UNIVERSITÉ DE PROVENCE
U.F.R. M.I.M.
ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE E.D. 184

THÈSE en Cotutelle

présentée pour obtenir les grades de

Docteur de l'Université de Provence

et de

Dottore di ricerca in Informatica (Università di Firenze)

Spécialité : Informatique

par

Lucia ACCIAI

sous la direction de Silvano DAL ZILIO et de Michele BOREALE

Titre:

Algèbres de Processus pour les Architectures Orientées Service

A Process-Algebraic view of Service Oriented Architectures

soutenue publiquement le 12 juillet 2007

JURY

M. Michele BOREALE	Università di Firenze	*Directeur*
M. Silvano DAL ZILIO	CNRS	*Directeur*
M. Rocco DE NICOLA	Università di Firenze	*Examinateur*
M. François DENIS	Université de Provence	*Examinateur*
M. Matthew HENNESSY	University of Sussex	*Rapporteur*
M. Cosimo LANEVE	Università di Bologna	*Rapporteur*
M. Yannick PENCOLÉ	LAAS - CNRS	*Examinateur*

Acknowledgements

A thesis is never solely the work of the author. Support and encouragement come from different sources in various ways. In particular, I would like to thank my supervisor Silvano dal Zilio for giving me the opportunity to work with him in this project, for his friendly guidance and support during this three years. I am equally grateful to my italian supervisor, Michele Boreale, who encouraged and guided me with high competence since my Master thesis. Thank you for acting more like a friend rather than a supervisor.

I would like to tank all members of the Equipe MoVe, for the friendly and supportive atmosphere inherent to the whole group and for the lunches all together to the "Resto U". Many thanks to my office mates, especially to Nicolas, for his friendship and all his efforts in revising a previous version of my thesis, and to Pascal, for all (french) crosswords solved together during his stay in Marseille. I thank all members of the Dipartimento di Sistemi e Informatica and my PhD colleagues in Florence, for welcoming me during my periodical stays in Italy.

Thanks to my committee, especially Matthew Hennessy and Cosimo Laneve, who accepted to be reviewers. I am grateful to them for having read so carefully my thesis and for their comments.

I would like to thank the Laboratoire d'Informatique Fondamentale, the Université de Provence and the TraLaLa Project for their financial support of my work, living and trip expenses during these three years.

I am also grateful to all people I have met in France, for their friendship and support. Thanks to Alessandro, Thomas, Angelo, Luciana, Laura, Cristina and Cyprian for all dinners and chats during my first year stay in Marseille. Special thanks to Rita, with whom I have shared a flat during the last two years. Thank you for the relaxed living atmosphere in the house, for the nights off and for the Sunday walks.

Many thanks to my bosom friends Francesca, Elena, Francesca, Elisa and Marzia for all good moments shared since our first year at high school. Thank you for being there when I needed your support. Thank to my family, who have never lost faith in

me and in all my choices. Thank you for tolerating, supporting and encouraging me. Above all, I thank Andrea for his love and patience.

Abstract

The term *service* has been used for more than two decades and it is more and more common today tanks to the recent diffusion of *Web Services* (WS). WS are a way of implementing *Service Oriented Architecture* (SOA): an architectural style introduced in the 90s and developed around the concept of service. While there exist a lot of "high-level" languages and proposals for describing and defining WS, this technology still lacks formal basis. We argue that process calculi are a promising tool for modeling and studying SOA, in general, and WS, in particular.

In this thesis we will consider a few basic aspects of SOA and propose formal methods – based on process calculi, type systems and behavioral equivalences – for reasoning on them. The aspects of SOA we are interested in are: XML message passing, distributed and concurrent evaluation of XML queries, responsiveness of services, failures and transactions.

In the first part of the thesis we focus on the processing model. We consider SOA and WS as *communication-centered applications*, and propose XPi, a core calculus for XML messaging. XPi features asynchronous communications, pattern matching, name and code mobility and integration of static and dynamic typing. We study the typing issues arising from interactions of these features. Next, we take into account *distribution* of documents and services. We propose an asynchronous process calculus, named Astuce, where XML data and expressions are represented as distributed processes. The calculus is accompanied by a static type system based on regular expression types. In the second part, we concentrate on *responsiveness* and *atomicity*, two relevant non-functional aspects of services. A responsive system is one for which it is guaranteed that any service invocation will be eventually replied. We describe services as π-calculus processes and define two different type systems each of which statically ensures responsive usage of return channels, that implies responsiveness of services. After that, we study transactional aspects of WS. In particular, we formalize the optimistic approach of the *Software Transactional Memory* model from a process calculi viewpoint. We define an extension of the asynchronous CCS with atomic blocks

and prove a few interesting properties of this model using behavioral equivalences. E.g. we prove that our proposal can be used for encoding concurrency primitives like choice and mutiset-synchronization *à la* Join-calculus.

Résumé

Le terme de service fait partie du vocabulaire informatique depuis plus de deux décennies maintenant. Néanmoins, on le retrouve de plus en plus utilisé aujourd'hui de part la diffusion grandissante du modèle des Services Web (WS). Les Services Web sont lié aux Architectures Orientées Service (SOA), une approche à la conception des systèmes distribué, développée dans les années 90, et fondée sur l'interaction entre composants logiciels faiblement couplés (les services) ; les Services Web sont un moyen d'implanter les applications basées sur l'approche SOA.

Alors qu'il existe un grand nombre de propositions de langages de haut niveau pour spécifier ou implanter les WS, les technologies existantes manquent pour la plupart de bases formelles. Un postulat de notre approche est que les calculs de processus fournissent un outil de choix dans l'étude de la sémantique des Services Web. Dans cette thèse, nous cherchons à modéliser certains aspects de base des SOA et proposons des méthodes formelles pour les étudier. Les aspects que nous étudions sont : l'échange de documents XML comme valeur entre services ; l'évaluation distribuée de requêtes XML ; la disponibilité des services (service responsiveness) ; le comportement transactionnel. Pour ce faire, les outils utilisés sont essentiellement basés sur les calculs de processus ; les systèmes de types ; et les équivalences comportementales.

Dans la première partie de la thèse, nous nous concentrons sur le modèle opérationnel des services. Nous proposons un calcul de processus typé, XPi, afin de modéliser les WS. XPi étend le pi-calcul de Milner avec la possibilité d'envoyer des documents XML comme valeur dans un message ; il ajoute aussi des opérateurs de filtrage sur les valeurs, des primitives pour la mobilité de noms et la migration de code, et il intègre un système de typage mêlant approches statique et dynamique. Nous introduisons également un second modèle formel dans lequel les documents, comme les processus, sont distribués. Dans ce modèle, la vérification de la conformité des services utilise un système de types basé sur des expressions régulières de types.

