```
Employee(employee) =
   [repeat#] [loop#]
     ( repeat.loop!<>
       | * repeat.loop?<>.
           ( portal.askTaskETL!<employee>
             | [ID] [SEC_DATA] [BALANCE] [AMOUNT]
               employee.getTaskETL?<ID,SEC_DATA,BALANCE,AMOUNT>.
                  -- ... evaluates the request ...
                  [nonDet#] [choice#](nonDet.choice!<>
                    | -- sends the evaluation
                    nonDet.choice?<>.
                      (portal.empEvaluation!<ID,rating,additionalInfo,yes>
                        | repeat.loop!<>)
                    + nonDet.choice?<>.
                      (portal.empEvaluation!<ID,rating,additionalInfo,no>
                        | repeat.loop!<>)
                    + nonDet.choice?<>.
                      (portal.empEvaluation!<ID,rating,additionalInfo,update>
                        | repeat.loop!<>)
                  )
           )
     )


SupervisorTaskList =
   * [ID] [SEC_DATA] [BALANCE] [AMOUNT] [ADDITIONAL_INFO]
     portal.addToSTL?<ID,SEC_DATA,BALANCE,AMOUNT,ADDITIONAL_INFO>.
       (portal.taskAddedToSTL!<ID>
        | [SUP] (portal.askTaskSTL?<SUP>. SUP.getTaskSTL!<ID,SEC_DATA,
                                            BALANCE,AMOUNT,
                                            ADDITIONAL_INFO>
                 +
                 portal.removeTaskSTL?<ID>. nil
                )
       )


Supervisor(supervisor) =
   [repeat#] [loop#]
     ( repeat.loop!<>
       | * repeat.loop?<>.
           ( portal.askTaskSTL!<supervisor>
             | [ID] [SEC_DATA] [BALANCE] [AMOUNT] [INFO]
               supervisor.getTaskSTL?<ID,SEC_DATA,BALANCE,AMOUNT,INFO>.
                  -- ... evaluates the request ...
                  [nonDet#] [choice#](nonDet.choice!<>
                     | -- sends the evaluation
```

```
                    nonDet.choice?<>.
                      (portal.supEvaluation!<ID,offer,motivations,yes>
                        | repeat.loop!<>)
                    + nonDet.choice?<>.
                      (portal.supEvaluation!<ID,offer,motivations,no>
                        | repeat.loop!<>)
                    + nonDet.choice?<>.
                      (portal.supEvaluation!<ID,offer,motivations,update>
                        | repeat.loop!<>)
                )
            )
      )

in
 [key#]
 ( Customer(key,francesco,sensoria,15000,10000)
   | [createInst#] [reqProcessing#] [reqUpdate#] [contractProcessing#]
     ( [authentication#] [notAuthorized#] [authorized#] (
         Portal(key,authentication,notAuthorized,authorized,createInst)
         | Authentication(authentication,notAuthorized,authorized) )
       | InformationUpload(createInst,reqProcessing)
       | InformationUpdate(reqProcessing,reqUpdate)
       | RequestProcessing(reqProcessing,reqUpdate,contractProcessing)
       | ContractProcessing(contractProcessing)
       | EmployeeTaskList()
       | SupervisorTaskList()
     )
 )
 | ValidationService()
 | Employee(employee)
 | Supervisor(supervisor)
end
```

Abstraction rules

The abstraction rules used for our analysis are the following.

```
Abstractions {
    Action creditRequest<$1> -> request(cr,$1)
    Action balanceNotValid<$1> -> fail(cr,$1)
    Action negativeResp<$1,*> -> fail(cr,$1)
    Action offer<$1,*,*> -> response(cr,$1)
    Action update<$1,*> -> fail(cr,$1)
    Action cancel<$1> -> cancel(cr,$1)
    Action supEvaluation<$1,*,*,yes> -> response(seval,$1)
    Action supEvaluation<$1,*,*,no> -> fail(seval,$1)
    Action empEvaluation<$1,*,*,yes> -> response(eeval,$1)
    Action empEvaluation<$1,*,*,no> -> fail(eeval,$1)
    Action validateBalance<$1,yes> -> response(beval,$1)
```

```
    Action validateBalance<$1,no> -> fail(beval,$1)
    Action taskAddedToETL<$1> -> request(eval,$1)
    Action taskAddedToSTL<$1> -> request(eval,$1)
    Action removeTaskSTL<$1> -> cancel(eval,$1)
    Action removeTaskETL<$1> -> cancel(eval,$1)
    Action taskAddedToSTL<$1> -> request(tostl,$1)
    Action reqUpdate<$1,*,*,*,*,*,*> -> request(upd,$1)
    Action update<$1,*> -> response(upd,$1)
    Action securities<$1,*> -> request(sec,$1)
    Action balance!<$1,*> -> request(bal,$1)
    Action reqProcessing<$1,*,*,*,*> -> request(rproc,$1)
    State login -> accepting_request(login)
}
```

SocL properties

We report the SocL formulae expressing the properties that the case study is expected to fulfill, written in the syntax of CMC.

```
(Availability) AG accepting_request(login)

(Responsiveness and correlation soundness)
  AG [request(cr,$id)]
     AF {response(cr,%id) or (fail(cr,%id) or cancel(cr,%id))} true

(Interruptibility) AG [request(cr,$id)] EF {cancel(cr,%id)} true

(i) AG [request(cr,$id)]
     not E[true {not response(seval,%id)} U {response(cr,%id)} true]

(ii) AG [request(cr,$id)]
    not E[true {not (fail(seval,%id) or fail(eeval,%id)
                     or fail(beval,%id))} U {fail(cr,%id)} true]

(iii) AG [request(eval,$id)] EF [cancel(cr,%id)]
         AF {cancel(eval,%id)} true

(iv) AG [request(upd,$id)]
        AF {cancel(cr,%id) or response(upd,%id)} true

(v) AG [request(cr,$id)]
    not E[true {not request(sec,%id)
                or request(bal,%id)} U {request(rproc,%id)}true]

(vi) AG [ request(cr,$id) ]
        AF {not cancel(cr,%id) or response(cr,%id)} true

(vii) AG [request(cr,$id)]
         AF {not cancel(cr,%id) or request(tostl,%id)}true
```

Appendix B

Proofs of results in Chapters 4 and 5

We report in this chapter the proofs of major results stated in Chapters 4 and 5.

B.1 Proofs of results in Section 4.1

Lemma B.1.1 (Substitution Lemma, Lemma4.1.1) *If* $\Gamma, \{x : r\} \vdash s \succ \Gamma', \{x : r'\} \vdash s'$ *and* $\sigma = \{x \mapsto \{v\}_{r''}\}$, *then* $\Gamma \cdot \sigma \vdash s \cdot \sigma \succ \Gamma' \cdot \sigma \vdash s' \cdot \sigma$.

Proof. The proof proceeds by induction on the length of the inference used to derive the typing judgement.

The base cases *(t-nil)* and *(t-kill)* are trivial to conclude. Let us consider the base case *(t-inv)*: by definition $s = s' = u_1 \bullet u_2!\langle\{\epsilon_1(\bar{y}_1)\}_{r_1}, \ldots, \{\epsilon_n(\bar{y}_n)\}_{r_n}\rangle$ where $u_1 \in r_i$ for each $i \in \{1, \ldots, n\}$. Hence, we also have that $u_1 \cdot \sigma \in r_i \cdot \sigma$ for any r_i and σ. We now distinguish three cases:

1. $\bar{y}_1 \cup \ldots \cup \bar{y}_n = \emptyset$. In this case, $\Gamma' = \Gamma$ and $r' = r$; thus by using rule *(t-inv)*, we can conclude that $\Gamma \cdot \sigma \vdash u_1 \bullet u_2!\langle\{\epsilon_1\}_{r_1}, \ldots, \{\epsilon_n\}_{r_n}\rangle \cdot \sigma \succ \Gamma' \cdot \sigma \vdash u_1 \bullet u_2!\langle\{\epsilon_1\}_{r_1}, \ldots, \{\epsilon_n\}_{r_n}\rangle \cdot \sigma$.

2. Let $I \stackrel{def}{=} \{j \in \{1, \ldots, n\} : x \in \bar{y}_j\}$. In this case, it must be $r' = \bigcup_{i \in I} r_i \cup r$ and $\Gamma' = \Gamma + \{z : r_1\}_{z \neq x, z \in \bar{y}_1} + \ldots + \{z : r_n\}_{z \neq x, z \in \bar{y}_n}$. Now, we can to conclude like in the previous case.

3. $x \notin \bar{y}_1 \cup \ldots \cup \bar{y}_n$. In this case $r' = r$ and, again, we can conclude like before.

Let us now consider the inductive case and reason by case analysis on the last rule used to infer the judgement. We explicitly show the most significant cases, the remaining ones are easier.

(t-rec): Let $s = p \bullet o?\bar{w}.t$ and $s' = p \bullet o?\bar{w}.t'$. Then, we can suppose that $\Gamma'', \{x : r''\} \vdash t \succ \Gamma', \{x : r'\} \vdash t'$ for some Γ'' and r''. We distinguish three cases:

1. $\mathrm{fv}(\bar{w}) = \emptyset$. In this case $\Gamma'' = \Gamma$ and $r'' = r$. By induction, we have $\Gamma \cdot \sigma \vdash t \cdot \sigma \succ \Gamma' \cdot \sigma \vdash t' \cdot \sigma$. Now, by using rule *(t-rec)*, we can conclude that $\Gamma \cdot \sigma \vdash p \bullet o?\bar{w}.(t \cdot \sigma) \succ \Gamma' \cdot \sigma \vdash p \bullet o?\bar{w}.(t' \cdot \sigma)$.
2. $x \in \mathrm{fv}(\bar{w})$. In this case $\Gamma'' = \Gamma + \{z : \{p\}\}_{z \neq x, z \in \mathrm{fv}(\bar{w})}$ and $r'' = r \cup \{p\}$ and by induction, $\Gamma \cdot \sigma + \{z : \{p\}\}_{z \neq x, z \in \mathrm{fv}(\bar{w})} \vdash t \cdot \sigma \succ \Gamma' \cdot \sigma \vdash t' \cdot \sigma$. Hence by using rule *(t-rec)*, we can conclude the wanted $\Gamma \cdot \sigma \vdash (p \bullet o?\bar{w}.t) \cdot \sigma \succ \Gamma' \cdot \sigma \vdash (p \bullet o?\bar{w}.t') \cdot \sigma$.
3. $\mathrm{fv}(\bar{w}) \neq \emptyset$ and $x \notin \mathrm{fv}(\bar{w})$. In this case $\Gamma'' = \Gamma + \{z : \{p\}\}_{z \in \mathrm{fv}(\bar{w})}$ and $r'' = r$, and we can conclude like before.

(t-del$_{var}$): Let $s = [z]\,t$ and $s' = [\{z\}^{r''-\{z\}}]\,t'$ for some r''. Since $z \in \mathrm{bv}(s)$, by type environment definition, we have $z \neq x$. Then, we can suppose that $\Gamma, \{x : r\}, \{z : \emptyset\} \vdash t \succ \Gamma', \{x : r'''\}, \{z : r''\} \vdash t'$ for some r''' such that $z \notin (r''' \cup \mathrm{reg}(\Gamma'))$. Thus, by induction $\Gamma \cdot \sigma, \{z : \emptyset\} \vdash t \cdot \sigma \succ \Gamma' \cdot \sigma, \{z : r'' \cdot \sigma\} \vdash t' \cdot \sigma$. Hence, by using rule *(t-del$_{var}$)*, we can conclude the wanted $\Gamma \cdot \sigma \vdash [z]\,t \cdot \sigma \succ \Gamma' \cdot \sigma \vdash [\{z\}^{r'' \cdot \sigma - \{z\}}]\,t' \cdot \sigma$.

(t-par): Let $s = s_1 \mid s_2$ and $s' = s'_1 \mid s'_2$. Then, we can suppose that $\Gamma, \{x : r\} \vdash s_1 \succ \Gamma_1, \{x : r'_1\} \vdash s'_1$ and $\Gamma, \{x : r\} \vdash s_2 \succ \Gamma_2, \{x : r'_2\} \vdash s'_2$. By induction, $\Gamma \cdot \sigma \vdash s_1 \cdot \sigma \succ \Gamma_1 \cdot \sigma \vdash s'_1 \cdot \sigma$ and $\Gamma \cdot \sigma \vdash s_2 \cdot \sigma \succ \Gamma_2 \cdot \sigma \vdash s'_2 \cdot \sigma$. Hence by letting $\Gamma' = \Gamma_1 + \Gamma_2$ we can conclude the wanted $\Gamma \cdot \sigma \vdash (s_1 \mid s_2) \cdot \sigma \succ \Gamma' \cdot \sigma \vdash (s'_1 \mid s'_2) \cdot \sigma$.

□

Theorem B.1.1 (Theorem 4.1.1) *If $\Gamma'_1 \vdash s'_1 \succ \Gamma_1 \vdash s_1$ and $s_1 \xrightarrow{\alpha} s_2$ then there exist a raw service s'_2 and two type environments Γ_2 and Γ'_2 such that $\Gamma_2 \sqsubseteq \Gamma_1$, $\Gamma'_1 \sqsubseteq \Gamma'_2$ and $\Gamma'_2 \vdash s'_2 \succ \Gamma_2 \vdash s_2$.*

Proof. The proof proceeds by induction on the length of the inference of $s_1 \xrightarrow{\alpha} s_2$.
Base Step: We reason by case analysis on the axioms of the typed operational semantics.

(kill) By rule *(t-kill)*, $s_1 = s'_1 = \mathbf{kill}(k)$ and $\Gamma_1 = \Gamma'_1$. Thus, since $s_2 = \mathbf{0}$ and by rule *(t-nil)*, it is trivial to conclude, by letting $\Gamma_2 = \Gamma'_2 = \Gamma'_1$.

(r-inv) By rule *(t-inv)*, $s_1 = s_1 = p \bullet o!\overline{\{\epsilon\}_r}$. Thus, since $s_2 = \mathbf{0}$ and by rule *(t-nil)*, it is trivial to conclude, by letting $\Gamma_2 = \Gamma'_2 = \Gamma'_1$.

(rec) By hypothesis, $s_1 = p \bullet o?\bar{w}.s_2$ and $\Gamma'_1 \vdash p \bullet o?\bar{w}.s'_2 \succ \Gamma_1 \vdash p \bullet o?\bar{w}.s_2$. By the premise of rule *(t-rec)*, we can conclude that $\Gamma'_2 \vdash s'_2 \succ \Gamma_2 \vdash s_2$ with $\Gamma'_2 = \Gamma'_1 + \{x : \{p\}\}_{x \in \mathrm{fv}(\bar{w})}$ and $\Gamma_2 = \Gamma_1$.

Inductive Step: We reason by case analysis on the last applied inference rule of the typed operational semantics.

(r-com) By hypothesis, $s_1 = s_l \mid s_r$, $s_2' = s_l' \mid s_r'$ and there exist two raw services s_l'' and s_r'', and two type environments Γ_l and Γ_r, such that $\Gamma_1 = \Gamma_l + \Gamma_r$ and $\Gamma_1' \vdash s_l'' \mid s_r'' \succ \Gamma_l + \Gamma_r \vdash s_l \mid s_r$. By the premises of rule *(t-par)*, $\Gamma_1' \vdash s_l'' \succ \Gamma_l \vdash s_l$ and $\Gamma_1' \vdash s_r'' \succ \Gamma_r \vdash s_r$. By the premises of rule *(r-com)*, $s_l \xrightarrow{\mathtt{n} \rhd \bar{w}} s_l'$ and $s_r \xrightarrow{\mathtt{n} \lhd \overline{\{v\}_r}} s_r'$, then, by induction, there exist two raw services s_l''' and s_r''', and four type environment Γ_l', Γ_l'', Γ_r' and Γ_r'', such that $\Gamma_1' \sqsubseteq \Gamma_l''$, $\Gamma_1' \sqsubseteq \Gamma_r''$, $\Gamma_l'' \vdash s_l''' \succ \Gamma_l' \vdash s_l'$ and $\Gamma_r'' \vdash s_r''' \succ \Gamma_r' \vdash s_r'$. By the definition of the preorder $\sqsubseteq$, $\Gamma_1' \sqsubseteq \Gamma_l''$ implies that there exists Γ such that $\Gamma_1' + \Gamma = \Gamma_l''$, and, similarly, $\Gamma_1' \sqsubseteq \Gamma_r''$ implies that there exists Γ' such that $\Gamma_1' + \Gamma' = \Gamma_r''$. By repeated applications of Lemma 4.1.2, we obtain $\Gamma_l'' + \Gamma' \vdash s_l''' \succ \Gamma_l' + \Gamma' \vdash s_l'$ and $\Gamma_r'' + \Gamma \vdash s_r''' \succ \Gamma_r' + \Gamma \vdash s_r'$. Since $\Gamma_l'' + \Gamma' = \Gamma_1' + \Gamma + \Gamma' = \Gamma_r'' + \Gamma$, by applying rule *(t-par)*, we can conclude.

(par$_3$) - (par$_{com}$) - (par$_{kill}$) These cases are similar to the previous one; the latter case relies on Lemma 4.1.3.