Dans la seconde partie de la thèse, nous nous concentrons sur deux aspects non fonctionnels des services : la disponibilité et l'atomicité. On parle de service

disponible lorsqu'on peut garantir qu'un message d'invocation est toujours suivi (à plus ou moins longue échéance) par un message de réponse. Dans cette partie, nous modélisons un service par un processus du pi-calcul et définissons deux systèmes de types, d'expressivité croissante, permettant de certifier statiquement si un processus est disponible. Nous étudions aussi les aspects transactionnels des WS en adaptant au cadre des calculs de processus l'approche dite "Software Transactional Memory" (STM). Plus précisément, nous étendons CCS asynchrone en ajoutant la possibilité de déclarer des blocs d'actions devant s'exécuter atomiquement ; cette extension modifie sensiblement la sémantique du calcul et permet de prouver de nouvelles propriétés intéressantes.

Contents

List of Tables

Chapter

ONE

Introduction

In this introductory chapter, we set the scene of the whole thesis. In Section 2.2, we introduce Service Oriented Architecture and the main related concepts. In Section 2.3, we describe the process algebraic approach we follow and give an overview of the thesis. In Section 2.4 we summarize the main contribution of this thesis and in Section 1.4 we discuss related works.

1.1 Service Oriented Architecture and Web Services

The term *service* has been used for more than two decades. By now for example, transaction monitoring software used the term "service" in the early 1990s. In the same years, many client-server development efforts used the term "services" to indicate methods furnished by servers and accessed by clients by using remote method calls. Recently, *Web Services* (WS) have given this term more prominence and have renewed the interests in *Service-Oriented Architecture* (SOA).

Service Oriented Architecture. SOA can be considered as an architectural evolution, rather than a revolution, which captures many of the best practices of previous software architectures. SOA is just a revival of the *Component Based Architecture*, *Interface Based Design* and *Distributed Systems* of the 1990s and inherits their key aspects and principles, such as *distribution*, *autonomy*, *reusability* and *composability* of services. While these concepts have existed for decades, the adoption of SOA is accelerating due to the emergence of standard-based integration technologies. That is, *eXtensible Markup Language* (XML), a simplified subset of *Standard Generalized Markup Language* introduced to facilitate the sharing of data across different informa-

tion systems, and WS, XML based systems designed to support interoperable machine-to-machine interaction over a network.

The goal promoted by SOA is *loose-coupling*, that is separating users from service implementations with the aim of maximizing the reuse of application-neutral services and of increasing efficiency. A service can provide a single specific function, typically a business function, or it can perform a set of related functions. For instance, a service can convert one type of currency into another, can translate words, can analyze an individual's credit history, can process a purchase order or can handle operations involved in an airline reservation system. Thanks to loose-coupling, clients communicate with services according to well-defined *interfaces* and then leave it up to the service implementation (back-end) to perform the necessary processing. Modifications of the back-end services, for instance a revision of the airline reservation application, do not affect the way clients communicate with the service, provided that the interface remains the same. Interfaces, also called *contracts*, are at the heart of SOA. They are used for describing the functional part of services and provide indication for formal parameters and the constraints and policies defining the "contractual terms" to invoke the service. Moreover, they provide a description of non-functional aspects of services, such as security, transactionality and Quality of Service.

Web Services. Recently, WS have become the most prevalent approach to implementing SOA. This is possible thanks to the widespread acceptance of WS as a de-facto standard. WS are communication-centered applications and in general they use *message-passing* as communication paradigm. Message-passing allows clients and services to communicate and understand with each other across a wide variety of platforms and language boundaries. In spite of their diffusion, a globally accepted formal definition of WS does not exist. The most relevant attempt to define WS is due to the *World Wide Web Consortium* (W3C) [27]. The W3C defines WS and the WS *protocol stack* – a collection of computer networking protocols that are used to define, locate, implement and make WS interact with each other – as follows:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

HTTP has been introduced in 1996 by the W3C and, since then, it has become the most popular Web transport protocol.

XML has recently become the de-facto standard for describing data to be exchanged on the Web. As its name indicates, XML is a markup language; it involves the use of *tags* that "mark up" and describe the content of a document. Each tag identifies information in a document and its structure. An XML document is typically associated with a *schema* (defined using e.g. DTD [138] or XML-Schema [137]) that specifies the structure of the document: what tags are allowed, their structure, the type of data expected in a tag and so on. *Valid* XML documents must be well-formed (opening and closing tags must correspond and be "well-nested") and conform to the associated schema.

Agreeing on the meaning and structure of XML tags makes the use of XML an effective way to exchange data, but this is not sufficient for data interchange on the Web. For instance, the sender still needs some agreed-upon protocol for formatting an XML document so that the receiver understand what the main part of the message is and what part contains additional instructions or supplemental content. That's where *Simple Object Access Protocol* (SOAP) comes in. SOAP is a protocol for exchanging XML data over networks. The most common SOAP's messaging pattern is the *Remote Procedure Call* pattern, where a node (the client) sends a message (a request) to another node (the server), which in turn replies to the client by sending another message (a reply). The basic item of transmission in SOAP is a SOAP message, which, a part from the body of the message, contains a mandatory envelope specifying additional information – such as the encoding style – necessary for understanding and handling the received message. It is also possible to specify an optional header to indicate some additional processing at intermediate nodes reached along the message path from the sender to the receiver.

At the top level of the stack, *Web Services Description Language* (WSDL [58]) defines an XML Schema for describing WS's interfaces. A WSDL document defines interfaces in terms of *ports* and *messages* – data transmitted to (and sent by) ports – in an abstract way. WSDL disciplines the usage of ports by defining and associating them *port types*. A port type describes a collection of message types that are expected to be received at (and sent by) the associated port. To discover the description of a WS, a client needs to find the service's WSDL document. A typical way to do this for the client is to find a pointer to the WSDL document in the *Universal Description, Discovery and Integration* (UDDI) registry: the "Yellow Pages" for WS.

More on standards for Web Services. The previously mentioned standards address the basics of interoperable services and of the WS protocol stack. They ensure that a client can find a service and issue a request understood by the service, independently from where the client and the service reside and from what language they are coded in. In particular, the WSDL notion of port types provides a way to describe a service-usage discipline. Port types can be considered a basic kind of contracts between service requester and service provider. But they say nothing about non-functional requirements (like Quality of Service, time and order) that would appear in a business-level contract governing an actual service provision. Contracts require more than just WSDL and more "high -level" standards have to be adopted for WS to become a current practice.

WS-*Policy* provides a general purpose model and corresponding syntax to describe the policies of a WS; WS-*Security* describes security-related enhancements to SOAP messaging that provide for message integrity and confidentiality; *Web Services Business Process Execution Language* (BPEL4WS) can be used for coordinating WS invocations and interactions. And more, *Web Services Choreography Description Language* (WS-CDL) is an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behavior; *Web Services Level Agreements* (WSLA) defines the agreed-upon performance characteristics of a service and the way to evaluate and measure them; *Web Services Offerings Language* (WSOL) enables formal specification of multiple classes of service for one WS; and WS-*Transactions* defines mechanisms for transactional interoperability between WS and provides means to compose transactional qualities of service into WS applications. All these – and plenty of others – *emerging* proposed standards allow to specify non-functional requirements in great detail.

Putting the standards together. In the *Web Services Architecture* specification [27], the W3C explains how these and related technologies can be assembled together to deliver the greatest number of benefits. This architecture defines the relationships and constraints among the basic aspects of WS and includes four complementary *models*, each of them oriented to describe and define one specific aspect. Two models focus on the functional part of contracts: the so-called *message*-oriented and *service*-oriented models. The first one focuses on messages: structure, transport and manipulation. The second one focuses on service related aspects and on sequences of actions the involved parties are supposed to perform for correct interaction, the so-called *choreography*. The third model, called *policy*-oriented, defines sets of obligations and permissions useful to formally describe the non-functional part of contracts.

The last model, called *resource*-oriented, focuses on those aspects related to resources, such as discovery and description.

Moreover, in [27] the W3C identifies two major classes of WS: *arbitrary* and *rest-compliant*. Arbitrary WS may expose an arbitrary set of operations on "concrete" resources, like purchase order and shipping. The primary purpose of rest-compliant WS, instead, is the manipulation of XML representations of Web resources using a uniform set of "stateless" operations, like in search engines. Hence, the *Representation State Transfer* (REST) architecture can be seen as a model for building WS. REST has been introduced by Roy Fielding [70] as an architectural style for the World Wide Web (WWW). The REST Web is the subset of the WWW (based on HTTP) in which agents provide uniform interface semantics – essentially create, retrieve, update and delete on web resources – rather than arbitrary or application-specific interfaces. Furthermore, the REST interactions are *stateless* in the sense that the meaning of a message does not depend on the state of the conversation.

1.2 This thesis: a process algebraic approach to Web Services

All previously mentioned proposals allow to describe services in functional and non-functional terms. However, as stated in [24], some of them say nothing about the correct functioning of such application (like WSDL) or say too much and their expressive power causes them to be more about implementation rather than specification of useful and interesting properties (like BPEL4WS). The raising question is if there exist any interesting or useful specification mechanisms in between pure connectivity (WSDL) and full implementation (BPEL4WS) enabling the rigorous verification of the correct functioning of such applications. The answer is positive: formal methods. While several proposals rely on Petri nets, (timed) automata and digraphs as conceptual models for describing WS and WS composition (see e.g. [50, 1, 17, 90, 106, 118]), it is generally recognized that mobile *process calculi* (a.k.a. process algebras) are natural candidates for the formal specification of WS. Moreover, the last 30 years of research on the algebraic specification of systems has yielded simple, but powerful, methods (e.g. *type systems* and *behavioral equivalences*) for specifying and verifying their behavior. Generally speaking, process calculi provide a simple and expressive framework in which to reason about properties of concurrent, distributed and mobile systems. In particular, the π-calculus of Milner, Parrow and Walker [111] features *name-passing* – a distilled form of message-passing, the base

communication ontology adopted by WS – and is a promising candidate for laying the basis of a rigorous semantic theory of WS. The π-calculus is a powerful language, designed for describing the behavior of concurrent systems and is built around abstractions of ports: *channels*. π-processes are built in terms of synchronization constraints over input/output request (messages) at a collection of ports (channels). Furthermore, the π-calculus is at the basis of many other languages which target specific aspects of concurrent and distributed systems, like the spi-calculus [3] and the applied π-calculus [2], which have been used to study security protocols, and the distributed π-calculus [83] and safeDpi [82], which have been used for resource access control.

The idea underlying the process algebraic approach to WS is that of exploiting the common base ontology of WS and process calculi (communication) and describing services and clients as processes, so as to use or adapt existing techniques, or define new ones, for guaranteeing some properties of services. Notably, one can be interested in proving that the "contractual terms" defined in service interfaces are respected both by clients and services. A lot of recent proposals adhere to this approach (see Section 1.4 for a digression on related works) and this is the direction we follow in this thesis.

With the aim of describing and studying WS, we use process calculi for defining services and clients, and type system and behavioral equivalences for reasoning on them. Contracts are really powerful instruments for defining services, moreover they are usually expressed in human language, which, as well-known, is extremely rich and ambiguous. It follows that there are a lot of aspects and problems one has to deal with when designing and analyzing WS. Of course, it is not possible to cover all of them in this work. Here, as a starting point, we have chosen to focus on some important functional (Part I) and non-functional (Part II) aspects of WS as explained below.

In the first part of this thesis, Chapter 3 and 4, we focus on the computational model and, in particular, on the message-oriented architectural model. This choice is due to the fact that communication – messages, messaging and pattern matching – is the basic interaction mechanism of WS. We propose two computational models for arbitrary and REST-compliant web services, which focuses respectively on message exchange among clients and services and distribution of resources. In the second part, Chapter 5 and 6, we target policy-oriented model, and specifically two non-functional aspects of WS: responsiveness and transactionality. We judge these two as basic properties of services. Responsiveness because it guarantees to clients that

each request will be eventually processed and replied. Transactionality because it ensures that the execution of "critical" operations always satisfies some transactional properties (like e.g. consistency and isolation). As already stated, our work does not cover the huge number of – functional and non-functional – contractual constraints and aspects of WS, and there is a lot of work to do in this direction. As an example, we have not directly addressed neither the coordination and orchestration aspects, nor the security-related and resource management problems. A more detailed plan of the thesis follows.

Given that WS are mainly viewed as *communication-centered applications*, we investigate this aspect first. In Chapter 3 we propose a core calculus for XML messaging, named *XPi*. XPi is an asynchronous version of the π-calculus where: addresses on the net are represented as *names* (channels); messages passed around are XML documents; and clients and services are represented as processes that may communicate by sending and retrieving messages on channels. In XPi, an output action corresponds to sending an XML message over a channel and an input corresponds to querying channel's content. A type system ensures that messages comply with channel's capacity – that is with the structure expected by contract terms – and that there will not be type mismatch at run-time. XPi is equipped with a behavioral equivalence based on barbed bisimulation, useful for reasoning on the behavior of service implementations.