(choice) By hypothesis, $s_1 = g_1 + g_2$ and there exist g_1' and g_2', and two type environments Γ and Γ', such that $\Gamma_1 = \Gamma + \Gamma'$ and $\Gamma_1' \vdash g_1' + g_2' \succ \Gamma + \Gamma' \vdash g_1 + g_2$. By the premise of rule *(t-sum)*, $\Gamma_1' \vdash g_1' \succ \Gamma \vdash g_1$. By the premises of rule *(choice)*, $g_1 \xrightarrow{\alpha} s_2$ and, by induction, we can conclude.

(prot) By hypothesis, $s_1 = \{\!|s|\!\}$ and there exists a raw service s' such that $\Gamma_1' \vdash \{\!|s'|\!\} \succ \Gamma_1 \vdash \{\!|s|\!\}$. By the premise of rule *(t-prot)*, $\Gamma_1' \vdash s' \succ \Gamma_1 \vdash s$. By the premise of rule *(prot)*, $s \xrightarrow{\alpha} s''$ such that $s_2 = \{\!|s''|\!\}$. By induction, there exist s''', Γ and Γ' such that $\Gamma' \vdash s''' \succ \Gamma \vdash s''$. Then, by rule *(t-prot)*, we can conclude.

(r-del$_{com}$) By hypothesis, $s_1 = [\{x\}^r]\, s$ and there exists a raw services s'' such that $\Gamma_1' \vdash [x]\, s'' \succ \Gamma_1 \vdash [\{x\}^r]\, s$. By the premise of rule *(t-del$_{var}$)*, we obtain $\Gamma_1', \{x : \emptyset\} \vdash s'' \succ \Gamma_1, \{x : r\} \vdash s$ (for simplicity, we assume $x \notin r$). By the premise of rule *(r-del$_{com}$)*, $s \xrightarrow{\mathtt{n}\, \sigma \uplus \{x \mapsto \{v\}_{r'}\}\, \ell\, \vec{v}'} s'$ with $s_2 = s' \cdot \{x \mapsto \{v\}_{r'}\}$. Hence, by Lemma B.1.1, $\Gamma_1' \cdot \{x \mapsto \{v\}_{r'}\} \vdash s'' \cdot \{x \mapsto \{v\}_{r'}\} \succ \Gamma_1 \cdot \{x \mapsto \{v\}_{r'}\} \vdash s \cdot \{x \mapsto \{v\}_{r'}\}$. Then, by definition of function $\mathcal{M}(\_, \_)$ and rule *(r-com)*, $s \cdot \{x \mapsto \{v\}_{r'}\} \xrightarrow{\mathtt{n}\, \sigma\, \ell\, \vec{v}'} s' \cdot \{x \mapsto \{v\}_{r'}\}$ and, by induction, we can conclude.

(del$_{kill\,1}$) By hypothesis, $s_1 = [k]\, s$ and there exists a raw services s'' such that $\Gamma_1' \vdash [k]\, s'' \succ \Gamma_1 \vdash [k]\, s$. By the premise of rule *(t-del$_{lab}$)*, $\Gamma_1' \vdash s'' \succ \Gamma_1 \vdash s$. By the premises of rule *(del$_{kill}$)*, $s \xrightarrow{k} s'$ with $s_2 = [k]\, s'$, then, by induction, there exist s''', Γ and Γ' such that $\Gamma' \vdash s''' \succ \Gamma \vdash s'$. Thus, by applying rule *(t-del$_{lab}$)*, we can conclude.

(r-del$_{kill\,2}$) - (r-del$_{kill\,3}$) - (r-del) These cases are similar to the previous one.

(r-del$_{var}$) By hypothesis, $s_1 = [\{x\}^r]\, s$ and there exists a raw services s'' such that $\Gamma_1' \vdash [x]\, s'' \succ \Gamma_1 \vdash [\{x\}^r]\, s$. By the premise of rule *(t-del$_{var}$)*, we obtain $\Gamma_1', \{x : \emptyset\} \vdash s'' \succ \Gamma_1, \{x : r\} \vdash s$ (for simplicity, we assume $x \notin r$). By the premise of rule *(r-del)*, $s \xrightarrow{\alpha} s'$ with $s_2 = [\{x\}^r]\, s'$. By induction, there exist s''', Γ and Γ' such that $\Gamma' \vdash s''' \succ \Gamma \vdash s'$ and $\Gamma \sqsubseteq \Gamma_1, \{x : r\}$. Thus, we can obtain $\Gamma'', \{x : \emptyset\} \vdash s''' \succ \Gamma''', \{x : r\} \vdash s'$ for some Γ'' and Γ''', and, by applying rule *(t-del$_{var}$)*, we can conclude.

(str) By straightforward induction and by relying on Lemma 4.1.4.

□

Theorem B.1.2 (Type Safety, Theorem 4.1.2) *If s is a well-typed service then $s \Uparrow$ does not hold.*

Proof. The proof is straightforward. We prove, by rule induction, that if $s \Uparrow$ then s is not well-typed. Looking at the rules in Table 4.4, we see that the only case to be considered is when there exists $r' \in \bar{r}$ such that $p \notin r'$: in such a case, there is no way to infer a type for $p \bullet o!\overline{\{\epsilon\}}_r$ using *(t-inv)*. The other cases follow by induction. Suppose, for example, that $[n]\, s \Uparrow$ because $s \Uparrow$. By induction, s is not well-typed and therefore we cannot use *(t-del$_{name}$)*, that is the only possible rule to infer that $[n]\, s$ is well-typed. □

B.2 Proofs of results in Section 4.3

B.2.1 μCOWS^m

Theorem B.2.1 (Theorem 4.3.1) *$\sim_m$ is a congruence for μCOWS^m closed terms.*

Proof. We shall prove that, given two μCOWS^m closed terms s_1 and s_2, if $s_1 \sim_m s_2$ then $\mathbb{C}[\![s_1]\!] \sim_m \mathbb{C}[\![s_2]\!]$ for every (possibly open) context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$. The base case, i.e. whenever $\mathbb{C} = [\![\ ]\!]$, is trivial. For the inductive case, we have the following possibilities:

- $\mathbb{C} = \mathtt{n}?\bar{w}.\, \mathbb{D}$.
  By induction, we may assume that $\mathbb{D}[\![s_1]\!] \sim_m \mathbb{D}[\![s_2]\!]$. If $\mathbb{C}$ is a closed context, then $\bar{w} = \bar{v}$, i.e. $\bar{w}$ only contains values, $\mathbb{D}$ is a closed context, and there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. By Lemma 4.3.1, we can prove the thesis by showing that $(\mathcal{F} \backslash \mathcal{R}_\emptyset) \cup \mathcal{R}_\emptyset'$ is a bisimulation up-to $\equiv$, where

  $$\mathcal{R}_\emptyset' = \{(\mathtt{n}?\bar{v}.\, \mathbb{D}[\![s_1]\!], \mathtt{n}?\bar{v}.\, \mathbb{D}[\![s_2]\!])\} \cup \mathcal{R}_\emptyset$$

  Indeed, $\mathcal{R}_\emptyset'$ is *transition closed* because $\mathcal{R}_\emptyset$ is transition closed and

  $$\mathtt{n}?\bar{v}.\, \mathbb{D}[\![s_1]\!] \xrightarrow{\mathtt{n} \rhd \bar{v}} \equiv \mathbb{D}[\![s_1]\!]$$
  $$\mathtt{n}?\bar{v}.\, \mathbb{D}[\![s_2]\!] \xrightarrow{\mathtt{n} \rhd \bar{v}} \equiv \mathbb{D}[\![s_2]\!]$$

From the hypothesis $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$, since $\mathcal{R}_\emptyset \subseteq \mathcal{R}'_\emptyset$, we have $\mathbb{D}[\![s_1]\!]\mathcal{R}'_\emptyset\mathbb{D}[\![s_2]\!]$.

Instead, if $\bar{w}$ and $\mathbb{D}$ contain free variables $\bar{x}$ at most, then the hypothesis $\mathbb{D}[\![s_1]\!] \sim_m \mathbb{D}[\![s_2]\!]$ implies that for all $\bar{v}$ such that $\mid \bar{x} \mid = \mid \bar{v} \mid$ we have $\mathbb{D}'[\![s_1]\!] \sim_m \mathbb{D}'[\![s_2]\!]$ for $\mathbb{D}' = \mathbb{D} \cdot \{\bar{x} \mapsto \bar{v}\}$. Indeed, since s_1 and s_2 are closed terms, $s_1 \cdot \{\bar{x} \mapsto \bar{v}\} = s_1$ and $s_2 \cdot \{\bar{x} \mapsto \bar{v}\} = s_2$. This means that for all $\bar{v}$ there exists a bisimulation $\mathcal{F}'$ such that $\mathbb{D}'[\![s_1]\!]\mathcal{R}''_\emptyset\mathbb{D}'[\![s_2]\!]$ with $\mathcal{R}''_\emptyset \in \mathcal{F}'$. By Lemma 4.3.1, we can prove the thesis by showing that $(\mathcal{F}' \backslash \mathcal{R}''_\emptyset) \cup \mathcal{R}'''_\emptyset$ is a bisimulation up-to $\equiv$, where

$$\mathcal{R}'''_\emptyset = \{(\mathtt{n}?\bar{w}\cdot\{\bar{x} \mapsto \bar{v}\}.\, \mathbb{D}'[\![s_1]\!]\,,\, \mathtt{n}?\bar{w}\cdot\{\bar{x} \mapsto \bar{v}\}.\, \mathbb{D}'[\![s_2]\!])\} \cup \mathcal{R}''_\emptyset$$

The rest of the proof proceeds as before.

- $\mathbb{C} = \mathbb{G} + g$.
  By induction, we may assume that $\mathbb{G}[\![s_1]\!] \sim_m \mathbb{G}[\![s_2]\!]$. If $\mathbb{G}$ is a closed context, then there exists a bisimulation $\mathcal{F}$ such that $\mathbb{G}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{G}[\![s_2]\!]$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. By Lemma 4.3.1, we can prove the thesis by showing that $(\mathcal{F} \backslash \mathcal{R}_\emptyset) \cup \mathcal{R}'_\emptyset$ is a bisimulation up-to $\equiv$, where
  $$\mathcal{R}'_\emptyset = \{(\mathbb{G}[\![s_1]\!] + g, \mathbb{G}[\![s_2]\!] + g)\} \cup \mathcal{R}_\emptyset \cup Id$$
  and Id is the identity relation. Indeed, if $\mathbb{G}[\![s_1]\!] + g \xrightarrow{\alpha} s$, due to the structure of $\mathbb{G}$ and g, then $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{v}$ and there are the following possibilities.
  - $g \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{v}} s$. Thus, $\mathbb{G}[\![s_2]\!] + g \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{v}} s$ and $s\,\mathcal{R}'_\emptyset\,s$, since $s\,Id\,s$ and $Id \subseteq \mathcal{R}'_\emptyset$.
  - $\mathbb{G}[\![s_1]\!] \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{v}} s$. From the hypothesis $\mathbb{G}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{G}[\![s_2]\!]$, then we have two possibilities:
    * There exists s' such that $\mathbb{G}[\![s_2]\!] \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{v}} s'$ and, for all $\bar{v}$ such that $\mathcal{M}(\bar{x}, \bar{v}) = \sigma$, we have $s \cdot \sigma\,\mathcal{R}_\emptyset\,s' \cdot \sigma$. Then, $\mathbb{G}[\![s_2]\!] + g \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{v}} s'$. Since $\mathcal{R}_\emptyset \subseteq \mathcal{R}'_\emptyset$, we get $s \cdot \sigma\,\mathcal{R}'_\emptyset\,s' \cdot \sigma$.
    * There exists s' such that $\mathbb{G}[\![s_2]\!] \xrightarrow{\emptyset} s'$ and, for all $\bar{v}$ such that $\mathcal{M}(\bar{x}, \bar{v}) = \sigma$, we have $s \cdot \sigma\,\mathcal{R}_\emptyset\,(s' \mid \mathtt{n}!(\bar{w} \cdot \sigma))$. Then, $\mathbb{G}[\![s_2]\!] + g \xrightarrow{\emptyset} s'$ and, since $\mathcal{R}_\emptyset \subseteq \mathcal{R}'_\emptyset$, we get $s \cdot \sigma\,\mathcal{R}'_\emptyset\,(s' \mid \mathtt{n}!(\bar{w} \cdot \sigma))$.

  If $\mathbb{C}$ is an open context, we can proceed similarly to the case $\mathbb{C} = \mathtt{n}?\bar{w}.\,\mathbb{D}$ when $\mathbb{C}$ contains free variables.

- $\mathbb{C} = g + \mathbb{G}$.
  We can proceed as in the case $\mathbb{C} = \mathbb{G} + g$.

- $\mathbb{C} = [x]\,\mathbb{D}$.
  If $\mathbb{D}$ is a closed context, then $[x]\,\mathbb{D} \equiv \mathbb{D}$ and, by induction, we immediately conclude. Instead, if $\mathbb{D}$ contains the free variable x at most[1], by induction and since

[1]The case where $\mathbb{D}$ contains more than one free variable can be dealt with in a similar way.

s_1 and s_2 are closed terms, it holds that, for all v, $\mathbb{D}_v[\![s_1]\!] \sim_m \mathbb{D}_v[\![s_2]\!]$, where $\mathbb{D}_v$=$\mathbb{D} \cdot \{x \mapsto v\}$. This means that, for all v, there exists a bisimulation $\mathcal{F}^v$ such that $\mathbb{D}_v[\![s_1]\!]\, \mathcal{R}^v_\emptyset\, \mathbb{D}_v[\![s_2]\!]$, with $\mathcal{R}^v_\emptyset \in \mathcal{F}^v$. Now, consider the following family of relations:

$$\mathcal{F} = \{\mathcal{R}^{[x]}_{\mathcal{N}} \ : \ \mathcal{N} \text{ is a set of names}\} \cup \bigcup_v \mathcal{F}^v$$

where $\mathcal{R}^{[x]}_{\mathcal{N}} = \{([x]\, s, [x]\, s') \ : \ \forall v \ \ s \cdot \{x \mapsto v\}\, \mathcal{R}^v_{\mathcal{N}}\, s' \cdot \{x \mapsto v\}\,,\ \mathcal{R}^v_{\mathcal{N}} \in \mathcal{F}^v\,,\ s \text{ and } s' \text{ satisfy } (*)\}$; property $(*)$ states that if $s \xrightarrow{\alpha}$ where $\alpha = \mathtt{n} \rhd [\bar{y}]\, \bar{w}$, with $x \in \bar{w}$, or $\alpha = \sigma \uplus \{x \mapsto v\}$, then $s' \xrightarrow{\alpha}$, and vice versa; $\bigcup_v \mathcal{F}^v = \{\mathcal{R}_{\mathcal{N}} \ : \ \mathcal{N} \text{ is a set of names}\,,\ \mathcal{R}^v_{\mathcal{N}} \in \mathcal{F}^v\,,\ \mathcal{R}_{\mathcal{N}} = \bigcup_v \mathcal{R}^v_{\mathcal{N}}\}$. The proof proceeds by proving that $\mathcal{F}$ is a bisimulation up-to $\equiv$. Indeed, by letting $s = \mathbb{D}[\![s_1]\!]$ and $s' = \mathbb{D}[\![s_2]\!]$, we obtain the thesis. In fact, $\mathbb{D}_v[\![s_1]\!]\, \mathcal{R}^v_\emptyset\, \mathbb{D}_v[\![s_2]\!]$ for all v, and $\mathbb{D}[\![s_1]\!]$ and $\mathbb{D}[\![s_2]\!]$ satisfy property $(*)$, because x does not occur free in s_1 and s_2 (since s_1 and s_2 are closed terms).

Thus, let us consider a relation $\mathcal{R}^{[x]}_{\mathcal{N}} \in \mathcal{F}$. If $[x]\, s \xrightarrow{\alpha} s''$, we proceed by case analysis on α. The most interesting case is $\alpha = \mathtt{n} \rhd [x, \bar{y}]\, \bar{w}$. By rule *(open$_{rec}$)*, we get that $s \xrightarrow{\mathtt{n} \rhd [\bar{y}]\, \bar{w}} s''$. From this, for all v, $s \cdot \{x \mapsto v\} \xrightarrow{\mathtt{n} \rhd [\bar{y}]\, (\bar{w} \cdot \{x \mapsto v\})} s'' \cdot \{x \mapsto v\}$. By definition of $\mathcal{R}^{[x]}_{\mathcal{N}}$, for all v, we get that $s \cdot \{x \mapsto v\} \mathcal{R}^v_{\mathcal{N}} s' \cdot \{x \mapsto v\}$, where $\mathcal{R}^v_{\mathcal{N}}$ belongs to a bisimulation. Hence, we would have two possibilities: $s' \cdot \{x \mapsto v\} \xrightarrow{\mathtt{n} \rhd [\bar{y}]\, (\bar{w} \cdot \{x \mapsto v\})}$ or $s' \cdot \{x \mapsto v\} \xrightarrow{\emptyset}$. In the former case, because of property $(*)$, we get that $s' \cdot \{x \mapsto v\} \xrightarrow{\mathtt{n} \rhd [\bar{y}]\, (\bar{w} \cdot \{x \mapsto v\})} s''' \cdot \{x \mapsto v\}$ and, for all $\bar{v}'$ s.t. $\mathcal{M}(\bar{y}, \bar{v}') = \sigma$, $s'' \cdot \{x \mapsto v\} \uplus \sigma\, \mathcal{R}^v_{\mathcal{N}}\, s''' \cdot \{x \mapsto v\} \uplus \sigma$, where s''' is such that $s' \xrightarrow{\mathtt{n} \rhd [\bar{y}]\, \bar{w}} s'''$. By rule *(open$_{rec}$)*, we conclude that $[x]\, s' \xrightarrow{\mathtt{n} \rhd [x, \bar{y}]\, \bar{w}} s'''$.