The model we propose in Chapter 3 is not appropriate for modelling WS described according to the REST architectural model, which focuses on *resources* and *distribution* rather than communication. In Chapter 4 we try to capture these aspects by focusing our attention on a computation paradigm inspired by search-engines and their applications. Actually, our aim is to define a model suitable for representing and querying large, distributed and dynamically generated XML documents. We propose a process calculus, *Astuce*, where XML data are processes that can be queried by means of concurrent pattern-matching and documents and pattern evaluations are concurrently distributed among locations. A type system, based on regular expression types, ensures that messages, patterns and retrieved information comply with the expected structure.

In the second part of the thesis, we focus on policy-oriented aspects, where policies are used for describing non-functional aspects of contracts. A policy is generically defined as either an *obligation* or a *permission* of either (not) do an action or (not) be in a particular state. Obviously, distinct contracts may give rise to distinct policies. Here we try to capture this fundamental guarantees, usually expected by service users:

responsiveness and transactionality.

Responsiveness ensures that any request to a service is eventually followed by a reply. In Chapter 5, we formally define responsiveness in a process calculus, represent services as π-calculus processes, and define two type systems each of which statically ensures responsiveness of processes. The first system allows one not only to ensure responsiveness, but also to say something about *latency* (response time) of services. In fact it allows for upper bounds on the number of actions preceding a reply. The second one is less restrictive than the first and expressive enough to let internal choice and orchestration patterns expressed in Cook and Misra's ORC orchestration language [59] (see § 1.4.1 for more details about ORC) be encodable into well typed processes.

Transactions, in a databases' theory sense, can be defined as minimal units of interaction with a database management system. Transactions are usually required to satisfy the well-known ACID properties: *atomicity* (a transaction must be either completed or aborted), *consistency* (a transaction cannot break global invariants), *isolation* (partial execution of transactions cannot be observed) and *durability* (the effects of a committed transaction cannot be undone). As WS transactions are *long-lived* (or *long-running*) and involve more than one WS, the ACID properties cannot in general be guaranteed. In fact, the most reasonable way for "undoing" the effects of a long-running transaction is to use programmable *compensations*. In other words, to use additional code that is activated when a failure occurs, with the aim of recovering a consistent state. Compensation is a global answer to failures that presuppose the existence of local mechanisms for ensuring transactionality in each node participating at the transaction. Local transactionality is usually implemented by using a lock-based approach, which is widely recognized as difficult and error prone. We study here the original approach found in the *Software Transactional Memory* (STM) model [84], where transactions are regulated optimistically. In particular, sequences of transactional actions are grouped into *atomic blocks* whose whole effect should occur atomically. In Chapter 6 we investigate this model from a process algebra perspective and define AtCCS, an extension of asynchronous CCS [110] with atomic blocks of actions. Suitable behavioral equivalences for processes and atomic blocks of actions are defined and a few interesting properties of this model are proved. E.g. we introduce some "laws of transactions" which allow to rewrite each atomic block in an equivalent one in normal form. We also show that the addition of atomic transactions results in a very expressive calculus, powerful enough to let easily encode other concurrency primitives such as (preemptive versions of) guarded choice and multiset-synchronization *à la* Join-calculus [72]. It is worth to notice that this last piece of work is only a first

attempt at studying the STM approach from a process calculi viewpoint, one where we consider a local model. Further work is in order to make the approach effective in a distributed setting.

1.3 Summary of contributions

We summarize below the main contributions of this thesis.

- We define XPi, an asynchronous core calculus for XML messaging which can be used for studying the communication features of WS. We introduce static and dynamic types for XPi and prove results on run-time safety. XPi is a joint work with M. Boreale and has been presented at FMOODS'05 [5].
- We present Astuce, a functional strongly-typed programming model for querying large and distributed XML documents. We introduce a type system for Astuce, which is based on regular expression types and is compatible with DTD and other schemes for XML, and prove type soundness of the calculus. Astuce is a joint work with M. Boreale and S. Dal Zilio and has been presented at TGC'06 [6].
- We propose type systems for statically ensuring responsiveness of services represented as π-calculus processes and apply them to non-trivial examples. This is a joint work with M. Boreale and has been presented at ASIAN'06 [7].
- We define AtCCS, a process calculus for atomic transactions that can be used for describing transactionality of services. AtCCS is based on the original optimistic approach of the STM model and is equipped with a congruence useful for reasoning on processes and atomic expressions and proving interesting equations, e.g. laws stating properties of atomic operators like commutativity and distributivity (with respect to choice) of action prefix. AtCCS is a joint work with M. Boreale and S. Dal Zilio and has been presented at ESOP'07 [8].

1.4 Related Work

In this section we consider works that are interesting to the present setting. Papers that are specifically related to our work will be examined in later chapters. We organize the section in three parts. In Section 1.4.1, we introduce the most popular languages proposed for the definition and description of WS. In Section 1.4.2, we make a roundup of various languages introduced for the processing of XML documents. Finally, in Section 1.4.3, we comment on some of the existing type systems for process calculi.

1.4.1 Languages for Web Services

The large spectrum of works on languages for WS can be divided into two subclasses: languages for the description of WS in terms of communication, orchestration, behavior and so on; and languages for the description of contracts.

Languages for communication and orchestration. Microsoft's *BizTalk* [23] is a system for orchestrating message-based applications on the Internet, which models processes as flowcharts, features short, long and timed transactions, and provide basic actions for sending or receiving data and controlling the workflow. BizTalk implements *XLang* [129]: an XML business process language inspired by the π-calculus. XLang provides a way to orchestrate applications and WS into larger-scale, federated applications by enabling developers to aggregate even the largest applications as components in a long-lived business process. XLang is a precursor of *Business Process Execution Language for Web Services* (BPEL4WS) [13]: an orchestration language for WS which allow to describe business processes in terms of the offered services and of the behavior of the involved actors.

A lot of works aiming at translating (a subset of) BPEL4WS into process algebras and at defining a formal semantics for it have been proposed: we present a brief overview of the most relevant ones. In [103], the authors consider a subset of BPEL4WS, which is sufficient to model the interactions among WS, and define its operational semantics. The language is supplied with a type system that is useful for disciplining communications. In [71], the authors propose a formal approach to model and verify the composition of WS workflow using the Finite State Processes (FSP) notation and the LTSA tool. Their paper introduces a translation of the main BPEL4WS structured activities into FSP. In [74], an approach to analyze BPEL4WS composite WS communicating through asynchronous messages is presented. The authors use guarded automata as an intermediate language from which different target languages (and tools) can potentially be employed. Ferrara [68] defines a two-way mapping between the Lotos process algebra and executable WS written in BPEL4WS, including in the mapping also faults, compensations, and event handlers. In [119], the authors propose a language called μ-BPEL including all primitives and structured activities within BPEL4WS, except compensation handler with scope. They propose a mapping from μ-BPEL to Timed Automata (TA) that is useful for using existing automatic tools for TA with the aim of checking properties of BPEL4WS processes. One of the main features of BPEL4WS is the fully programmable fault and compensation handling mechanism, which allows the user to specify the compensation behavior of

processes in application-specific manners. In [104, 120, 44] the authors formalize this key aspects and propose operational semantics for them. We will say more about compensation in Section 6.1.

Works aiming at reasoning only on some key aspects of WS and integrating such aspects into process calculi are closer to ours. Bierman and Sewell [21] define *Iota*: a concurrent XML scripting language used to program Home Area Networks. Iota is a strongly typed functional language with concurrency primitives inspired by the π-calculus. The type system ensures well-formedness of XML documents, but it says nothing about validity, that is about the conformance of documents with respect to DTD or schemes. Gardner and Maffeis [75] define *Xdπ*: a peer-to-peer model for reasoning about dynamic web data. Xdπ is a calculus for describing interaction between data and processes across distributed locations; it is focused on process migration and lacks of a type system.

Brown *et al.* [34] have defined *πDuce*: an extension of the π-calculus with native XML datatypes and operators for constructing and deconstructing XML documents. πDuce features asynchronous communication and code/name mobility. The considered pattern matching mechanism embodies built-in type checks. The language in [52] is basically a π-calculus enriched with a rich form of "semantic" subtyping and pattern matching. Pattern matching, similarly to πDuce's, performs type checks on messages.

A recent proposal is Cook and Misra's ORC [59] orchestration language: a basic programming model for structured orchestration of services. SCC [31] is a process calculus for orchestrating services influenced by ORC. SCC features explicit notions of service definition, service invocation and session handling. Similarly, COWS [102] is an ad-hoc process calculus for orchestrating WS. COWS borrows from already existing caluli its main ingredients – asynchronous communication, pattern matching, protection and killing activities – and, as BPEL4WS, uses a mechanism based on *correlation sets* for correlating different interactions, rather than explicitly representing sessions.

Contract languages. A large number of works can be classified in this field. Works aiming at defining schemes for XML documents are contracts languages: they allow to specify the format of documents expected by, and exchanged with, WS. We recall DTD and XML-Schema proposed by the W3C, RELAX-NG [122], *XDuce* [88, 89] and $\mathbb{C}$*Duce* [16] regular expression types and the schema language in [48].

Besides the schema languages for XML, we recall a series of works on *session types* [128, 86, 76, 130, 77] and *behavioral types* [92, 121, 54]. Session types are associated to session channels and specify the sequence and the types of messages sent and received respectively by clients and services. Behavioral types are associated to the

overall processes and abstract their behavior from a predefined point of view, e.g. by focusing on the sequences of calls performed and messages exchanged not only into a session. Behavioral types can be used for regulating aspects of safety, liveness, security and resource usage management. A more precise description of behavioral types can be found in § 1.4.3. Both session and behavioral type systems can be used for ensuring a certain kind of conformance between service requester and service supplier, hence for guaranteeing that the behavioral part of a contract is respected. With this purpose, Carpineti *et al.* [49], describe contracts as CCS processes and define suitable subcontract and compliance relations. They extract contracts out of processes and prove that a client completes his interaction with a service provided that the corresponding contracts comply. In [53], the authors depart from [49] and solve the transitivity problem of the subcontract relation. They give a direct characterization of strong compliance between clients and services and develop a new subcontract relation.

1.4.2 XML processing languages

There is a significant body of work in this field; each of these proposals supports sophisticated query primitives, but issues raised by communication and mobility are absent.

XDuce is a statically typed functional language for XML document processing. In the spirits of DTD, types are regular expressions and a powerful notion of subtyping naturally arises. XDuce supports regular expression pattern matching, which combines if-expressions, tag-checks and extraction of sub-nodes. $\mathbb{C}$*Duce* extends XDuce with a richer set of basic types, constructed types and higher-order functions. Closely related to XDuce, is the work in [46] where a spatial logic for reasoning about labeled directed graphs is introduced. This logic is used for providing a query language – for analyzing, manipulating and building graphs – and transducers – for relating input and output graphs. Cardelli and Ghelli [45] have defined *TQL*: a logic and a query language for XML, which is based on a spatial logic for the Ambient calculus [47] and operates on information represented as unordered trees. The high expressivity of the logic allows to express complex types, constraints and queries.

A different approach is followed in *ubQL* [123]. ubQL is a distributed query language for programming large-scale distributed query systems such as resource sharing systems. The language is inspired by the π-calculus and is obtained by adding a small set of mobile process primitives on top of any traditional query language. Relevant features are that queries are encapsulated into processes and can migrate between sites and streaming data can be handled.

We cannot forget a series of works aiming at evaluating XPath [133] or XQuery [132] expressions on streams of XML data. These proposals can be roughly divided in two approaches. The first is to provide efficient single-pass evaluators, working with one query at a time (generally XPath queries) on multiple documents, like in XSQ [56], SPEX [115] and XSM [105]. The second approach, in relation to peer-to-peer and event-notification systems, is to filter XML streams by a large number of queries, like in XFilter [10], YFilter [66], XTrie [55] andXPush Machine [80].

Finally, we would consider some works on streaming XML transformation. *XTiSP* have been introduced [112] and after ameliorated [113, 114] by Nakano. It is an XML transformation language intended for stream processing. A recent proposal in this field is *XStream* [73]. XStream is a Turing complete programming language that allow the programmer to write XML transformation in a functional style and use term rewriting for evaluating them in a streamed fashion.

1.4.3 Type systems for process calculi

In Chapter 3 and 5 we will introduce two sorting systems à la Milner for the π-calculus. Milner's sorting system [109] has been defined for preventing arity mismatch errors in communications involving polyadic π-calculus processes. The type system in [117] refines the one in [109] by distinguishing the ability to either read from a channel, write to a channel, or both read and write. A subsorting relation, which allow the use of a channel to be restricted to input-only or output-only in a given context, is defined. Beside these, a large number of type systems have been proposed for ensuring some interesting properties of π-calculus processes: a brief overview follows.