In the latter case, i.e. when $\alpha = \emptyset$, we have two possibilities. In one case, $s \xrightarrow{\emptyset} s'''$ with $s'' \equiv [x]\, s'''$. Then, for all v, $s \cdot \{x \mapsto v\} \xrightarrow{\emptyset} s''' \cdot \{x \mapsto v\}$. By definition of $\mathcal{R}^{[x]}_{\mathcal{N}}$, there exits s'''' such that $s' \cdot \{x \mapsto v\} \xrightarrow{\emptyset} s'''' \cdot \{x \mapsto v\}$ and $s''' \cdot \{x \mapsto v\} \mathcal{R}^v_{\mathcal{N}} s'''' \cdot \{x \mapsto v\}$, for some $\mathcal{R}^v_{\mathcal{N}}$ belonging to a bisimulation. Since the communication does not involve x, we get that $s' \xrightarrow{\emptyset} s''''$ and, hence, $[x]\, s' \xrightarrow{\emptyset} [x]\, s''''$. We conclude by noticing that, by definition of $\mathcal{R}^{[x]}_{\mathcal{N}}$, it holds that $[x]\, s''' \mathcal{R}^{[x]}_{\mathcal{N}}\, [x]\, s''''$. The other possibility is that $s \xrightarrow{\{x \mapsto v'\}} s'''$ with $s'' = s''' \cdot \{x \mapsto v'\}$. Then, for all v, $s \xrightarrow{\emptyset} s''' \cdot \{x \mapsto v\}$. By definition of $\mathcal{R}^{[x]}_{\mathcal{N}}$, there exits s'''' such that $s' \cdot \{x \mapsto v\} \xrightarrow{\emptyset} s'''' \cdot \{x \mapsto v\}$ and $s''' \cdot \{x \mapsto v\} \mathcal{R}^v_{\mathcal{N}} s'''' \cdot \{x \mapsto v\}$, for some $\mathcal{R}^v_{\mathcal{N}}$ belonging to a bisimulation. By property $(*)$, $s' \xrightarrow{\{x \mapsto v'\}} s''''$ and, hence, $[x]\, s' \xrightarrow{\emptyset} s'''' \cdot \{x \mapsto v'\}$. Finally, from $s''' \cdot \{x \mapsto v\} \mathcal{R}^v_{\mathcal{N}} s'''' \cdot \{x \mapsto v\}$, by letting $v = v'$ and by definition of $\mathcal{R}^{[x]}_{\mathcal{N}}$, we get that $s''' \cdot \{x \mapsto v'\} \mathcal{R}^{[x]}_{\mathcal{N}}\, s'''' \cdot \{x \mapsto v'\}$.

- $\mathbb{C} = \mathbb{D} \mid s$.
  By induction, we may assume that $\mathbb{D}[\![s_1]\!] \sim_m \mathbb{D}[\![s_2]\!]$. If $\mathbb{D}$ is a closed context, then there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. By Lemma 4.3.1, we can prove the thesis by showing that the family of relations

  $$\mathcal{F}' = \{\, \mathcal{R}'_{\mathcal{N}'} \;:\; \mathcal{R}_\mathcal{N} \in \mathcal{F}\,,\, \mathcal{N} \subseteq \mathcal{N}' \,\}$$
  $$\mathcal{R}'_{\mathcal{N}'} = \{\, (\,[\bar{n}]\,(s' \mid s), [\bar{n}]\,(s'' \mid s)\,) \;:\; s'\mathcal{R}_\mathcal{N}s''\,, \text{ for some } \bar{n} \text{ and } s \text{ s.t. } \mathcal{N} \cap \mathrm{re}(s) = \emptyset \,\}$$

  where $\mathrm{re}(s)$ is the set of the receiving endpoint used in s, is a bisimulation up-to $\equiv$. Notice that for each relation $\mathcal{R}_\mathcal{N} \in \mathcal{F}$ there exists a family of relations $\mathcal{R}'_{\mathcal{N}'} \in \mathcal{F}'$ for $\mathcal{N} \subseteq \mathcal{N}'$ such that $\mathcal{R}_\mathcal{N} \subseteq \mathcal{R}'_{\mathcal{N}'}$ (in fact, s can be $\mathbf{0}$ and $\bar{n}$ can be empty).

  Let us consider a relation $\mathcal{R}'_{\mathcal{N}'} \in \mathcal{F}'$. If $[\bar{n}]\,(s' \mid s) \xrightarrow{\alpha} s'''$, then the proof proceeds by case analysis on α. We only take a look at the cases of bound invocation and communication of private names.

  - $\alpha = \mathtt{n} \lhd [\bar{m}]\,\bar{v}$
    We have two possibilities:
    * $s \xrightarrow{\mathtt{n}\lhd[\bar{m}']\,\bar{v}} s_3$, $\bar{m}' \subseteq \bar{m}$, $s''' \equiv [\bar{n}\backslash\bar{m}]\,(s' \mid s_3)$.
      If $\mathtt{n} \in \mathcal{N}'$ then we immediately conclude, because Definition 4.3.4 does not impose any requirement in this case. Instead, if $\mathtt{n} \notin \mathcal{N}'$ then $[\bar{n}]\,(s'' \mid s) \xrightarrow{\mathtt{n}\lhd[\bar{m}]\,\bar{v}} [\bar{n}\backslash\bar{m}]\,(s'' \mid s_3)$. Since $s'\mathcal{R}_\mathcal{N}s''$, by definition of $\mathcal{F}'$ we get that $[\bar{n}\backslash\bar{m}]\,(s' \mid s_3)\,\mathcal{R}'_{\mathcal{N}'\cup\bar{m}}\,[\bar{n}\backslash\bar{m}]\,(s'' \mid s_3)$.
    * $s' \xrightarrow{\mathtt{n}\lhd[\bar{m}']\,\bar{v}} s_3$, $\bar{m}' \subseteq \bar{m}$, $s''' \equiv [\bar{n}\backslash\bar{m}]\,(s_3 \mid s)$.
      If $\mathtt{n} \in \mathcal{N}'$ then we immediately conclude. Otherwise, since $s'\mathcal{R}_\mathcal{N}s''$ for $\mathcal{R}_\mathcal{N}$ belonging to the bisimulation $\mathcal{F}$, we get that there exists s_4 such that $s'' \xrightarrow{\mathtt{n}\lhd[\bar{m}']\,\bar{v}} s_4$ and $s_3\,\mathcal{R}_{\mathcal{N}\cup\bar{m}'}\,s_4$. Thus, we have that $[\bar{n}]\,(s'' \mid s) \xrightarrow{\mathtt{n}\lhd[\bar{m}]\,\bar{v}} [\bar{n}\backslash\bar{m}]\,(s_4 \mid s)$. From this and by definition of $\mathcal{F}'$ we get that $[\bar{n}\backslash\bar{m}]\,(s_3 \mid s)\,\mathcal{R}_{\mathcal{N}'\cup\bar{m}}\,[\bar{n}\backslash\bar{m}]\,(s_4 \mid s)$.
  - $\alpha = \emptyset$, $s' \xrightarrow{\mathtt{n}\lhd[\bar{m}]\,\bar{v}} s_3$, $s \equiv [\bar{x}]\,s_4 \xrightarrow{\mathtt{n}\rhd[\bar{x}]\,\bar{w}} s_5$, $|\bar{m}|=|\bar{x}|$.
    By rules *(str)*, *(del_{com})* and *(com)*, we get that $(s' \mid s) \equiv [\bar{x}]\,(s' \mid s_4)$, $(s' \mid s_4) \xrightarrow{\{\bar{x}\mapsto\bar{m}\}} (s_3 \mid s_5)$, $\mathcal{M}(\bar{w},\bar{v}) = \{\bar{x} \mapsto \bar{m}\}$, and $s''' \equiv [\bar{n},\bar{m}]\,(s_3 \mid s_5 \cdot \{\bar{x} \mapsto \bar{m}\})$. Since $\mathtt{n}$ is a receiving endpoint of s, then $\mathtt{n} \notin \mathcal{N}$. From this and since $s'\mathcal{R}_\mathcal{N}s''$ for $\mathcal{R}_\mathcal{N}$ belonging to the bisimulation $\mathcal{F}$, we get that there exists s_6 such that $s'' \xrightarrow{\mathtt{n}\lhd[\bar{m}]\,\bar{v}} s_6$ and $s_3\mathcal{R}_{\mathcal{N}\cup\bar{m}}s_6$, where $\mathcal{R}_{\mathcal{N}\cup\bar{m}} \in \mathcal{F}$. Hence, $[\bar{n}]\,(s'' \mid s) \xrightarrow{\emptyset} [\bar{n},\bar{m}]\,(s_6 \mid s_5 \cdot \{\bar{x} \mapsto \bar{m}\})$. Since $s_3\mathcal{R}_{\mathcal{N}\cup\bar{m}}s_6$, by definition of $\mathcal{F}'$, we conclude that $[\bar{n},\bar{m}]\,(s_3 \mid s_5\cdot\{\bar{x} \mapsto \bar{m}\})\mathcal{R}'_{\mathcal{N}'\cup\bar{m}}[\bar{n},\bar{m}]\,(s_6 \mid s_5\cdot\{\bar{x} \mapsto \bar{m}\})$. The generalisation to the case $|\bar{m}|<|\bar{x}|$, where also some non-restricted values are communicated, is trivial.

By letting $s' = \mathbb{D}[\![s_1]\!]$ and $s'' = \mathbb{D}[\![s_2]\!]$, we obtain the thesis. We can generalise to the case where $\mathbb{C}$ is an open context as in the previous cases.

- $\mathbb{C} = s \mid \mathbb{D}$.
  We can proceed as in the case $\mathbb{C} = \mathbb{D} \mid s$.

- $\mathbb{C} = [n]\,\mathbb{D}$.
  By letting $s = \mathbf{0}$ in the family of relations defined in the case $\mathbb{C} = \mathbb{D} \mid s$, we obtain the thesis.

- $\mathbb{C} = *\,\mathbb{D}$.
  By induction, we may assume that $\mathbb{D}[\![s_1]\!] \sim_m \mathbb{D}[\![s_2]\!]$. If $\mathbb{D}$ is a closed context, then there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. By Lemma 4.3.1, we can prove the thesis by showing that the family of relations

  $$\{\,\{\,(*\,s \mid s'', *\,s' \mid s'')\ :\ s\mathcal{R}_\mathcal{N}s'\text{, for some } s''\,\}\ :\ \mathcal{R}_\mathcal{N} \in \mathcal{F}\,\}$$

  is a bisimulation up-to $\equiv$. The proof proceeds similarly to that for the bisimulation defined in the case $\mathbb{D} \mid s$.

  By letting $s = \mathbb{D}[\![s_1]\!]$, $s' = \mathbb{D}[\![s_2]\!]$ and $s'' = \mathbf{0}$, we obtain the thesis. We can generalise to the case where $\mathbb{C}$ is an open context as in the previous cases.

□

Theorem B.2.2 (Soundness of $\sim_m$ w.r.t. $\simeq_m$, Theorem 4.3.2) *Given two μCOWS^m closed terms s_1 and s_2, if $s_1 \sim_m s_2$ then $s_1 \simeq_m s_2$.*

Proof. By Theorem B.2.1, we have that $\sim_m$ is context closed. Thus, we only need to prove that $\sim_m$ is barb preserving and computation closed.

- *Barb preservation.* Suppose $s_1 \downarrow_\mathrm{n}$. Then, by Definition 4.3.1, this means that $s_1 \xrightarrow{\mathrm{n} \lhd [\bar{n}]\,\bar{v}} s_1'$, for some s_1', $\bar{n}$ and $\bar{v}$. Since $\sim_m = \sim_m^\emptyset$, by Definition 4.3.4, we get that $s_2 \xrightarrow{\mathrm{n} \lhd [\bar{n}]\,\bar{v}} s_2'$, for some s_2'. Thus, by Definition 4.3.1, $s_2 \downarrow_\mathrm{n}$.

- *Computation closure.* By hypothesis, there exists a labelled bisimulation $\mathcal{F}$ such that $s_1\mathcal{R}_\emptyset s_2$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. Thus, if $s_1 \xrightarrow{\emptyset} s_1'$, by Definition 4.3.4, we get that $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1'\mathcal{R}_\emptyset s_2'$. This means that $s_1' \sim_m s_2'$.

□

Theorem B.2.3 (Completeness of $\sim_m$ w.r.t. $\simeq_m$, Theorem 4.3.3) *Given two μCOWS^m closed terms s_1 and s_2, if $s_1 \simeq_m s_2$ then $s_1 \sim_m s_2$.*

Proof. We define a family of relations $\mathcal{F} = \{\mathcal{R}_{\mathcal{N}} \ : \ \mathcal{N} \text{ set of names}\}$ such that $\simeq_m$ is included in $\mathcal{R}_\emptyset$ and show that it is a labelled bisimulation. Let $\mathcal{N}$ be the set $\{n_1, \ldots, n_m\}$, then $s_1 \mathcal{R}_{\mathcal{N}} s_2$ if there exist $\mathtt{m}_1, \ldots, \mathtt{m}_m$ fresh such that

$$[n_1, \ldots, n_m]\,(s_1 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle) \simeq_m [n_1, \ldots, n_m]\,(s_2 \mid \mathtt{m}_1!\langle n_1\rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m\rangle)$$

Take $s_1 \mathcal{R}_{\mathcal{N}} s_2$ and a transition $s_1 \xrightarrow{\alpha} s_1'$; we then reason by case analysis on α. For the sake of simplicity, we consider in detail the case where $\mathcal{N} = \emptyset$, i.e. we assume $s_1 \simeq_m s_2$. We have to consider three cases, one for each clause in Definition 4.3.4.

- $\alpha = \emptyset$.
  By computation closure of $\simeq_m$, we have that $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \simeq_m s_2'$. $\simeq_m \subseteq \mathcal{R}_\emptyset$, hence $s_1' \mathcal{R}_\emptyset s_2'$, as required.

- $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$.
  We consider first the case where all sent values are not restricted names, i.e. $\alpha = \mathtt{n} \lhd \bar{v}$. We let the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}?\bar{v}.\mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle$ for $\mathtt{m}$ fresh, and consider the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\emptyset} \mathbb{C}'[\![s_1']\!]$, where $\mathbb{C}' = [\![\cdot]\!] \mid \mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle$. By hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s$ and $\mathbb{C}'[\![s_1']\!] \simeq_m s$. Since $\mathtt{m}$ is fresh, this fact implies that $s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2'$ and $s \equiv \mathbb{C}'[\![s_2']\!]$, otherwise s would not be able to exhibit a barb $\downarrow_{\mathtt{m}}$ (whereas $\mathbb{C}'[\![s_1']\!]$ can). Now, we consider the computation $\mathbb{C}'[\![s_1']\!] \xrightarrow{\emptyset} s_1'$. By hypothesis and the fact that s_1' is not be able to exhibit the barb $\downarrow_{\mathtt{m}}$, $\mathbb{C}'[\![s_2']\!] \xrightarrow{\emptyset} s_2'$ and $s_1' \simeq_m s_2'$. Since by definition $\simeq_m \subseteq \mathcal{R}_\emptyset$, then we get $s_1' \mathcal{R}_\emptyset s_2'$, as required.

  Now, we consider the more general case where $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$, with $\bar{n} = \langle n_1, \ldots, n_m\rangle$ and $m \neq 0$. For the sake of presentation, suppose that $\bar{v} = (\bar{n}, \bar{v}')$. Take the context $\mathbb{C} = [\![\cdot]\!] \mid [x_1, \ldots, x_m]\,\mathtt{n}?(x_1, \ldots, x_m, \bar{v}').(\mathtt{m}!\langle x_1\rangle \mid \ldots \mid \mathtt{m}!\langle x_m\rangle)$, for $\mathtt{m}$ fresh, and consider the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\emptyset} s_3$ with $s_3 = [\bar{n}]\,(s_1' \mid \mathtt{m}!\langle n_1\rangle \mid \ldots \mid \mathtt{m}!\langle n_m\rangle)$. By hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s_4$ and $s_3 \simeq_m s_4$. Since $s_3 \downarrow_{\mathtt{m}}$, by barb preservation, we get $s_4 \downarrow_{\mathtt{m}}$. This fact implies that s_2 has performed an invoke activity $\mathtt{n}!(\bar{v}'', \bar{v}')$ matching $\mathtt{n}?(x_1, \ldots, x_m, \bar{v}')$. If $\bar{v}''$ would contain some non-restricted names, e.g. v''', then we could define a context $\mathbb{D} = [\![\cdot]\!] \mid \mathtt{m}?\langle v'''\rangle.\mathtt{m}'!\langle\rangle$, for $\mathtt{m}'$ fresh, that can tell s_3 and s_4 apart. Indeed, $\mathbb{D}[\![s_4]\!] \xrightarrow{\emptyset} s_5$ with $s_5 \downarrow_{\mathtt{m}'}$, while for each s_6 such that $\mathbb{D}[\![s_3]\!] \xrightarrow{\emptyset} s_6$, it holds that $s_6 \not\downarrow_{\mathtt{m}'}$, that would contradict $s_5 \simeq_m s_6$ (implied by the hypothesis $s_3 \simeq_m s_4$). Thus, $\bar{v}''$ is a tuple of restricted names and, possibly by exploiting α-conversion, we get that $s_2 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s_2'$ and $s_4 = [\bar{n}]\,(s_2' \mid \mathtt{m}!\langle n_1\rangle \mid \ldots \mid \mathtt{m}!\langle n_m\rangle)$. From $s_3 \simeq_m s_4$ and by definition of $\mathcal{F}$, we conclude that $s_1' \mathcal{R}_{\bar{n}} s_2'$.

- $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$.
  For the sake of presentation, suppose that $\bar{w} = (\bar{x}, \bar{v})$. We consider the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}!(\bar{v}', \bar{v})$, for $\bar{v}'$ fresh, and the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\emptyset} s_1' \cdot \{\bar{x} \mapsto \bar{v}'\}$.

By computation closure, $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s_3$ and $s_1' \cdot \{\bar{x} \mapsto \bar{v}'\} \simeq_m s_3$. We have two possibilities:

- $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\, \bar{w}} s_2'$ and $s_3 = s_2' \cdot \{\bar{x} \mapsto \bar{v}'\}$;
- $s_2 \xrightarrow{\emptyset} s_2'$ and $s_3 = s_2' \mid \mathtt{n}!(\bar{v}', \bar{v})$.

In both cases the thesis follows from the fact that, by definition of $\mathcal{F}$, $\simeq_m \subseteq \mathcal{R}_\emptyset$.

Let us now consider the more general case $\mathcal{N} \neq \emptyset$. Let $\mathcal{N}$ be the set $\{n_1, \ldots, n_k\} = \bar{n}$. As before, we have three possibilities. We show here the most interesting case: $s_1 \xrightarrow{\mathtt{n} \lhd [\bar{m}]\, \bar{v}} s_1'$ with $\mathtt{n} \notin \mathcal{N}$, $\bar{m} = \langle m_1, \ldots, m_r \rangle$ and $\bar{v} = (\bar{m}, \bar{v}')$; the other cases can be dealt with in a similar way. Given $\mathtt{m}_1, \ldots, \mathtt{m}_k$ fresh, since $\mathtt{n} \notin \mathcal{N}$, we get that $[\bar{n}]\,(s_1 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle) \xrightarrow{\mathtt{n} \lhd [\bar{m}]\, \bar{v}} [\bar{n}]\,(s_1' \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle)$. Now, consider the context $\mathbb{C} = [\![\cdot]\!] \mid [x_1, \ldots, x_r]\, \mathtt{n}?(\langle x_1, \ldots, x_r \rangle, \bar{v}').\, (\mathtt{m}'!\langle x_1 \rangle \mid \ldots \mid \mathtt{m}'!\langle x_r \rangle)$, for $\mathtt{m}'$ fresh, and the computation $\mathbb{C}[\![[\bar{n}]\,(s_1 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle)]\!] \xrightarrow{\emptyset} s_3$ with $s_3 = [\bar{n}, \bar{m}]\,(s_1' \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle \mid \mathtt{m}'!\langle m_1 \rangle \mid \ldots \mid \mathtt{m}'!\langle m_r \rangle)$. As we have done for the case $\mathcal{N} = \emptyset$, we can prove that $\mathbb{C}[\![[\bar{n}]\,(s_2 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle)]\!] \xrightarrow{\emptyset} s_4$, $[\bar{n}]\,(s_2 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle) \xrightarrow{\mathtt{n} \lhd [\bar{m}]\, \bar{v}} [\bar{n}]\,(s_2' \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle)$, $s_4 = [\bar{n}, \bar{m}]\,(s_2' \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_k!\langle n_k \rangle \mid \mathtt{m}'!\langle m_1 \rangle \mid \ldots \mid \mathtt{m}'!\langle m_r \rangle)$, and $s_3 \simeq_m s_4$. From this and by definition of $\mathcal{F}$, we conclude that $s_1' \mathcal{R}_{\bar{n} \cup \bar{m}} s_2'$. □

B.2.2 μCOWS

Lemma B.2.1 (Lemma 4.3.3) *Let s_1 and s_2 be two* μCOWS *closed terms and $\mathcal{R}$ be a relation belonging to a labelled bisimulation such that $s_1 \mathcal{R} s_2$. Then,* $\mathrm{noConf}(s_1, \mathtt{n}, \bar{v}, \ell) = \mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell)$ *for any* $\mathtt{n}$, $\bar{v}$ *and* ℓ, *with* $\ell \leqslant |\bar{v}|$.

Proof. The proof proceeds by contradiction. Suppose that there exists $\mathtt{n}$, $\bar{v}$ and $\ell \leqslant |\bar{v}|$ such that $\mathrm{noConf}(s_1, \mathtt{n}, \bar{v}, \ell) = \mathbf{false}$ and $\mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$. By definition, $\mathrm{noConf}(s_1, \mathtt{n}, \bar{v}, \ell) = \mathbf{false}$ implies that there exists a context $\mathbb{C}$ such that $s_1 = \mathbb{C}[\![\mathtt{n}?\bar{w}.s]\!]$ and s_1 can immediately perform the receive activity $\mathtt{n}?\bar{w}$ and $|\mathcal{M}(\bar{w}, \bar{v})| < \ell$. This means that there exists s_1' such that $s_1 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\, \bar{w}} s_1'$ for some $\bar{x} \subseteq \bar{w}$. Since $|\mathcal{M}(\bar{w}, \bar{v})| < \ell \leqslant |\bar{v}|$, we have that $|\bar{x}| \neq |\bar{w}|$. Hence, since $s_1 \mathcal{R} s_2$ for $\mathcal{R}$ belonging to a labelled bisimulation, there exists s_2' such that $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\, \bar{w}} s_2'$. From this, we get that $s_2 = \mathbb{D}[\![\mathtt{n}?\bar{w}.s]\!]$ for some context $\mathbb{D}$ such that s_2 can immediately perform the receive activity $\mathtt{n}?\bar{w}$. Thus, by definition of predicate $\mathrm{noConf}(\_, \_, \_, \_)$, we obtain $\mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell) = \mathbf{false}$, that is a contradiction. Of course, the case where $\mathrm{noConf}(s_1, \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$ and $\mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell) = \mathbf{false}$ proceeds as before. □

Theorem B.2.4 (Theorem 4.3.4) *$\sim_\mu$ is a congruence for* μCOWS *closed terms.*

Proof. We shall prove that, given two μCOWS closed terms s_1 and s_2, if $s_1 \sim_\mu s_2$ then $\mathbb{C}[\![s_1]\!] \sim_\mu \mathbb{C}[\![s_2]\!]$ for every (possibly open) context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$ and proceeds similarly to that of Theorem B.2.1. Here, we take a look only at the most relevant case of the inductive step, i.e. the case $\mathbb{C} = \mathbb{D} \mid s$ regarding the parallel composition. By induction, we may assume that $\mathbb{D}[\![s_1]\!] \sim_\mu \mathbb{D}[\![s_2]\!]$. If $\mathbb{D}$ is a closed context, then there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. By Lemma 4.3.2, we can prove the thesis by showing that the family of relations

$$\mathcal{F}' = \{\, \mathcal{R}'_{\mathcal{N}'} \;:\; \mathcal{R}_\mathcal{N} \in \mathcal{F}\,,\ \mathcal{N} \subseteq \mathcal{N}' \,\}$$

$$\mathcal{R}'_{\mathcal{N}'} = \{\, (\,[\bar{n}]\,(s' \mid s), [\bar{n}]\,(s'' \mid s)\,) \;:\; s'\mathcal{R}_\mathcal{N}s''\,, \text{ for some } \bar{n} \text{ and } s \text{ s.t. } \mathcal{N} \cap \mathrm{re}(s) = \emptyset \,\}$$

where $\mathrm{re}(s)$ is the set of the receiving endpoint used in s, is a bisimulation up-to $\equiv$. Let us consider a relation $\mathcal{R}'_{\mathcal{N}'} \in \mathcal{F}'$. If $[\bar{n}]\,(s' \mid s) \xrightarrow{\alpha} s'''$, then the proof proceeds by case analysis on α. We consider here only a few relevant cases.

- $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$, $s \xrightarrow{\alpha} s''''$ and $s''' = [\bar{n}]\,(s' \mid s'''')$.
  By rule *(par$_{com}$)*, since $s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s''''$ and $s' \mid s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s' \mid s''''$, we get that $\mathrm{noConf}(s', \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$. Since $s'\mathcal{R}_\mathcal{N}s''$ for a relation $\mathcal{R}_\mathcal{N}$ belonging to the bisimulation $\mathcal{F}$, by Lemma B.2.1 we have that $\mathrm{noConf}(s'', \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$. Hence, by rules *(par$_{com}$)* and *(del$_2$)*, $[\bar{n}]\,(s'' \mid s) \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} [\bar{n}]\,(s'' \mid s'''')$. By definition of $\mathcal{F}'$, we conclude that $[\bar{n}]\,(s' \mid s'''')\,\mathcal{R}'_{\mathcal{N}'}\,[\bar{n}]\,(s'' \mid s'''')$.

- $\alpha = \mathtt{n}\,\emptyset\ |\bar{v}|\ \bar{v}$, $s' \xrightarrow{\alpha} s_3$ and $s''' = [\bar{n}]\,(s_3 \mid s)$.
  By rule *(par$_{com}$)*, since $s' \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} s_3$ and $s' \mid s \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} s_3 \mid s$, we get that $\mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$. Since $s'\mathcal{R}_\mathcal{N}s''$ for a relation $\mathcal{R}_\mathcal{N}$ belonging to the bisimulation $\mathcal{F}$, we have two possibilities:

  1. there exists s_4 such that $s'' \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} s_4$ and $s_3\mathcal{R}_\mathcal{N}s_4$. Since $\mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) = \mathbf{true}$ and by rules *(par$_{com}$)* and *(del$_2$)*, $[\bar{n}]\,(s'' \mid s) \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} [\bar{n}]\,(s_4 \mid s)$. By definition of $\mathcal{F}'$, we conclude that $[\bar{n}]\,(s_3 \mid s)\,\mathcal{R}'_{\mathcal{N}'}\,[\bar{n}]\,(s_4 \mid s)$.
  2. there exists s_4 such that $s'' \xrightarrow{\emptyset} s_4$ and $s_3\mathcal{R}_\mathcal{N}s_4$. By rules *(par$_2$)* and *(del$_2$)*, $[\bar{n}]\,(s'' \mid s) \xrightarrow{\emptyset} [\bar{n}]\,(s_4 \mid s)$. By definition of $\mathcal{F}'$, we conclude that $[\bar{n}]\,(s_3 \mid s)\,\mathcal{R}'_{\mathcal{N}'}\,[\bar{n}]\,(s_4 \mid s)$.

- $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$, $\ell \neq |\bar{v}|$, $s' \xrightarrow{\alpha} s_3$ and $s''' = [\bar{n}]\,(s_3 \mid s)$.
  Like the case 1 above.

By letting $s' = \mathbb{D}[\![s_1]\!]$ and $s'' = \mathbb{D}[\![s_2]\!]$, we obtain the thesis. We can generalise to the case where $\mathbb{C}$ is an open context as shown in Theorem B.2.1. □

Theorem B.2.5 (Soundness of $\sim_\mu$ w.r.t. $\simeq_\mu$, Theorem 4.3.5) *Given two μ*COWS *closed terms s_1 and s_2, if $s_1 \sim_\mu s_2$ then $s_1 \simeq_\mu s_2$.*

Proof. By Theorem B.2.4, we have that $\sim_\mu$ is context closed. Thus, we only need to prove that $\sim_\mu$ is barb preserving and computation closed.

- *Barb preservation.* Suppose $s_1 \downarrow_{\mathtt{n}}$. Then, by Definition 4.3.1, this means that $s_1 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\, \bar{v}} s_1'$, for some s_1', $\bar{n}$ and $\bar{v}$. Since $\sim_\mu = \sim_\mu^{\emptyset}$, by Definition 4.3.8, we get that $s_2 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\, \bar{v}} s_2'$, for some s_2'. Thus, by Definition 4.3.1, $s_2 \downarrow_{\mathtt{n}}$.

- *Computation closure.* By hypothesis, there exists a labelled bisimulation $\mathcal{F}$ such that $s_1 \mathcal{R}_\emptyset s_2$ with $\mathcal{R}_\emptyset \in \mathcal{F}$. Thus, if $s_1 \xrightarrow{\emptyset} s_1'$, by Definition 4.3.8, we get that $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \mathcal{R}_\emptyset s_2'$, which means that $s_1' \sim_\mu s_2'$. Similarly, if $s_1 \xrightarrow{\mathtt{n}\, \emptyset\, \ell\, \bar{v}} s_1'$ then, by Definition 4.3.8, we get that $s_2 \xrightarrow{\alpha} s_2'$ and $s_1' \sim_\mu s_2'$, where either $\alpha = \mathtt{n}\, \emptyset\, \ell\, \bar{v}$ or $(\ell = |\bar{v}| \wedge \alpha = \emptyset)$.

□

Theorem B.2.6 (Completeness of $\sim_\mu$ w.r.t. $\simeq_\mu$, Theorem 4.3.6) *Given two μCOWS closed terms s_1 and s_2, if $s_1 \simeq_\mu s_2$ then $s_1 \sim_\mu s_2$.*

Proof. We define a family of relations $\mathcal{F} = \{\mathcal{R}_\mathcal{N} \ : \ \mathcal{N} \text{ set of names}\}$ such that $\simeq_\mu$ is included in $\mathcal{R}_\emptyset$, and show that it is a labelled bisimulation. Let $\mathcal{N}$ be the set $\{n_1, \ldots, n_m\}$, then $s_1 \mathcal{R}_\mathcal{N} s_2$ if there exist $\mathtt{m}_1, \ldots, \mathtt{m}_m$ fresh such that

$$[n_1, \ldots, n_m]\,(s_1 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m \rangle) \simeq_\mu [n_1, \ldots, n_m]\,(s_2 \mid \mathtt{m}_1!\langle n_1 \rangle \mid \ldots \mid \mathtt{m}_m!\langle n_m \rangle)$$

Take $s_1 \mathcal{R}_\mathcal{N} s_2$ and a transition $s_1 \xrightarrow{\alpha} s_1'$; we then reason by case analysis on α. For the sake of simplicity, we consider here the case where $\mathcal{N} = \emptyset$ (the general case can be dealt with in a similar way, as shown in the proof of Theorem 4.3.3), i.e. we assume $s_1 \simeq_\mu s_2$. We have to consider the following possibilities.

- $\alpha = \emptyset$.
  By computation closure of $\simeq_\mu$, we have that $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \simeq_\mu s_2'$. $\simeq_\mu \subseteq \mathcal{R}_\emptyset$, hence $s_1' \mathcal{R}_\emptyset s_2'$, as required.

- $\alpha = \mathtt{n}\, \emptyset\, \ell\, \bar{v}$ and $\ell \neq |\bar{v}|$.
  By computation closure of $\simeq_\mu$, we have that $s_2 \xrightarrow{\mathtt{n}\, \emptyset\, \ell\, \bar{v}} s_2'$ and $s_1' \simeq_\mu s_2'$. As before, we obtain $s_1' \mathcal{R}_\emptyset s_2'$.

- $\alpha = \mathtt{n}\, \emptyset\ |\bar{v}|\ \bar{v}$.
  By computation closure of $\simeq_\mu$, we have two possibilities:

  1. $s_2 \xrightarrow{\mathtt{n}\, \emptyset\, \not|\, \bar{v}} s_2'$ and $s_1' \simeq_\mu s_2'$; hence, $s_1' \mathcal{R}_\emptyset s_2'$.
  2. $s_2 \xrightarrow{\emptyset} s_2'$ and $s_1' \simeq_\mu s_2'$; hence, $s_1' \mathcal{R}_\emptyset s_2'$.

- $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$.
  We consider first the case where all sent values are not restricted names, i.e. $\alpha = \mathtt{n} \lhd \bar{v}$. We let the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}?\bar{v}.\mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle$ for $\mathtt{m}$ fresh, and consider the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\emptyset} \mathbb{C}'[\![s_1']\!]$, where $\mathbb{C}' = [\![\cdot]\!] \mid \mathtt{m}!\langle\rangle \mid \mathtt{m}?\langle\rangle$. By hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s$ and $\mathbb{C}'[\![s_1']\!] \simeq_\mu s$. This fact implies that $s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2'$ and $s \equiv \mathbb{C}'[\![s_2']\!]$, otherwise s would not be able to exhibit a barb $\downarrow_{\mathtt{m}}$ (whereas $\mathbb{C}'[\![s_1']\!]$ can). Now, we consider the the computation $\mathbb{C}'[\![s_1']\!] \xrightarrow{\emptyset} s_1'$. By hypothesis and the fact that s_1' is not be able to exhibit the barb $\downarrow_{\mathtt{m}}$, $\mathbb{C}'[\![s_2']\!] \xrightarrow{\emptyset} s_2'$ and $s_1' \simeq_\mu s_2'$. Since by definition $\simeq_\mu \subseteq \mathcal{R}_\emptyset$, then we get $s_1' \mathcal{R}_\emptyset s_2'$, as required.