Kobayashi *et al.* [99] have introduced a type system ensuring *linearity* of names. This system guarantees that, at run-time, any linear name in a process will occur exactly once in input and once in output. Closely related to this, there is a series of works by Berger, Honda and Yoshida. In [136], they introduce a type system that guarantees *strong normalization* (termination and determinacy) of π-calculus processes. The type system in [135] is a refinement of those in [136] and ensures a *linear liveness* property. Two kinds of names are considered: *linear* (used exactly once) and *affine* (used at most once) and it is ensured that a well typed process eventually prompts for a free output on them. Kobayashi's type systems in [94, 100, 95, 96] can be used to guarantee that certain actions are *lock free*, that is, they succeed in synchronization if they become available. These type systems either lack a reasonable type inference algorithm or are not strong enough to ensure deadlock-freedom of processes using re-

cursion. In [97] an inference system is proposed and a new type system, which is a refinement of those in [95, 96] and allows to deal with recursion, is defined. In [64], *termination* of π-processes is ensured by using four different type systems obtained by successive refinements. In [125], a type system for ensuring *uniform receptiveness* of names is defined. It guarantees that names are used exactly once in input and are (immediately) available as input subjects at least as long as there are processes that can output on them.

In recent years, *behavioral* type systems for π-calculus processes have been widely studied. The aim of these works is to abstract the behavior of processes by using types, expressed as processes in (possibly "decidable") process calculi, and to check safety and/or liveness properties on types. Igarashi and Kobayashi's type system [92] is inspired by the previously cited works on linearity and deadlock-livelock freedom. Types for π-calculus processes are restriction-free CCS processes. Roughly, types are obtained from π-processes by considering each action prefix in turn and replacing any bound subject with a tag and turning each object into a CCS-annotation describing the behavior of the prefix continuation. The works in [121, 54] present type systems inspired by [92]. The main difference between these works and Igarashi and Kobayashi's, is that behavioral types here are more precise than in [92], because they are described by using full CCS.

Chapter

TWO

Introduction

2.1 Présentation

Le thème central de cette thèse est la modélisation et la vérification des *Web Services* (WS). Cette étude est menée, principalement, à l'aide des outils de la théorie des algèbres de processus et des systèmes de types.

Le terme de *service* fait partie du vocabulaire informatique depuis plus de deux décennies maintenant. Néanmoins, on le retrouve de plus en plus utilisé aujourd'hui de part la diffusion grandissante du modèle des *architectures orientées service*, ou SOA, pour *Service-Oriented Architecture*). Il s'agit d'une approche à la conception des systèmes distribué, développée dans les années 90, et fondée sur l'interaction entre composants logiciels faiblement couplés. On utilise dans ce cas le terme de service, à la place de celui de composant, tandis que le terme Web Services s'utilise pour décrire une classe d'applications basées sur l'approche SOA qui utilisent les technologies liées à XML.

Alors qu'il existe un grand nombre de propositions de langages de haut niveau pour spécifier ou implanter les WS, les technologies existantes manquent pour la plupart de bases formelles. Un postulat de notre approche est que les calculs de processus fournissent un outil de choix dans l'étude de la sémantique des Services Web.

Dans cette thèse, nous cherchons à modéliser certains aspects de base des SOA et proposons des méthodes formelles pour les étudier. Les aspects que nous étudions sont : l'échange de documents XML comme valeur entre services; l'évaluation distribuées de requêtes XML; la disponibilité des services (*service responsiveness* en anglais); le comportement transactionnel.

Pour mener à bien notre étude, les outils utilisés sont essentiellement basés sur les calculs de processus; les systèmes de types; et les équivalences comportementales.

La thèse se découpe en deux parties. Dans la première partie, nous nous concentrons sur le modèle opérationnel des services. Nous proposons un calcul de processus typé, *XPi*, afin de modéliser les WS. XPi étend le π-calcul de Milner avec la possibilité d'envoyer des documents XML comme valeur dans un message; il ajoute aussi des opérateurs de filtrage sur les valeurs, des primitives pour la mobilité de noms et la migration de code, et il intègre un système de typage mêlant approches statique et dynamique.

Nous introduisons également un second modèle formel dans lequel les documents, comme les processus, sont distribués. Dans ce modèle, la vérification de la conformité des services utilise un système de types basé sur des expressions régulières.

Dans la seconde partie de la thèse, nous nous concentrons sur deux aspects non fonctionnels des services: la disponibilité et l'atomicité. On parle de service disponible lorsqu'on peut garantir qu'un message d'invocation est toujours suivi (à plus ou moins longue échéance) par un message de réponse. Dans cette partie, nous modélisons un service par un processus du π-calcul et définissons deux systèmes de types, d'expressivité croissante, permettant de certifier statiquement si un processus est disponible.

Avant de conclure, nous étudions les aspects transactionnels des WS en adaptant au cadre des calculs de processus l'approche dite *Software Transactional Memory* (STM). Plus précisément, nous étendons une version asynchrone de CCS (le *Calculus of Communicating Systems* de Milner) en ajoutant la possibilité de déclarer des blocs d'actions devant s'exécuter atomiquement; cette extension modifie sensiblement la sémantique du calcul et permet de prouver de nouvelles propriétés intéressantes.

2.2 Architectures orientées service et Web Services

Le terme de *service* fait partie du vocabulaire de l'informatique depuis plus de deux décennies maintenant. On le trouve employé dans le courant des années 90, par exemple, dans le domaine des systèmes de suivis de transactions. À la même période, on retrouve l'expression de service utilisé dans le cadre des systèmes clients-serveurs, du type de CORBA, en particulier pour indiquer la capacité à réaliser des appels de méthodes distants.

Plus récemment, les *Web Services* ont remis au goût du jour l'utilisation de ce

terme, tout en renouvelant l'intérêt pour les *architectures orientées service*. Le modèle SOA peut être vu comme une évolution architecturale, plus que comme une révolution, qui capture la plupart des règles de "bonnes pratiques" énoncées par les concepteurs de logiciels distribués. À première vue, le concept de SOA est avant tout une réinterprétation de plusieurs approches popularisées dans les années 90: *architecture basée composants* (Component Based Architecture); *conception basée sur les interfaces* (Interface Based Design); et *systèmes répartis* (Distributed Systems). Le modèle SOA hérite d'ailleurs des aspects principaux et des principes de ces trois approches, en se concentrant sur: la distribution; l'autonomie; la réutilisabilité; la composition des services.

Bien qu'il s'agisse d'aspects centraux dans la conception des systèmes distribués, l'importance croissante du modèle SOA s'explique plus particulièrement par l'adoption de nouveaux standards d'intégration. On peut citer principalement XML (pour *eXtensible Markup Language*), un sous-ensemble simplifié de SGML, mis au point pour faciliter l'échange de données semi-structurées entre systèmes d'informations hétérogènes. Se basant sur XML, d'autres standards d'intégrations, regroupés autour du terme de Web Services, servent de fondement au modèle SOA. Il s'agit d'un ensemble de technologies facilitant les interactions machines-machines sur les grands réseaux, comme par exemple SOAP, WSDL, BPEL4WS, etc.