  Now, we consider the more general case where $\alpha = \mathtt{n} \lhd [\bar{n}]\,\bar{v}$, with $\bar{n} = \langle n_1, \ldots, n_m\rangle$ and $m \neq 0$. For the sake of presentation, suppose that $\bar{v} = (\bar{n}, \bar{v}')$. Take the context $\mathbb{C} = [\![\cdot]\!] \mid [x_1, \ldots, x_m]\,\mathtt{n}?(x_1, \ldots, x_m, \bar{v}').(\mathtt{m}!\langle x_1\rangle \mid \ldots \mid \mathtt{m}!\langle x_m\rangle)$, for $\mathtt{m}$ fresh, and the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\mathtt{n}\emptyset m\,\bar{v}} s_3$ with $s_3 = [\bar{n}]\,(s_1' \mid \mathtt{m}!\langle n_1\rangle \mid \ldots \mid \mathtt{m}!\langle n_m\rangle)$. Since $|\,\bar{v}\,| > m$, by hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\mathtt{n}\emptyset m\,\bar{v}} s_4$ and $s_3 \simeq_\mu s_4$. Since $s_3 \downarrow_{\mathtt{m}}$, by barb preservation, we get $s_4 \downarrow_{\mathtt{m}}$. This fact implies that s_2 has performed an invoke activity $\mathtt{n}!(\bar{v}'', \bar{v}')$ matching $\mathtt{n}?(x_1, \ldots, x_m, \bar{v}')$. If $\bar{v}''$ would contain some non-restricted names, e.g. v''', then we could define a context $\mathbb{D} = [\![\cdot]\!] \mid \mathtt{m}?\langle v'''\rangle.\mathtt{m}'!\langle\rangle$, for $\mathtt{m}'$ fresh, that can tell s_3 and s_4 apart. Indeed, $\mathbb{D}[\![s_4]\!] \xrightarrow{\emptyset} s_5$ with $s_5 \downarrow_{\mathtt{m}'}$, while for each s_6 such that $\mathbb{D}[\![s_3]\!] \xrightarrow{\emptyset} s_6$, it holds that $s_6 \not\downarrow_{\mathtt{m}'!}$, that would contradict $s_5 \simeq_\mu s_6$ (implied by the hypothesis $s_3 \simeq_\mu s_4$). Thus, $\bar{v}''$ is a tuple of restricted names and, possibly by exploiting α-conversion, we get that $s_2 \xrightarrow{\mathtt{n} \lhd [\bar{n}]\,\bar{v}} s_2'$ and $s_4 = [n]\,(s_2' \mid \mathtt{m}!\langle n_1\rangle \mid \ldots \mid \mathtt{m}!\langle n_m\rangle)$. From $s_3 \simeq_\mu s_4$ and by definition of $\mathcal{F}$, we conclude that $s_1' \mathcal{R}_{\bar{n}} s_2'$.

- $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$.
  We have the following possibilities:

  - $\alpha = \mathtt{n} \rhd \langle\rangle$.
    We consider the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}!\langle\rangle$ and the computation $\mathbb{C}[\![s_1]\!] \xrightarrow{\emptyset} s_1'$. By hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s$ and $s_1' \simeq_\mu s$. We have two possibilities:
    1. $s_2 \xrightarrow{\mathtt{n} \rhd \langle\rangle} s_2'$ and $s = s_2'$;
    2. $s_2 \xrightarrow{\emptyset} s_2'$ and $s = s_2' \mid \mathtt{n}!\langle\rangle$;

    In both cases the thesis follows from the fact that, by definition of $\mathcal{F}$, $\simeq_\mu \subseteq \mathcal{R}_\emptyset$.
  - $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{x}$.
    We consider the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}!\bar{v}$ for some $\bar{v}$ such that $\mathcal{M}(\bar{x}, \bar{v}) = \sigma$, $\mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, |\,\bar{x}\,|) = \mathbf{true}$ and $\mathbb{C}[\![s_1]\!] \xrightarrow{\mathtt{n}\emptyset|\bar{v}|\,\bar{v}} s_1' \cdot \sigma$. By hypothesis, we have two possibilities:

1. $\mathbb{C}[\![s_2]\!] \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} s$ and $s_1 \cdot \sigma \simeq_\mu s$. There are two cases:
   (a) $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{x}} s_2'$ and $s = s_2' \cdot \sigma$;
   (b) $s_2 \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{v}|\,\bar{v}} s_2''$. Since the length of the generated substitution is $|\,\bar{v}\,|$ (i.e. the argument of the executed receive is a tuple of only variables), by rule *(com)*, $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{x}} s_2'$ and, hence, we can proceed as in the previous case.
2. $\mathbb{C}[\![s_2]\!] \xrightarrow{\emptyset} s$ and $s_1' \simeq_\mu s$. This means that $s_2 \xrightarrow{\emptyset} s_2'$ and $s = s_2' \mid \mathtt{n}!\bar{v}$.

In both cases the thesis follows from the fact that, by definition of $\mathcal{F}$, $\simeq_\mu \subseteq \mathcal{R}_\emptyset$.

– $\alpha = \mathtt{n} \rhd [\bar{x}]\,\bar{w}$ with $|\,\bar{x}\,| \neq |\,\bar{w}\,|$.
Without loss of generality, we may assume that $\bar{w} = (\bar{x}, \bar{v})$. We consider the context $\mathbb{C} = [\![\cdot]\!] \mid \mathtt{n}!(\bar{v}', \bar{v})$ with $\bar{v}'$ such that for any receive activity $\mathtt{n}?\bar{w}'$ that s_1 and s_2 can immediately perform, with $\bar{w}'$ more or equal defined than $\bar{w}$, $\mathcal{M}(\bar{w}', (\bar{v}', \bar{v}))$ does not hold. Then, consider the transition $\mathbb{C}[\![s_1]\!] \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{x}|\,(\bar{v}', \bar{v})} s_1' \cdot \{\bar{x} \mapsto \bar{v}'\}$. By hypothesis, $\mathbb{C}[\![s_2]\!] \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{x}|\,(\bar{v}', \bar{v})} s$ and $s_1' \cdot \{\bar{x} \mapsto \bar{v}'\} \simeq_\mu s$. We have the following possibilities:

1. $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{y}]\,\bar{w}'} s_2'$.
   Since the substitution generated by the communication is $|\,\bar{x}\,|$ long, we have that $|\,\bar{x}\,| = |\,\bar{y}\,|$ and, by possibly exploiting α-conversion, we get that $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}''} s_2'$ with $\bar{w}''$ obtained from $\bar{w}'$ by replacing $\bar{y}$ with $\bar{x}$. From this, we have that $\bar{w}''$ is as defined as $\bar{w}$. Moreover, by rule *(com)*, we have that $\mathcal{M}(\bar{w}'', (\bar{v}', \bar{v}))$ holds. Hence, by the constraints on $\bar{v}'$, we get $\bar{w}'' = \bar{w}$. Finally, $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} s_2'$ and $s = s_2' \cdot \{\bar{x} \mapsto \bar{v}'\}$.
2. $s_2 \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{x}|\,(\bar{v}', \bar{v})} s_2'$ and $s = s_2' \mid \mathtt{n}!(\bar{v}', \bar{v})$.
   By rule *(com)* and by following the same reasoning used before, we can state that $s_2 \xrightarrow{\mathtt{n} \rhd [\bar{x}]\,\bar{w}} s_2''$. Therefore, $\mathbb{C}[\![s_2]\!] \xrightarrow{\mathtt{n}\,\emptyset\,|\bar{x}|\,(\bar{v}', \bar{v})} s_2'' \cdot \{\bar{x} \mapsto \bar{v}'\}$ and $s_1' \cdot \{\bar{x} \mapsto \bar{v}'\} \simeq_\mu s_2'' \cdot \{\bar{x} \mapsto \bar{v}'\}$.

In both cases the thesis follows from the fact that, by definition of $\mathcal{F}$, $\simeq_\mu \subseteq \mathcal{R}_\emptyset$.

□

B.2.3 COWS

Theorem B.2.7 (Theorem 4.3.7) *$\sim$ is a congruence for* COWS.

Proof. We shall prove that, given two COWS closed terms s_1 and s_2, if $s_1 \sim s_2$ then $\mathbb{C}[\![s_1]\!] \sim \mathbb{C}[\![s_2]\!]$ for every context $\mathbb{C}$. The proof is by induction on the structure of the context $\mathbb{C}$. The base case, i.e. whenever $\mathbb{C} = [\![\cdot]\!]$, is trivial. For the inductive case, we take a look at two cases:

- $\mathbb{C} = [k]\,\mathbb{D}$.
  If $\mathbb{D}$ is a closed context, then $[k]\,\mathbb{D} \equiv \mathbb{D}$ and, by induction, we immediately conclude. Instead, if $\mathbb{D}$ contains the free killer label k at most[2], by induction we may assume that $\mathbb{D}[\![s_1]\!] \sim \mathbb{D}[\![s_2]\!]$. This means that, there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ for some $\mathcal{R}_\emptyset \in \mathcal{F}$. Now, consider the following family of relations:

  $$\{\ \{\,([k]\,s, [k]\,s')\ :\ s\,\mathcal{R}_\mathcal{N}s'\,,\ s \text{ and } s' \text{satisfy } (*)\,\} \cup \mathcal{R}_\mathcal{N}\ \ :\ \mathcal{R}_\mathcal{N} \in \mathcal{F}\ \}$$

  where property $(*)$ states that if $s \xrightarrow{k} s''$ then $s' \xrightarrow{k} s'''$ and $s''\mathcal{R}_\mathcal{N}s'''$. The proof proceed by proving that the above family of relations is a bisimulation up-to $\equiv$. Indeed, by letting $s = \mathbb{D}[\![s_1]\!]$ and $s' = \mathbb{D}[\![s_2]\!]$, we obtain the thesis. In fact, since s_1 and s_2 are closed terms, $\mathbb{D}[\![s_1]\!]$ and $\mathbb{D}[\![s_2]\!]$ satisfy property $(*)$.

  Thus, let us consider a relation $\mathcal{R}'_\mathcal{N}$ belonging to the above family. If $[k]\,s \xrightarrow{\alpha} s_3$, we proceed by case analysis on α. The most interesting case is $\alpha = \dagger$, with $s \xrightarrow{k} s_4$ and $s_3 = [k]\,s_4$. By property (*), $s' \xrightarrow{k} s_5$ and $s_4\mathcal{R}_\mathcal{N}s_5$. From this, we get that $[k]\,s' \xrightarrow{\dagger} [k]\,s_5$ and, by definition of $\mathcal{R}'_\mathcal{N}$, we conclude $[k]\,s_4\mathcal{R}'_\mathcal{N}[k]\,s_5$. The remaining cases can be easily proved by exploiting the fact that $s\,\mathcal{R}_\mathcal{N}s'$.

- $\mathbb{C} = \{\!|\mathbb{D}|\!\}$.
  By induction, we may assume that $\mathbb{D}[\![s_1]\!] \sim \mathbb{D}[\![s_2]\!]$. If $\mathbb{C}$ is a closed context, then there exists a bisimulation $\mathcal{F}$ such that $\mathbb{D}[\![s_1]\!]\mathcal{R}_\emptyset\mathbb{D}[\![s_2]\!]$ for some $\mathcal{R}_\emptyset \in \mathcal{F}$. Thus, we can prove the thesis by showing that $\{\,\{\,(\{\!|s|\!\}, \{\!|s'|\!\})\ :\ s\mathcal{R}_\mathcal{N}s'\,\}\,,\ \mathcal{R}_\mathcal{N} \in \mathcal{F}\,\}$ is a bisimulation up-to $\equiv$. Instead, if $\mathbb{D}$ contains free variables, we must consider $\mathbb{D}[\![s_1]\!]$ and $\mathbb{D}[\![s_2]\!]$ under all possible substitution for such variables.

□

B.3 Proofs of results in Section 4.4

Theorem B.3.1 (Theorem 4.4.1) *Let* $\mathrm{uvar}(\alpha) = \emptyset$ *and* $\alpha \neq \mathtt{n} \lhd [n]$. $s \xrightarrow{\alpha} s'$ *if and only if, for any favourable condition* Φ, $\Phi \vdash s \overset{\Phi',\alpha}{\rightarrowtail} \Phi' \vdash s'$ *for some favourable condition* Φ'.

Proof. The proof of the "if" part proceeds by induction on the length of the inference of $s \xrightarrow{\alpha} s'$. For the base case, we reason by case analysis on the axioms of the original operational semantics.

(kill) In this case, $\alpha = k$, $s = \mathbf{kill}(k)$ and $s' = \mathbf{0}$. By rule *(s-kill)*, $\mathbf{kill}(k) \xrightarrow{true\,,k} \mathbf{0}$. Then, by rule *(constServ)*, we get that $\Phi \vdash \mathbf{kill}(k) \overset{\Phi',k}{\rightarrowtail} \Phi' \vdash \mathbf{0}$, where $\Phi' =$

[2]The case where $\mathbb{D}$ contains more than one free killer label can be dealt with in a similar way, while the case where $\mathbb{D}$ contains free variables can be dealt with as in Theoerm B.2.1.

$\mathcal{B}(\Phi \wedge \mathit{true}, \mathbf{0}, \emptyset)$ (since $\mathrm{uvar}(\Phi) = \emptyset$). By definition, $\mathcal{B}(\Phi \wedge \mathit{true}, \mathbf{0}, \emptyset) = \mathcal{B}(\Phi, \mathbf{0}, \emptyset) \wedge \mathcal{B}(\mathit{true}, \mathbf{0}, \emptyset)$. Since Φ is favourable, by Lemma 4.4.1, we have that $\mathcal{E}(\mathcal{B}(\Phi, \mathbf{0}, \emptyset)) \neq \mathit{false}$. Since $\mathcal{B}(\mathit{true}, \mathbf{0}, \emptyset) = \mathit{true} \neq \mathit{false}$, we can conclude that $\mathcal{E}(\Phi') \neq \mathit{false}$.

(rec) In this case, $\alpha = \mathtt{n} \rhd w$ and $s = \mathtt{n}?w.s'$. By rule *(s-rec)*, $\mathtt{n}?w.s' \xrightarrow{\mathit{true}, \mathtt{n} \rhd w} s'$. Then, by rule *(constServ)*, we get that $\Phi \vdash \mathtt{n}?w.s' \overset{\Phi', \mathtt{n} \rhd w}{\succ\!\!\longrightarrow} \Phi' \vdash s'$, where $\Phi' = \mathcal{B}(\Phi \wedge \mathit{true}, \mathbf{0}, \emptyset)$. As before, we can conclude that $\mathcal{E}(\Phi') \neq \mathit{false}$.

(inv) In this case, $\alpha = \mathtt{n} \lhd v$, $s = \mathtt{n}!\epsilon$ where $[\![\epsilon]\!] = v$, and $s' = \mathbf{0}$. By rule *(s-inv)*, $\mathtt{n}!\epsilon \xrightarrow{\mathit{true}, \mathtt{n} \lhd v} \mathbf{0}$. Then, by rule *(constServ$_{inv}$)*, we get that $\Phi \vdash \mathtt{n}!\epsilon \overset{\Phi', \mathtt{n} \lhd v}{\succ\!\!\longrightarrow} \Phi' \vdash \mathbf{0}$, where $\Phi' = \mathcal{B}(\Phi \wedge \mathit{true}, \mathbf{0}, \emptyset)$. As before, we can conclude that $\mathcal{E}(\Phi') \neq \mathit{false}$.

For the inductive step, we reason by case analysis on the last applied inference rule of the original operational semantics.

(choice) In this case, $s = g + g'$. By the premise of the rule *(choice)*, $g \xrightarrow{\alpha} s'$. By induction, $\Phi \vdash g \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$ for any favourable Φ and some favourable Φ'. By the premise of the rule *(constServ)*, we get that $g \xrightarrow{\Phi'', \alpha} s'$ where Φ'' is such that $\Phi' = \mathcal{B}(\Phi \wedge \Phi'', s', \emptyset)$. By rule *(s-choice)*, $g + g' \xrightarrow{\Phi'', \alpha} s'$. Finally, by rule *(constServ)*, we can conclude $\Phi \vdash g + g' \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$.

(del$_{com\,2}$) In this case, $s = [x]\, s_1$ and $s' = s_2 \cdot \{x \mapsto v\}$. By the premise of the rule *(del$_{com\,2}$)*, $s_1 \xrightarrow{\mathtt{n}\{x \mapsto v\}\, 1\, v} s_2$. By induction, we get that $\Phi \vdash s_1 \overset{\Phi', \mathtt{n}\{x \mapsto v\}\, 1\, v}{\succ\!\!\longrightarrow} \Phi' \vdash s_2$ for any favourable Φ and some favourable Φ'. By the premise of rule *(constServ)*, we get that $s_1 \xrightarrow{\Phi'', \mathtt{n}\{x \mapsto v\}\, 1\, v} s_2$ and, by rule *(del$_{com\,2}$)*, $[x]\, s_1 \xrightarrow{\Phi'', \mathtt{n}\emptyset\, 1\, v} s_2 \cdot \{x \mapsto v\}$. Finally, by rule *(constServ)*, we can conclude.

(del$_{kill\,1}$), (del$_{kill\,2}$), (del$_{kill\,3}$), (del$_2$), (str), (prot), (par$_{kill}$), (par$_3$), (par$_{com\,2}$) These cases are similar to the previous one; the latter case relies on the fact that $\mathrm{noConf}(s_2, \mathtt{n}, v, 1) = \mathit{true}$ implies that $\mathrm{confRec}(s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ such that $v \neq v_i$ for all $i \in I$.

(com$_2$) First, we consider the case $\alpha = \mathtt{n}\emptyset\, 0\, v$. By the premises of rule *(com$_2$)*, $s = (s_1 \mid s_2)$, $s' = (s_1' \mid s_2')$, $s_1 \xrightarrow{\mathtt{n} \rhd v} s_1'$ and $s_2 \xrightarrow{\mathtt{n} \lhd v} s_2'$. By induction, we get that $\Phi_1 \vdash s_1 \overset{\Phi_1', \mathtt{n} \rhd v}{\succ\!\!\longrightarrow} \Phi_1' \vdash s_1'$ and $\Phi_2 \vdash s_2 \overset{\Phi_2', \mathtt{n} \lhd v}{\succ\!\!\longrightarrow} \Phi_2' \vdash s_2'$, for any favourable conditions Φ_1 and Φ_2, and some favourable Φ_1' and Φ_2'. By the premises of rules *(constServ)* and *(constServ$_{inv}$)*, we get that $s_1 \xrightarrow{\Phi_1'', \mathtt{n} \rhd v} s_1'$ and $s_2 \xrightarrow{\Phi_2'', \mathtt{n} \lhd v} s_2'$, where conditions Φ_1'' and Φ_2'' are such that $\Phi_1' = \mathcal{B}(\Phi_1 \wedge \Phi_1'', s_1', \emptyset)$ and $\Phi_2' = \mathcal{B}(\Phi_2 \wedge \Phi_2'', s_2', \emptyset)$. By rule *(s-com)*, $s_1 \mid s_2 \xrightarrow{\Phi', \mathtt{n}\emptyset\, 0\, v} s_1' \mid s_2'$, where $\Phi' = \Phi_1'' \wedge \Phi_2'' \wedge \mathtt{n} = \mathtt{n} \wedge v = v$. Finally, by rule *(constServ)*, we can conclude that $\Phi \vdash s_1 \mid s_2 \overset{\Phi'', \mathtt{n}\emptyset\, 0\, v}{\succ\!\!\longrightarrow} \Phi'' \vdash s_1' \mid s_2'$, where

$\Phi'' = \mathcal{B}(\Phi \wedge \Phi', s_1' \mid s_2', \emptyset)$. The case $\alpha = \mathtt{n}\,\sigma\,1\,v$ proceeds as above, by also relying on the fact that $\text{noConf}(s_1 \mid s_2, \mathtt{n}, v, 1) = \mathit{true}$ implies that $\text{confRec}(s_1 \mid s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ with $v \neq v_i$ for all $i \in I$.

Consider now the "only if" part of the theorem. By the premises of rules *(constServ)* and *(constServ$_{inv}$)*, we get that $s \xrightarrow{\Phi'', \alpha} s'$ where $\Phi' = \mathcal{B}(\Phi \wedge \Phi'', s', \emptyset)$. By hypothesis $\mathcal{E}(\Phi') \neq \mathit{false}$, hence $\mathcal{E}(\Phi'') \neq \mathit{false}$ too. The proof proceeds by induction on the length of the inference of $s \xrightarrow{\Phi'', \alpha} s'$. We omit the details because the proof proceeds as that of the "if" part, but the steps are executed in the reverse order. For the base case, we reason by case analysis on the axioms of the symbolic operational semantics. We take a look at one base case:

(s-rec) In this case, $\Phi'' = \mathit{true}$, $\alpha = \mathtt{n} \rhd w$ and $s = \mathtt{n}?w.s'$. Trivially, by rule *(rec)*, $\mathtt{n}?w.s' \xrightarrow{\mathtt{n} \rhd w} s'$.

For the inductive step, we reason by case analysis on the last applied inference rule of the symbolic operational semantics. We take a look at two cases:

(s-choice) In this case, $s = g + g'$. By the premise of the rule *(s-choice)*, $g \xrightarrow{\Phi'', \alpha} s'$. By induction, we get that $g \xrightarrow{\alpha} s'$. Finally, by rule *(choice)*, we can conclude $g + g' \xrightarrow{\alpha} s'$.

(s-com) In this case, $s = (s_1 \mid s_2)$, $\Phi'' = (\Phi_1 \wedge \Phi_2 \wedge \mathtt{n} = \mathtt{n} \wedge v \neq \text{confRec}(s_1 \mid s_2, \mathtt{n}))$, $\alpha = \mathtt{n}\{x \mapsto v\}\,1\,v$ and $s' = (s_1' \mid s_2')$. Since $\mathcal{E}(\Phi'') \neq \mathit{false}$, we get that $\mathcal{E}(\Phi_1) \neq \mathit{false}$, $\mathcal{E}(\Phi_2) \neq \mathit{false}$ and $\text{confRec}(s_1 \mid s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ such that $v \neq v_i$ for all $i \in I$. This means that $\text{noConf}(s_1 \mid s_2, \mathtt{n}, v, 1)$ holds true. By induction and since $\mathcal{E}(\Phi_1) \neq \mathit{false}$ and $\mathcal{E}(\Phi_2) \neq \mathit{false}$, we have that $s_1 \xrightarrow{\mathtt{n} \rhd x} s_1'$ and $s_2 \xrightarrow{\mathtt{n} \lhd v} s_2'$. Thus, by rule *(com$_2$)*, we can conclude that $s_1 \mid s_2 \xrightarrow{\mathtt{n}\{x \mapsto v\}\,1\,v} s_1' \mid s_2'$. □

B.4 Proofs of results in Section 5.1.1

Theorem B.4.1 (Completeness, Theorem 5.1.1) *Given an Orc expression f and a communication endpoint $\mathtt{r}$, $f \overset{l}{\hookrightarrow} f'$ implies $[\![f]\!]_{\mathtt{r}} \equiv \langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s \overset{\alpha}{\Longrightarrow} \langle\!\langle f' \rangle\!\rangle_{\mathtt{r}} \mid s$, where $\alpha = \mathtt{r} \lhd \langle v \rangle$ if $l = !v$, and $\alpha = \mathtt{n}\,\emptyset\,\ell\,\bar{v}$ if $l = \tau$.*

Proof. The proof proceeds by induction on the length of the inference of $f \overset{l}{\hookrightarrow} f'$.
Base Step: We reason by case analysis on the axioms of the operational semantics.

(SiteCall) In this case $f = S(v)$, $l = !v'$ and $f' = 0$. By encoding definition, $\langle\!\langle S(v) \rangle\!\rangle_{\mathtt{r}} \mid s = \mathtt{S}!\langle v, \mathtt{r} \rangle \mid s$, where $s = *\,[x, y]\,\mathtt{S}?\langle x, y \rangle.y!\langle v' \rangle$. Thus, $\mathtt{S}!\langle v, \mathtt{r} \rangle \mid s \xrightarrow{\mathtt{S}\,\emptyset\,2\,\langle v, \mathtt{r} \rangle} \mathtt{r}!\langle v' \rangle \mid s \xrightarrow{\mathtt{r} \lhd \langle v' \rangle} \mathbf{0} \mid s$. Since $\langle\!\langle \mathbf{0} \rangle\!\rangle_{\mathtt{r}} = \mathbf{0}$, we can conclude.

(Def) In this case $f = E(w)$ with $E(x) \triangleq g$, $l = \tau$ and $f' = g \cdot \{x \mapsto w\}$. By encoding definition, $\langle\!\langle E(w) \rangle\!\rangle_{\mathtt{r}} \mid s = [\mathtt{r}'](\mathtt{E}!\langle \mathtt{r}, \mathtt{r}' \rangle \mid [z']\,\mathtt{r}'?\langle z' \rangle.z'!\langle w \rangle) \mid s$, where $s = *\,[y, z]\,\mathtt{E}?\langle y, z \rangle.[\mathtt{r}''](z!\langle \mathtt{r}'' \rangle \mid [x](\mathtt{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_y))$. Thus, $\langle\!\langle E(w) \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{E}\,\emptyset\,2\,\langle \mathtt{r}, \mathtt{r}' \rangle} [\mathtt{r}'](*\,[z']\,\mathtt{r}'?\langle z' \rangle.z'!\langle w \rangle \mid [\mathtt{r}''](\mathtt{r}'!\langle \mathtt{r}'' \rangle \mid [x](\mathtt{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}}))) \mid s \triangleq s'$. Then, $s' \xrightarrow{\mathtt{r}'\,\emptyset\,1\,\langle \mathtt{r}'' \rangle} [\mathtt{r}''](\mathtt{r}''!\langle w \rangle \mid [x](\mathtt{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}})) \mid s \triangleq s''$. In case $w = z''$, since $\langle\!\langle g \cdot \{x \mapsto z''\} \rangle\!\rangle_{\mathtt{r}} \equiv [\mathtt{r}''](\mathtt{r}''!\langle z'' \rangle \mid [x](\mathtt{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}}))$, we can directly conclude. Instead, in case $w = v'$, we have $s'' \xrightarrow{\mathtt{r}''\,\emptyset\,1\,\langle v' \rangle} \langle\!\langle g \rangle\!\rangle_{\mathtt{r}} \cdot \{x \mapsto v'\} \mid s$. Thus, since $\langle\!\langle g \rangle\!\rangle_{\mathtt{r}} \cdot \{x \mapsto v'\} \equiv \langle\!\langle g \cdot \{x \mapsto v'\} \rangle\!\rangle_{\mathtt{r}}$, we can conclude.

Inductive Step: We reason by case analysis on the last applied inference rule of the operational semantics.

(Sym1) In this case, $f = f_1 \mid f_2$, $f' = f_1' \mid f_2$. By the premise of the rule *(Sym1)*, $f_1 \overset{l}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s = \langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid \langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}} \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid s' \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}} \mid s'$ where $s \equiv s' \mid s''$. By rules *(par$_3$)* or *(par$_{com}$)*, we can conclude.

(Sym2) Similar to the previous case.

(Seq1) In this case, $f = f_1 > x > f_2$, $l = \tau$ and $f' = f_1' > x > f_2$. By the premise of the rule *(Seq1)*, $f_1 \overset{\tau}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s = [\mathtt{r}'](\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid *\,[x]\,\mathtt{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}) \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid s' \overset{\mathtt{n}\,\emptyset\,\ell\,\bar{v}}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid s'$ where $s \equiv s' \mid s''$. By rules *(par$_{com}$)* and *(del$_2$)*, we can conclude that $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s \overset{\mathtt{n}\,\emptyset\,\ell\,\bar{v}}{\Longrightarrow} \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\mathtt{r}} \mid s$.

(Seq2) In this case, $f = f_1 > x > f_2$, $l = !v$ and $f' = (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\}$. By the premise of the rule *(Seq2)*, $f_1 \overset{!v}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s = [\mathtt{r}'](\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid *\,[x]\,\mathtt{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}) \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid s' \overset{\mathtt{r}' \lhd \langle v \rangle}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid s'$ where $s \equiv s' \mid s''$. Thus, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s \overset{\mathtt{r}'\,\emptyset\,1\,\langle v \rangle}{\Longrightarrow} [\mathtt{r}'](\langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid *\,[x]\,\mathtt{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}} \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}}) \mid s \equiv \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\mathtt{r}} \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid s$.

(Asym2) In this case, $f = f_1$ **where** $x :\in f_2$, $l = \tau$ and $f' = f_1$ **where** $x :\in f_2'$. By the premise of the rule *(Asym2)*, $f_2 \overset{\tau}{\hookrightarrow} f_2'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s = [\mathtt{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid [k](\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x \rangle.\mathbf{kill}(k))) \mid s$. By induction, $\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid s' \overset{\mathtt{n}\,\emptyset\,\ell\,\bar{v}}{\Longrightarrow} \langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid s'$ where $s \equiv s' \mid s''$. Thus, by rule *(del$_2$)*, we can conclude that $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s \overset{\mathtt{n}\,\emptyset\,\ell\,\bar{v}}{\Longrightarrow} [\mathtt{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid [k](\langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x \rangle.\mathbf{kill}(k))) \mid s \equiv \langle\!\langle f_1 \textbf{ where } x :\in f_2' \rangle\!\rangle_{\mathtt{r}} \mid s$.

(Asym1) Similar to the previous case.

(Asym3) In this case, $f = f_1$ **where** $x :\in f_2$, $l = \tau$ and $f' = f_1 \cdot \{x \mapsto v\}$. By the premise of the rule *(Asym3)*, $f_2 \overset{!v}{\hookrightarrow} f_2'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s = [\mathtt{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid$

$[k]\,(\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x\rangle.\mathbf{kill}(k))) \mid s$. By induction, $\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid s' \xrightarrow{\mathtt{r}' \lhd \langle v\rangle} \langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid s'$ where $s \equiv s' \mid s''$. Then, by encoding definition and rule *(del$_2$)*, $s''' \triangleq [\mathtt{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x\rangle.\mathbf{kill}(k)) \mid \mathtt{r}'!\langle v\rangle) \mid s$ is a reduct of $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s$. We can conclude that $s''' \xrightarrow{\mathtt{r}'\,\emptyset\,1\,\langle v\rangle} [\mathtt{r}']\,(\langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid \mathbf{kill}(k))) \mid s \xrightarrow{\dagger} [\mathtt{r}']\,(\langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid [k]\,\mathbf{0}) \mid s \equiv \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid s$.

□

Theorem B.4.2 (Soundness, Theorem 5.1.2) *Given an Orc expression f and a communication endpoint $\mathtt{r}$, $[\![f]\!]_{\mathtt{r}} \equiv \langle\!\langle f \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s'$ implies that there exists an Orc expression f' such that $f \overset{l}{\hookrightarrow} f'$ and $s' \overset{\alpha}{\Longrightarrow} \langle\!\langle f' \rangle\!\rangle_{\mathtt{r}} \mid s$, where $\alpha = \mathtt{r} \lhd \langle v\rangle$ if $l = !v$, and $\alpha = \mathtt{n}'\,\emptyset\,\ell'\,\bar{v}')$ if $l = \tau$.*

Proof. The proof proceeds by induction on the definition of the encoding $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}}$.
Base Step: We reason by case analysis on the non-inductive cases of the definition.

($f = \mathbf{0}$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = \mathbf{0}$ and $s = \mathbf{0}$. Since $[\![f]\!]_{\mathtt{r}} \equiv \mathbf{0} \xrightarrow{\alpha}\!\!\!\!\rightarrow$, we can trivially conculde.

($f = S(w)$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = \mathtt{S}!\langle w, \mathtt{r}\rangle$ and $s = *\,[x, y]\,\mathtt{S}?\langle x, y\rangle.y!\langle v'\rangle$. If $w = z$ then $[\![f]\!]_{\mathtt{r}} \xrightarrow{\alpha}\!\!\!\!\rightarrow$, thus we can trivially conculde. If $w = v$ then $[\![f]\!]_{\mathtt{r}} \equiv \mathtt{S}!\langle v, \mathtt{r}\rangle \mid s \xrightarrow{\mathtt{S}\,\emptyset\,2\,\langle v,\mathtt{r}\rangle} \mathtt{r}!\langle v'\rangle \mid s = s'$. By rule *(SiteCall)* we have $l = !v'$ and $f' = \mathbf{0}$. Thus, $s' \xrightarrow{\mathtt{r} \lhd \langle v'\rangle} \mathbf{0} \mid s$. Since $\langle\!\langle \mathbf{0} \rangle\!\rangle_{\mathtt{r}} = \mathbf{0}$, we can conclude.

($f = E(w)$) Assuming $E(x) \triangleq g$, we have $\langle\!\langle E(w) \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}']\,(\mathtt{E}!\langle \mathtt{r}, \mathtt{r}'\rangle \mid [z]\,\mathtt{r}'?\langle z\rangle.z!\langle w\rangle)$ and $s = *\,[y', z']\,\mathtt{E}?\langle y', z'\rangle.[\mathtt{r}'']\,(z'!\langle \mathtt{r}''\rangle \mid [x]\,(\mathtt{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{y'}))$. Then, $[\![f]\!]_{\mathtt{r}} \xrightarrow{\mathtt{E}\,\emptyset\,2\,\langle \mathtt{r}, \mathtt{r}'\rangle} [\mathtt{r}']\,([z]\,\mathtt{r}'?\langle z\rangle.z!\langle w\rangle) \mid [\mathtt{r}'']\,(\mathtt{r}'!\mathtt{r}'' \mid [x]\,(\mathtt{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}})) \mid s = s'$. By rule *(Def)* we have $l = \tau$ and $f' = g \cdot \{x \mapsto w\}$. Moreover, $s' \xrightarrow{\mathtt{r}'\,\emptyset\,1\,\langle \mathtt{r}''\rangle} [\mathtt{r}'']\,(\mathtt{r}''!\langle w\rangle \mid [x]\,(\mathtt{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}})) \mid s \triangleq s''$. If $w = y$, we directly conclude, since $s'' = \langle\!\langle g \cdot \{x \mapsto y\} \rangle\!\rangle_{\mathtt{r}}$. If $w = v$, we can conclude by $s'' \xrightarrow{\mathtt{r}''\,\emptyset\,1\,\langle v\rangle} \langle\!\langle g \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}}$.