2.2.1 Architectures orientées service

L'objectif que se fixe l'approche SOA est d'assurer la coopération de composants logiciels *faiblement couplés*, s'exécutant sur un réseau hétérogène, sans administration centrale; c'est-à-dire le modèle du Web. Comme dans le cas de l'approche objets, l'idée est d'abstraire le plus possible des utilisateurs finaux les détails de l'implantation des services. Les buts visés sont de maximiser la réutilisation du code; d'augmenter le niveau d'abstraction des spécifications; d'améliorer l'efficacité et la fiabilité des applications.

La granularité des services peut être très diverse. Un service peut fournir une fonction spécifique très simple, typiquement une "fonction métier", ou il peut exécuter tout un ensemble de fonctions liées entre elles. Par exemple, on trouve des services qui se résument à calculer le montant d'une conversion entre devises ou bien à trouver les synonymes d'un mot. On trouve également des services aux fonctionnalités très complètes, comme par exemple les services permettant de traiter des ordres d'achats ou bien les réservations sur les lignes aériennes.

Selon le principe de "composition faiblement couplée", un client ne communique

avec un service qu'à travers une interface bien définie, fixée à l'avance. Un avantage de ce choix est qu'un service peut être changé en cours d'exécution sans pour autant mettre en péril l'exécution d'un traitement. Par exemple, l'application de gestion d'une ligne aérienne peut être mise à jour dynamiquement, sans rien changer pour l'utilisateur, à condition que l'interface demeure la même.

Finalement, un élément de base du modèle SOA est le recours à la notion d'interface, également appeléee *contrat*. On retrouve ici un des nombreux traits communs avec les approches à base de composants, ou avec les modèles de programmation répartis du type CORBA. En premier lieu, les contrats sont employés pour décrire la partie fonctionnelle des services. Ils indiquent les paramètres formels de chaque service, les possibles contraintes, ainsi que les politiques (les limites contractuelles) qui régissent l'appel au service. Dans certains formalismes, les contrats peuvent aussi être utilisé pour fournir une description non fonctionnelle des services, tel que les contraintes concernant la sécurité, la qualité de services, ou encore ayant trait au comportement transactionnel.

2.2.2 Web Services

Récemment, les WS sont devenus l'approche la plus répandue pour l'implantation d'applications suivant le modèle SOA. À tel point que les WS sont devenus aujourd'hui un standard *de facto*. Mais c'est un standard ouvert et encore mal défini. Ainsi, clients et services communiquent entre eux au-dessus d'une grande variété de langages de programmation et de plate-formes logicielle. De plus, malgré la large diffusion de ce modèle, une définition formelle des WS, globalement admise par chacun, n'existe pas.

Dans cette thèse, nous choisissons pour définir les WS l'approche qui consiste à définir un service Web comme une application utilisant une "pile de protocole" bien spécifiés, basés sur XML. De manière grossière, la "*pile de protocole* WS" est une collection de protocoles réseaux servant à définir, localiser, implanter et faire interagir les WS entre eux. (Parmi tout ces protocoles on peut nommer prioritairement WSDL, SOAP, UDDI.) Nous nous rapprochons en cela de la définition due au *World Wide Web Consortium* (W3C) [27], qui nous semble une des plus appropriées. Ainsi, le W3C définit les WS de la manière suivante:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its descrip-

tion using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

2.2.3 Pile de protocoles Web

Nous présentons les différents protocoles WS, en privilégiant l'ordre chronologique de leur apparition plutôt que l'ordre d'utilisation sur la pile.

HTTP. Parmi les premiers protocoles entrant en jeux dans la définition des WS, apparaissent les protocoles liés au mode de transport, c'est-à-dire à la manière dont les appels et les communications sont véhiculés entre services. Les protocoles de transport déjà existant pour Internet sont utilisés. Le plus souvent, on retrouve l'utilisation de HTTP, un protocole introduit en 1996 par le W3C et qui, depuis lors, c'est imposé comme le protocole de transport le plus populaire pour le Web. Si HTTP est le choix privilégié, on peut noter que certaines spécifications prennent également en compte l'utilisation des protocoles liés aux courriers électroniques, en particulier SMPT, et que beaucoup de protocoles WS sont totalement agnostique quand au choix du protocole de transport.

XML. À côté du choix du "mode de transport", il convient également de choisir le mode de représentation des données qui sont échangés par les services. Le choix le plus courant est XML (le *eXtensible Markup Language*), une technologie beaucoup plus récente que HTTP, destinée à être le format privilégié pour l'échange de données sur le Web. XML est devenu très rapidement un standard de fait. Comme son nom l'indique, XML est un langage se basant sur des balises pour marquer ("mark up") le sens des différentes données contenues dans un document, ainsi que pour les hiérarchiser. Les balises permettent d'identifier et de localiser l'information dans un document, mais elles servent aussi à structurer cette information.

XML Schema. Les standards présentés ici prennent en compte la nécessité de typer les échanges entre services, c'est-à-dire le besoin de faire respecter aux messages un certains nombres de contraintes; contraintes qui permettent à chaque noeud du réseau qui reçoit un message de le comprendre. On utilise ici les technologies de typage pour XML, En effet, le plus souvent, un document XML est associé à un *schéma*, définit grâce à des technologies telles que les DTD [138] ou les schémas XML [137], qui spécifie et contraint la structure du document. Typiquement, un schéma décrit les balises que peut contenir le document et dans quel ordre ceux-ci peuvent apparaître. Il peut

également servir à décrire le type des données pouvant apparaître sous une balise. Un document associé à un schéma est *valide* si il est bien formé (les "balises ouvrantes et fermantes" sont bien balancées) et si il respecte les contraintes imposées par le schéma.

SOAP. Bien que la possibilité de fixer un choix sur le sens et la structure des balises constitue un avantage lorsqu'il s'agit de s'échanger des documents XML, ceci est loin d'être suffisant dans un scénario d'échange de données sur le Web. Par exemple, l'émetteur et le récepteur des données doivent encore se mettre d'accord sur un protocole de formatage. C'est ici qu'entre en jeux SOAP (qui fut jusqu'à récemment un acronyme pour *Simple Object Access Protocol*). SOAP est un protocole permettant d'encapsuler et de sérialiser des appels de fonctions distantes sur un réseau. Le "pattern" le plus courant pour l'échange de message SOAP est celui de l'appel de fonction distant (*Remote Procedure Call*), dans lequel un noeud du réseau (le client) envoie un message (une requête) à un autre noeud (le serveur). Dans cette pattern, le client attend un message de réponse du serveur à sa requête. Les messages SOAP sont basé sur la métaphore du courrier postal: l'élément de base dans une transmission SOAP est le *message*, qui se compose d'un corps englobé dans une *enveloppe*. L'enveloppe contient les en-têtes et les informations nécessaires à acheminer et traiter le message. Le protocole SOAP permet également de définir des traitements à exécuter sur les "noeuds intermédiaires" qui routent le message.