Inductive Step: We reason by case analysis on the inductive cases of the definition.

($f = f_1 \mid f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = \langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid \langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}$. By encoding definition, the transition $\xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}}$ cannot be produced by a synchronization between $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}}$ and $\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}$. Then, we have only the following cases:

- $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s_1$.
  By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}} \mid s$. By rule *(Sym1)*, $f \overset{l}{\hookrightarrow} f_1' \mid f_2 = f'$. By rule *(par$_3$)* or *(par$_{com}$)*, we can conclude that $s' \equiv s_1 \mid \langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}} \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}} \mid s \mid \langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}} \equiv \langle\!\langle f' \rangle\!\rangle_{\mathtt{r}} \mid s$.
- $\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\,\emptyset\,\ell\,\bar{v}} s_2$.
  Similar to the previous case.

($f = f_1 > x > f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}'](\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid *[x]\,\mathtt{r}'?\langle x\rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}})$. Lemma 5.1.1 implies that the inference of the transition $\xrightarrow{\mathtt{n}\emptyset\ell\bar{v}}$ derives from $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}'} \mid s \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_1$. By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid s$. We have two cases:

- $l = \tau$.
  By rule *(Seq1)*, $f \overset{\tau}{\hookrightarrow} f_1' > x > f_2$. By rules *($par_{com}$)* and *($del_2$)*, $s' \overset{\mathtt{n}'\emptyset\ell'\bar{v}'}{\Longrightarrow} [\mathtt{r}'](\langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid *[x]\,\mathtt{r}'?\langle x\rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}) \mid s \equiv \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\mathtt{r}} \mid s$.
- $l = !v$.
  By rule *(Seq2)*, $f \overset{!v}{\hookrightarrow} (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\}$. By rules *($com_2$)* and *($del_2$)*, $s' \overset{\mathtt{r}'\emptyset 1\langle v\rangle}{\Longrightarrow} [\mathtt{r}'](\langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid *[x]\,\mathtt{r}'?\langle x\rangle.\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}}) \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid s \equiv \langle\!\langle (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid s$.

($f = f_1$ **where** $x :\in f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}', x]\,(\,\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\,\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x\rangle.\mathbf{kill}(k)\,)\,)$. By Lemma 5.1.1, we have two cases:

- $\langle\!\langle f_1 \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_1$.
  By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}'} \mid s$. By rule *(Asym1)*, $f \overset{l}{\hookrightarrow} f_1'$ **where** $x :\in f_2$. By rules *(del_2)* and *(par_{com})*, $s' \overset{\alpha}{\Longrightarrow} [\mathtt{r}', x]\,(\,\langle\!\langle f_1' \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\,\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid \mathtt{r}'?\langle x\rangle.\mathbf{kill}(k)\,)\,) \mid s \equiv \langle\!\langle f_1' \textbf{ where } x :\in f_2 \rangle\!\rangle_{\mathtt{r}}$.
- $\langle\!\langle f_2 \rangle\!\rangle_{\mathtt{r}'} \mid s \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s_2$.
  By induction, $f_2 \overset{l}{\hookrightarrow} f_2'$ and $s_2 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_2' \rangle\!\rangle_{\mathtt{r}'} \mid s$. We have two cases:
  - $\alpha = \mathtt{n}'\emptyset\ell'\bar{v}'$.
    Similar to the previous case.
  - $\alpha = \mathtt{r}'!\langle v\rangle$.
    By rule *(Asym3)*, $f \overset{\tau}{\hookrightarrow} f_1 \cdot \{x \mapsto v\}$. By rules *($com_2$)*, *($par_{com}$)* and *($del_2$)*, $s' \overset{\mathtt{r}'\emptyset 1\langle v\rangle}{\Longrightarrow} \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}} \mid [k]\,(\,\langle\!\langle f_2' \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}'} \mid \mathbf{kill}(k)\,) = s''$. Then, by rules *(kill)*, *(par_{kill})*, *($del_{kill\,1}$)* and *(par_3)*, $s'' \xrightarrow{\dagger} \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\mathtt{r}}$.

($f = g \cdot \{x \mapsto y\}$) In this case $\langle\!\langle f \rangle\!\rangle_{\mathtt{r}} = [\mathtt{r}']\,(\mathtt{r}'!\langle y\rangle \mid [x]\,(\mathtt{r}'?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{\mathtt{r}}))$. Since $\mathtt{r}'!\langle y\rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow$, we have $\langle\!\langle g \rangle\!\rangle_{\mathtt{r}} \mid s \xrightarrow{\mathtt{n}\emptyset\ell\bar{v}} s''$. By induction, $g \overset{l}{\hookrightarrow} g'$ and $s'' \overset{\alpha}{\Longrightarrow} \langle\!\langle g' \rangle\!\rangle_{\mathtt{r}} \mid s$. By rules *($par_{com}$)* and *($del_2$)*, we can conclude that $s' \overset{\alpha}{\Longrightarrow} [\mathtt{r}']\,(\mathtt{r}'!\langle y\rangle \mid [x]\,(\mathtt{r}'?\langle x\rangle \mid \langle\!\langle g' \rangle\!\rangle_{\mathtt{r}})) \mid s \equiv \langle\!\langle g' \cdot \{x \mapsto y\} \rangle\!\rangle_{\mathtt{r}} \mid s$.

□

Appendix C

Encoding SRML into COWS

We report here the 'machine readable' syntax of SRML, the specification of the automotive case study presented in Section 2.4.1 using this textual notation, and its encoding in COWS.

C.1 SRML syntax

A Backus-Naur Form syntax of SRML is presented in Table C.1. The set of names is ranged over by *mod*, *pr*, *bp*, *lp*, *comp*, *br*, *req*, *wire*, *int*, *tr*, *param*, *type* and *lvar* used for a module, provides/serves-interface, business protocol, layer protocol, component, business role, requires-interface, wire, interaction, transition, parameter, type and local variable. The names of nodes in a SRML module, when we refer in general to either a provides-interface, a requires-interface or a component, are ranged over by *name*.

The language of *expressions*, ranged over by *e*, is deliberately omitted; we assume that expressions contain, at least, names and invocation of **ask** interactions, whose names differ from those of the expression functions (i.e. an expression cannot contain an invocation of the interaction **ask** *sqrt*(*integer*) : *integer* since *sqrt* is an expression function). A particular kind of expressions are the *conditions*, ranged over by *c*, whose evaluation is a boolean value. We will use $lvar'$ to denote the value that a state variable *lvar* has after the corresponding transition. The language is also parameterized by an unspecified set of *Service Level Agreement constraints* (ranged over by *SLAc*), and by an unspecified set of service descriptions (ranged over by *ServiceDesc*) that represent the behavioural specifications of abstract references (i.e., requires-interfaces in SRML).

The syntax of the module definition is given in the upper part of Table C.1. A module M is defined by a number of components COMPS, one provides-interface/serves-interface, a number of requires-interfaces REQS, one external policy SLAc, a number of wires and specifications SPECS. COMPS represents a set of one or more components, each defined by a name *comp* and a type *br* that refers to one BR element in the specifications SPECS, and equipped with a set of initial assignments and a termination condition.

An external provides-interface is defined by a name *pr* and a type *bp* that refers to one BP element in SPECS. A serves-interface is defined by a name *pr* and a type *lp* that refers to one LP element in SPECS. REQS represents a set of one or more requires-interfaces, each defined by a name *req* and a type *bp* that refers to one BP element in the specifications SPECS. Each external interface is equipped with a trigger condition that launches the discovery. SPECS is the set of specifications, which can be business roles (BR), business protocol BP and layer protocol (LP) elements.

The syntax of the specifications referred to by a module definition is given in the lower part of Table C.1. The business role BR is defined by one declaration of interactions INTS and one orchestration description. INTS represents the interactions supported by SRML. There are different types of interaction: asynchronous one-way **`rcv`** and **`snd`**, asynchronous conversational **`r&s`** and **`s&r`**, and synchronous **`ask`**, **`rpl`**, **`tll`**, and **`prf`**. The orchestration consists of an optional declaration of local variables LVARS and one or more transitions TRANS. A transition has (1) an optional trigger TRIGS that is either a condition, a receive event or a receive of a synchronous interaction (when a trigger is not specified we consider the default condition to be **true**), (2) an optional guard that is a condition (where **true** is the default condition),(3) optional effects (i.e., a number of assignments GASGS), and (4) an optional sends section represented by the term GSENDS consisting in one or more send interaction events, sends of synchronous interactions, return events for **`rpl`** and **`prf`** interactions (denoted by *int* ↰ [*e*]), and assignments to output parameters. The interaction events and assignments in GSENDS may have a condition, likewise assignments in GASGS.

C.2 SRML specification of the automotive case study

Table C.2 presents an excerpt of the specification of the module *OnRoadRepair* illustrated in Figure 5.1. *OnRoadRepair* is defined by a number of component-/serves-/requires-interfaces and their associated type (e.g., *OR* of type *Orchestrator*). The types are defined below (see **`SPECIFICATIONS`**).

The internal policies **`init`** and **`term`** of *OR* define the initialisation and termination conditions of the component. Initially, the local variable *s* has value *INIT*. The component is compulsorily terminated when either the final state is reached (i.e. $s = FINAL$) or a fatal error occurs (i.e. $s = ERR$). According to the internal policy **`trigger`** of *GA* the discovery process is triggered by the condition $s = READY$.

The wires *SO* and *OG* connect pairs of nodes by defining a relationship between the interactions and the parameters of the corresponding specifications. For simplicity, we consider only the case of wires that define a straight one-to-one mapping.

Each specification is composed by a syntactical interface (**`INTERACTIONS`**). In SRML interactions are asynchronous and can be one-way (i.e., receive **`rcv`** or send **`snd`**) or conversational (i.e., receive-and-send **`r&s`**, or send-and-receive **`s&r`**). A number of interaction events is associated with each conversational interaction: an initiation event (denoted by

(*Module*)

M ::= **MODULE** *mod* **is**
COMPONENTS COMPS
PRVORSRV
[**REQUIRES** REQS]
[**EXTERNAL POLICY** *SLAc*]
WIRES WIRES
SPECIFICATIONS SPECS

(*Provides-Interface or Serves Interface*)

PRVORSRV ::= **PROVIDES** *pr* : *bp*
| **SERVES** *pr* : *lp*

(*Components*)

COMPS ::= COMPS COMPS
| *comp* : *br*
[**init** IASGS]
[**term** *c*]

(*Initial assignments*)

IASGS ::= IASGS ∧ IASGS
| *lvar* = *e*

(*Requires interfaces*)

REQS ::= REQS REQS
| *req* : *bp* [**trigger** *c*]

(*Event type*)

ET ::= 🔔 | ✉ | ✓ | ✘ | ✞

(*Wires*)

WIRES ::= WIRES WIRES
| *wire* : *name* *name*
WLINES

(*Wire lines*)

WLINES ::= WLINES WLINES
| *int* ↔ *int*
[: ET *param* ↔ *param*, . . .
. . . , ET *param* ↔ *param*]

(*Specifications*)

SPECS ::= SPECS SPECS
| BR | BP

(*Business role*)

BR ::= **BUSINESS ROLE** *br* **is**
INTERACTIONS INTS
ORCHESTRATION
[**local** LVARS]
TRANS

(*Business Protocol*)

BP ::= **BUSINESS PROTOCOL** *bp* **is**
INTERACTIONS INTS
BEHAVIOUR ServiceDesc

(*Layer Protocol*)

LP ::= **LAYER PROTOCOL** *lp* **is**
INTERACTIONS INTS
BEHAVIOUR Description

(*Interactions*)

INTS ::= INTS INTS
| **rcv** *int* [🔔 PARAMS]
| **snd** *int* [🔔 PARAMS]
| **r&s** *int* [🔔 PARAMS
| [✉ PARAMS]]
| **s&r** *int* [🔔 PARAMS
| [✉ PARAMS]]
| **ask** *int*(TYPES) : *type*
| **rpl** *int*(TYPES) : *type*
| **tll** *int*(TYPES)
| **prf** *int*(TYPES)

(*Parameters*)

PARAMS ::= PARAMS , PARAMS
| *param* : *type*

(*Types*)

TYPES ::= TYPES , TYPES
| *type*

(*Local variables*)

LVARS ::= LVARS , LVARS
| *lvar* : *type*

(*Transitions*)

TRANS ::= TRANS TRANS
| **transition** *tr*
[**triggeredBy** TRIGS]
[**guardedBy** *c*]
[**effects** GASGS]
[**sends** GSENDS]

(*Trigger*)

TRIGS ::= *c* | *int* ET
| *int*($param_1, \ldots, param_n$)

(*Guarded assignments*)

GASGS ::= GASGS ∧ GASGS
| [*c* ⊃] ASG

(*Assignment*)

ASG ::= *lvar*[′] = *e*
| *int*.*param* = *e*

(*Guarded sends*)

GSENDS ::= GSENDS ∧ GSENDS
| [*c* ⊃] *int* [ET]
| [*c* ⊃] *int*($e_1, \ldots e_n$)
| [*c* ⊃] *int* ↰ [*e*]
| [*c* ⊃] ASG

Table C.1: SRML syntax

```
MODULE OnRoadRepair is
  COMPONENTS OR : Orchestrator init s = INIT term s = FINAL ∨ s = ERR
  SERVES SM : SensorMonitor
  REQUIRES GA : Garage trigger s = READY
           ...
  EXTERNAL POLICY carUserSLAconstraints
  WIRES SO : SM OR activation ↔ init : 🔔 sensorData ↔ data
        OG : OR GA bookGarage ↔ acceptBooking : 🔔 data ↔ info,
                                                ✉ price ↔ servicePrice
        ...
  SPECIFICATIONS
    BUSINESS ROLE Orchestrator is
      INTERACTIONS
          rcv init   🔔 data : carData
          s&r bookGarage   🔔 data : carData
                           ✉ price : moneyVal
          ...
      ORCHESTRATION
        local s : [INIT, READY, WAITING, GA_PRICE, ..., FINAL, ERR],
              data : carData, much : moneyVal, ...
        transition data_receiving
          triggeredBy init🔔
          guardedBy s = INIT
          effects s′ = READY ∧ data′ = init.data

        transition reqToGarage
          guardedBy s = READY
          effects s′ = WAITING
          sends bookGarage.data = data ∧ bookGarage🔔

        transition respFromGarage
          triggeredBy bookGarage✉
          guardedBy s = WAITING
          effects s′ = GA_PRICE ∧ much′ = bookGarage.price
        ...
    LAYER PROTOCOL SensorMonitor is
      INTERACTIONS snd activation   🔔 sensorData : carData
      BEHAVIOUR SensorMonitorBehaviour

    BUSINESS PROTOCOL Garage is
      INTERACTIONS r&s acceptBooking   🔔 info : carData
                                       ✉ servicePrice : moneyVal
      BEHAVIOUR GarageBehaviour
```

Table C.2: The textual definition of the module *OnRoadRepair*

🔔), a reply-event (denoted by ✉), and so on. Interactions can involve a number of parameters for each phase of the conversation (e.g., 🔔-parameters for the initiation, ✉-parameters for the reply, etc.). One-way interactions have associated only one 🔔-event and one 🔔-parameter.

Every instance of *Orchestrator* can engage in the interactions *init* and *bookGarage*. The former is of type **rcv** and permits to receive data from the sensor monitor installed in the car. The data are represented by the parameter *data* of type *carData*. The interaction *bookGarage* is used for engaging with a garage service. This interaction is conversational (of type **s&r**) and has one 🔔-parameter *data* and one ✉-parameter *price* through which the price for repairing the car can be obtained. In the initial state, i.e. when $s = INIT$,

```
MODULE RepairService is
  COMPONENTS GO : GarageOrchestrator init s = INIT term s = FINAL
  PROVIDES CR : Customer
  REQUIRES ...
  EXTERNAL POLICY garageSLAconstraints
  WIRES CG : CR GO  getRequest ↔ handleRequest : 🔔 dataFromCar ↔ d,
                                                 ✉ cost ↔ c
        ...
  SPECIFICATIONS
    BUSINESS ROLE GarageOrchestrator is
      INTERACTIONS
          r&s handleRequest  🔔 d : carData
                             ✉ c : moneyVal
          ...
      ORCHESTRATION
        local s : [INIT, HANDLING, ..., FINAL], data : carData
        transition reqResp
          triggeredBy handleRequest🔔
          guardedBy s = INIT
          effects s' = HANDLING ∧ data' = handleRequest.d
          sends handleRequest.c = computePrice(data') ∧ handleRequest✉
        ...
    BUSINESS PROTOCOL Customer is
      INTERACTIONS s&r getRequest  🔔 dataFromCar : carData
                                   ✉ cost : moneyVal
      BEHAVIOUR CustomerBehaviour
```

Table C.3: The textual definition of the module *RepairService*

an *Orchestrator* can perform only the transition *data_receiving*, which is triggered by the event *init*🔔 and changes the internal state (as usual, we denote by s' and $data'$ the next value of the local state variables s and *data*). The transition *reqToGarage* has no trigger and is executed as soon as the guard $s = READY$ is true. The transition sends the event *bookGarage*🔔 and assigns the sensor data (stored in the local variable *data*) to the parameter *bookGarage.data*. The event is sent to the (dynamically discovered) garage service. Finally, by means of transition *respFromGarage*, the price required by the garage service can be received and stored in the local variable *much*.

An excerpt of the specification of the module *RepairService* is shown in Table C.3. It contains the component *GO* (of type *GarageOrchestrator*) connected to the provides-interface *CR* (of type *Customer*) by the wire *CG*. The *GarageOrchestrator* provides the interaction *handleRequest* of type **r&s**, which is made available through the provides-interface to bind to customers upon selection (e.g. *bookGarage*). The interaction *handleRequest* can be engaged by executing the transition *reqResp*. In this way, the data of the customer's car are received and processed to calculate the cost of the repair (through *computePrice*$(\cdot)$), after which the computed cost is sent back to the customer.

C.3 A flavour of the encoding

We outline here the encoding of SRML in COWS through the automotive case study introduced above. A more complete account of the encoding can be found in the following

technical report [30]. Firstly, we present the static aspects of the encoding, i.e. how a SRML configuration is implemented in COWS, and then the dynamic ones, by showing a feasible computation of the resulting COWS term.

Static aspects of the encoding. The COWS term representing all the entities involved in the automotive case study, where $\langle\!\langle \cdot \rangle\!\rangle$ represents the encoding in COWS of the enclosed term, is

$$\langle\!\langle \textbf{MODULE}\ \mathit{OnRoadRepair}\ \textbf{is}\ \ldots \rangle\!\rangle \mid \langle\!\langle \textbf{MODULE}\ \mathit{RepairService}\ \textbf{is}\ \ldots \rangle\!\rangle \mid \mathit{Middleware} \mid \mathit{Environment} \mid \mathit{BottomLayer}$$

where *Middleware* is the term (*Broker* | *Registry* | *ConstraintSolver* | *MatchmakingAgent* | ...), while *Environment* contains, at least, a COWS term representing the car's sensor monitor that interacts with the module instance through the serves-interface. The encoding of the bindings performed through uses-interfaces is in progress. Hence, the term *BottomLayer* is left unspecified.

For example, a sensor monitor can be represented by the following COWS term:

$$[id_{sm}]\,(\ \mathit{OnRoadRepair} \bullet \mathit{create}!\langle \mathit{sensorMonitor}, id_{sm} \rangle \mid \mathit{OnRoadRepair} \bullet \mathit{activation}!\langle id_{sm}, \text{🕭}, \text{“}\mathit{gps} = (4348.1143N, 1114.7206E), \mathit{fuelPr} = 60psi, \mathit{brakeBias} = 70/30, \ldots \text{”} \rangle\ )$$

This term directly invokes the service factory of the module *OnRoadRepair* without resorting to a discovery mechanism (recall that *OnRoadRepair* is an activity module). The operation *create* does not correspond to an interaction supported by the original SRML module but to the factory of the COWS encoding of *OnRoadRepair*. It has the effect of creating a new instance of the module and initialising it with the sensor monitor partner name *sensorMonitor* and the fresh instance identifier id_{sm}. In parallel, the sensor monitor sends the collected data by invoking the COWS operation corresponding to the interaction *activation* provided by the interface *SM* of *OnRoadRepair*.

A SRML module corresponds to a persistent COWS service that can be instantiated by invoking the operation *create* with the partner name of the module (that coincides with the name of the module, as e.g., *RepairService*). We assume that names of modules are distinct; this is reasonable because, at the real implementation level, module partner names can be thought of as URIs.

The encoding of *RepairService* is:

$$\mathit{Broker} \bullet \mathit{pub}!\langle \mathit{RepairService}, \text{“}\mathit{Customer}\ \textbf{is}\ \ldots\text{”}, \mathit{garageSLAconstraints} \rangle \mid *\,[x_{cust}, x_{ext\_id}]\,\mathit{RepairService} \bullet \mathit{create}?\langle x_{cust}, x_{ext\_id} \rangle.\ [id_{intra}]\,(\ \mathit{ProvidesInt} \mid \mathit{RequiresInt} \mid \mathit{Wires} \mid \mathit{Components}\ )$$

With respect to the architecture of the encoding of a service module we have seen in Section 5.1.5, we have that *Factory* corresponds to the replicated receive along the endpoint *RepairService* • *create*, while *InstanceHandler*, *Orchestration* and *DiscoveryHandler* correspond to *ProvidesInt*, *Wires* | *Components* and *RequiresInt*, respectively. The encoding

of the module *OnRoadRepair* is similar, except for the absence of the publication activity (i.e. the invoke along the endpoint $Broker \bullet pub$) and the replacement of *ProvidesInt* with the term *ServesInt* implementing the serves-interface *SM*.

To instantiate a module, a service has to provide its partner name (to allow the created instance to reply) and a conversation identifier (stored in x_{ext_id}) that will be used for correlating inter-module communication to avoid interference among instances of the same module. To guarantee absence of interference during intra-module communication when a new module instance is created, a fresh conversation identifier id_{intra} is generated. This identifier is necessary because communication among entities of an instance (i.e. components, wires and interfaces) are performed along the same endpoints used by other instances of the same module. The intra-module identifier differs from the external identifier to prevent external entities from directly contacting internal entities. Such an identifier is also used in the communication with *Broker* during the discovery phase.

The encoding of a wire is a persistent COWS service that catches a send event (by means of a receive activity) from a connected entity, adapts the communication endpoint and forwards the adapted event (by means of an invoke activity) to the other entity. For example, the wire *OG* between *OR* and *GA* in *OnRoadRepair* is:

$$
\begin{aligned}
&*\,[x_{data}]\, OG_{roleA} \bullet bookGarage?\langle id_i, \text{🔔}, x_{data}\rangle.\, GA \bullet acceptBooking!\langle id_i, \text{🔔}, x_{data}\rangle \\
&\mid *\,[x_{servicePrice}]\, OG_{roleB} \bullet acceptBooking?\langle id_i, \text{✉}, x_{servicePrice}\rangle. \\
&\qquad\qquad OR \bullet bookGarage!\langle id_i, \text{✉}, x_{servicePrice}\rangle
\end{aligned}
$$

The term above uses two distinguished partner names to interact with the connected entities: the partner name OG_{roleA} is used to catch messages coming from the left end of the wire, while OG_{roleB} is used for the right end (see the specification of *OG* in Table C.2).

An instance of a module can interact with instances of other service modules only after the successful completion of the discovery phase. In particular, when a requires-interface of the considered instance is triggered, it starts the discovery process by interacting with *Broker*. Consider, for example, the requires-interface *GA* of *OnRoadRepair*. After its activation, it sends a message with the business protocol *Garage* and the external policy *carUserSLAconstraints* to *Broker*. Then, *MatchmakingAgent* and *ConstraintSolver* execute a matchmaking process between the pair ("*Garage* **is** ...", *carUserSLAconstraints*) and the pairs of business protocols and SLA constraints stored in *Registry*. If matching succeeds, *Broker* sends back to *GA* a message with binding information.

The encoding of *GA* is as follows:

$GA \bullet trigger?\langle id_i \rangle.$
$(Broker \bullet disc!\langle OnRoadRepair, id_i, \text{“}Garage\ \texttt{is}\ \ldots\text{”}, carUserSLAconstraints \rangle$
$\mid [x_p, x_{acceptBooking}]\, OnRoadRepair \bullet GA?\langle id_i, x_p, x_{acceptBooking} \rangle.$
$[id_{ext}]\, (x_p \bullet create!\langle OnRoadRepair, id_{ext} \rangle$
$\mid x_p \bullet bindingInfo!\langle id_{ext}, acceptBookingResp \rangle$
$\mid * [x_{info}]\, GA \bullet acceptBooking?\langle id_i, \text{🔔}, x_{info} \rangle.$
$(x_p \bullet x_{acceptBooking}!\langle id_{ext}, \text{🔔}, x_{info} \rangle$
$\mid [x_{servicePrice}]\, OnRoadRepair \bullet acceptBookingResp?\langle id_{ext}, \text{✉}, x_{servicePrice} \rangle.$
$OG_{roleB} \bullet acceptBooking!\langle id_i, \text{✉}, x_{servicePrice} \rangle)$
$\mid \ldots))$

where id_i is the conversation identifier for the intra-module communication of the considered *OnRoadRepair*'s instance. The discovery process is triggered by a signal along the endpoint $GA \bullet trigger$, which is sent by the encoding of the component *OR* when the instance state is set to *READY* by transition *data_receiving*.

An instance of a service module can receive messages from the customer service that has created it by means of a provides-interface. For example, the encoding of the provides-interface *CR* of *RepairService* is

$[x_{getRequest}]\, RepairService \bullet bindingInfo?\langle x_{ext\_id}, x_{getRequest} \rangle.$
$* [x_{dataFromCar}]\, RepairService \bullet getRequest?\langle x_{ext\_id}, \text{🔔}, x_{dataFromCar} \rangle.$
$(CG_{roleA} \bullet getRequest!\langle id_{intra}, \text{🔔}, x_{dataFromCar} \rangle$
$\mid [x_{cost}]\, CR \bullet getRequest?\langle id_{intra}, \text{✉}, x_{cost} \rangle.\ x_{cust} \bullet x_{getRequest}!\langle x_{ext\_id}, \text{✉}, x_{cost} \rangle)$

The encoding of a provides-interface is symmetric to that of a requires-interface, i.e. it replaces the external identifier within an incoming message with the internal identifier. Notice that, in case of conversational interactions, to allow a provides-interface to reply to the corresponding requires-interface, the latter has to send to the former some binding information (e.g., in case of *GA*, the operation name *acceptBookingResp*).

We do not show here the encoding of components (we refer the interested reader to [30]). It suffices to know that a component is implemented by a COWS term that performs invoke/receive activities corresponding to SRML interactions according to the types of the interactions and the orchestration logic of the component. The term scheduling execution of the different transitions has been implemented assuming a fair execution of concurrent activities.

Dynamic aspects of the encoding. Suppose now that the COWS service implementing *RepairService* has already been published in the *Broker*'s registry. This means that it has already communicated to *Broker* its partner name, the business protocol of its provides-interface, and its external policy, by performing the invoke activity $Broker \bullet pub!\langle RepairService, \text{“}Customer\ \texttt{is}\ \ldots\text{”}, garageSLAconstraints \rangle$. Suppose also that the sensor monitor has already contacted, and instantiated, the module *OnRoadRepair* by invoking operation *create*, and that the created instance has performed transition *data_receiving*. A possible evolution of this scenario is described below.

(1) *OnRoadRepair triggers the process of discovery and binding.*

1. Execution of transition *data_receiving* of *OnRoadRepair* has set the state to *READY*. Thus, the triggering condition of its requires-interface *GA* holds true and, hence, the encoding of *GA* starts the discovery process. Assume that the broker, through *MatchmakingAgent* and *ConstraintSolver*, selects the pair ("*Customer* **is** ...", *garageSLAconstraints*) published in the repository by *RepairService* as the best match for the pair ("*Garage* **is** ...", *carUserSLAconstraints*) sent by *GA*. Then, *Broker* returns the message $\langle id_i, RepairService, getRequest\rangle$ along the endpoint $OnRoadRepair \bullet GA$. Therefore, x_p is replaced by the partner name *RepairService*, and $x_{acceptBooking}$ by *getRequest*. This way, the encoding of *GA* evolves into the following term:

$$
\begin{aligned}
&[id_{ext}]\,(\,RepairService \bullet create!\langle OnRoadRepair, id_{ext}\rangle \\
&\quad \mid RepairService \bullet bindingInfo!\langle id_{ext}, acceptBookingResp\rangle \\
&\quad \mid *\,[x_{info}]\,GA \bullet acceptBooking?\langle id_i, 🔔, x_{info}\rangle. \\
&\qquad (\,RepairService \bullet getRequest!\langle id_{ext}, 🔔, x_{info}\rangle \\
&\qquad\quad \mid [x_{servicePrice}] \\
&\qquad\quad OnRoadRepair \bullet acceptBookingResp?\langle id_{ext}, ✉, x_{servicePrice}\rangle. \\
&\qquad\quad OG_{roleB} \bullet acceptBooking!\langle id_i, ✉, x_{servicePrice}\rangle\,) \\
&\quad \mid \ldots\,)
\end{aligned}
$$

2. The requires-interface *GA* invokes the factory of module *RepairService* by executing the invoke activity $RepairService \bullet create!\langle OnRoadRepair, id_{ext}\rangle$. Hence, the following instance of *RepairService* is created:

$$
\begin{aligned}
&[id_{intra}]\,(\,ProvidesInt \mid RequiresInt \\
&\qquad \mid Wires \mid Components\,) \cdot \{x_{cust} \mapsto OnRoadRepair, x_{ext\_id} \mapsto id_{ext}\}
\end{aligned}
$$

GA also communicates the binding information to *CR* by invoking the operation *bindingInfo*.

(2) *OnRoadRepair initiates the conversation with RepairService.*

1. The component *OR* of the *OnRoadRepair*'s instance executes transition *reqToGarage* corresponding to the interaction *bookGarage*🔔. The block **sends** of this transition corresponds to the COWS activity $OG_{roleA} \bullet bookGarage!\langle id_i, 🔔, \text{"}gps = \ldots\text{"}\rangle$. Notably, in the encoding of component *OR* we take into account that it is connected to *GA* by means of the wire *OG*.

2. The wire *OG* catches the send event and adapts the endpoint of the activity of *OR* (i.e., *bookGarage*🔔) to the corresponding activity of the requires-interface *GA*. The executed COWS activity is $GA \bullet acceptBooking!\langle id_i, 🔔, \text{"}gps = \ldots\text{"}\rangle$.

3. The requires-interface *GA* catches the message and replaces the identifier id_i inside the message with the external identifier id_{ext}. Then, it invokes operation *getRequest* provided by the module *RepairService*, i.e. it performs the COWS activity $RepairService \bullet getRequest!\langle id_{ext}, 🔔, \text{“}gps = \ldots\text{”}\rangle$.

4. The message $\langle id_{ext}, 🔔, \text{“}gps = \ldots\text{”}\rangle$ sent by *GA* is delivered to the instance of *RepairService* created at step (1-ii) by means of the correlation identifier id_{ext}. This instance can receive messages from the instance of *OnRoadRepair* through the provides-interface *CR*, that replaces the external identifier in the incoming messages with the internal identifier. Thus, $CG_{roleA} \bullet getRequest!\langle id_{intra}, 🔔, \text{“}gps = \ldots\text{”}\rangle$ is executed.

(3) *RepairService processes the interaction and replies.*

1. The encoding of the wire *CG* acts as that of *OG*, i.e. it just renames the endpoints according to its specification. Then, it catches the message $\langle id_{intra}, 🔔, \text{“}gps = \ldots\text{”}\rangle$ sent over the endpoint $CG_{roleA} \cdot getRequest$ and forwards it along $GO \cdot handleRequest$. Hence, the performed activity is $GO \cdot handleRequest!\langle id_{intra}, 🔔, \text{“}gps = \ldots\text{”}\rangle$. Notice that the component *GO* exploits the partner name *GO* to receive messages from other entities.

2. The encoding of *GO* executes transition *reqResp*. This means that it performs the activity $GO \cdot handleRequest?\langle id_{intra}, 🔔, x_d\rangle$ and replies with $CG_{roleB} \cdot handleRequest!\langle id_{intra}, ✉, \text{“Eur 75”}\rangle$, where “Eur 75” is the value returned by $computePrice(\text{“}gps = \ldots\text{”})$.

3. The wire *CG* catches the reply message, replaces the name of operation *handleRequest* with *getRequest* and forwards the message to *CR*. The executed activity is $CR \bullet getRequest!\langle id_{intra}, ✉, \text{“Eur 75”}\rangle$.

4. *CR* renames the operation *getRequest* in *acceptBookingResp*, replaces the internal identifier id_{intra} with the external one id_{ext}, and sends the reply message to the instance of module *OnRoadRepair*. The executed activity is $OnRoadRepair \cdot acceptBookingResp!\langle id_{ext}, ✉, \text{“Eur 75”}\rangle$. Notice that, if there were more than one instance of *OnRoadRepair*, the identifier id_{ext} would guarantee that the message is properly delivered to the (requires-interface of the) proper instance of *OnRoadRepair*.

(4) *OnRoadRepair receives and processes the reply.*

1. *GA* catches the reply message, changes the operation name, replaces the identifier and forwards the message to *OG*. Thus, the executed activity is $OG_{roleB} \cdot acceptBooking!\langle id_i, ✉, \text{“Eur 75”}\rangle$.

2. *OG* changes again the name of the operation and delivers the message to the component *OR*. The executed activity is $OR \bullet bookGarage!\langle id_i, ✉, \text{“Eur 75”}\rangle$.

3. Finally, the receiving event triggers transition *respFromGarage* of *OR*, thus *OR*'s encoding executes $OR \bullet bookGarage?\langle id_i, \boxtimes, x_{price} \rangle$.

It is worth noticing that, if during the above computation a fatal error occurs within the component *OR* of the *OnRoadRepair*'s instance under consideration (i.e., its instance state is set to *ERR*), the encoding of *OR* would execute a forced termination of the COWS term implementing *OR*. This is done by means of a kill activity $\mathbf{kill}(k)$.