WSDL. Pour finir cette liste de protocole, le *Web Services Description Language* [58] (WSDL) est un vocabulaire XML permettant de décrire l'interface des WS. Un document WSDL définit les actions (*operations*) fournies par un WS ainsi que les données transmises aux actions (*messages*). Une collection d'opérations reliées entre elles est appelé un port (*port type*). Cette description est abstraite, mais une spécification WSDL permet également de décrire l'implantation concrète du service en fournissant des informations de liaison (*binding*). Les informations de liaison associent à chaque port le protocole réseaux qui doit être employé. Elles donnent aussi, par exemple, des URLs ou des numéro IP à utiliser, et spécifient le format des messages. On peut également noter qu'il existe un mécanisme permettant de découvrir dynamiquement la description WSDL d'un service. La manière générique de trouver une description WSDL consiste à interroger un registre des services, en utilisant le protocole *Universal Description, Discovery and Integration* (UDDI), sorte de "pages jaunes" pour les services.

2.2.4 Remarques concernant les standards ws

Les standards présentés précédemment adressent les fonctionnalités de bases permettant de faire interopérer les services: format et typage des données; format des messages; transport. La réunion de ces standards forme la pile de protocole WS. C'est elle qui fournit les moyens de s'assurer qu'un client peut trouver les services dont il a besoin, et qu'il peut envoyer des requêtes qui seront comprises par ces services, indépendamment du lieu où s'exécute le client ou du langage utilisé pour son implantation.

Ces standards représentent un service minimum, mais ne sont pas suffisant à eux seul. Pour permettre une plus grande adoption de modèle des services Web, il reste encore à développer et faire adopter des standards "de plus haut-niveau". Ainsi, le standard WSDL apparaît souvent insuffisant lorsqu'il s'agit de décrire le cahier des charges d'un service (son contrat). En particulier, WSDL permet uniquement de définir un ensemble d'obligations fonctionnelles. Rien n'est dit sur les besoins non-fonctionnel du service ou sur les contraintes sémantique qu'on voudrait pouvoir imposer aux données qu'il produit.

Cette remarque sur les limitations de WSDL est très commune et il existe aujourd'hui beaucoup de propositions visant à étendre et enrichir WSDL. Néanmoins, il s'agit d'une véritable jungle de nouvelles spécifications, chacune s'adressant le plus souvent à améliorer les standards existant de manière parcellaire. Nous donnons quelques exemples de ces standards XML liés aux WS, en étant loin d'être exhaustif:

- WS-*Policy* fournit un modèle général, et un métalangage (une syntaxe correspondante), pour décrire les politiques liées à un WS.
- WS-*Security* décrit des extensions aux échanges de messages SOAP liées aux aspect sécurité. Plus précisément à l'intégrité et la confidentialité des échanges.
- *Business Process Execution Language for Web Services* (BPEL4WS) est une proposition de standard pour coordonner les interactions et les invocations de WS.
- Encore au-dessus, *Web Services Choreography Description Language* (WS-CDL) est un métalangage XML qui permet de décrire des collaborations multi-agents (peer-to-peer) en définissant, d'un point de vue global, leurs comportements observable.
- *Web Services Level Agreements* (WSLA) définit des protocoles utilisé pour fixer, entre services, des choix sur les caractéristiques de performances et sur les moyens de les évaluer.
- *Web Services Offerings Language* (WSOL) rend possible la spécification formelle

de multiples classes de services pour un WS fixé.

- Finalement, WS-*Transactions* définit des mécanismes transactionnels pour l'interaction entre services et fournit des moyens pour composer ces politiques transactionnelles entre elles.

Il faut retenir que tous ces "standards émergents", et bien d'autres encore, ont été motivé par le besoin de décrire, dans les plus grands détails possible, les contraintes non-fonctionnelles d'une application.

2.2.5 Intégration des standards

Dans la spécification *Web Services Architecture* [27] (WSA), le W3C tente de démontrer comment l'ensemble des technologies que nous avons présentés dans la section précédente peuvent s'intégrer afin de fournir une plate-forme d'exécution la plus riche possible. Cette architecture définie les relations, ainsi que les contraintes, entre les aspects de bases des services. Elle inclus quatre modèles complémentaires, chacun orienté vers la description d'un aspect spécifique. Deux de ces modèles se concentrent sur la partie fonctionnelle des contrats, le modèle nommé *orienté-messages* et le modèle *orienté-service*. Le premier de ces deux modèles s'intéresse plus particulièrement aux messages échangés: à leur structure, leur transport et à leur manipulation. Alors que le second modèle se concentre sur les aspects liés aux services en eux-mêmes et sur la *chorégraphie*, c'est-à-dire la séquence des actions réalisées par chacune des parties impliquées dans une interaction. Le troisième modèle, *orienté-politique*, permet de définir des ensembles d'obligations et de permissions, qui sont stipulées dans la partie non-fonctionnelles des contrats. Finalement, le dernier modèle, *orienté-ressources*, se concentre sur les aspects de découverte et de description des ressources liés aux services.

Dans la spécification WSA, le W3C identifie deux classes principales de WS, la classe *arbitrary* et la classe *rest-compliant*. Les WS *arbitrary* peuvent exposer un ensemble arbitraire d'opérations sur des ressources concrètes. À l'opposé, le but premier d'un WS REST est de manipuler les représentations XML de ressources Web en utilisant un ensemble uniforme d'opérations *stateless*. Le terme REST est l'acronyme de *Representational State Transfer* et correspond à un type d'architecture logicielle qui se concentre sur la manière dont les ressources doivent être définie et adressée. REST a été introduit par Roy Fielding, dans sa thèse [70], comme un style de programmation pour le Web. Plus particulièrement, le Web REST est le sous-ensemble du Web, basé sur HTTP, dans lequel les agents fournissent une interface à la sémantique uniforme; ils fournissent essentiellement les opérations create, retrieve, update et delete. De plus,

les interactions REST sont sans état (*stateless*), dans le sens où la signification d'un message ne dépend pas de l'état de la conversation ou de la ressource qui est adressée.

Un exemple typique d'application *arbitrary* est un service de gestion de bons de commandes, qui permet de mettre en réseaux des bases de données propriétaires, tandis que les moteurs de recherches sont un bon exemple d'architecture REST.

2.3 Une approche algèbre de processus pour l'étude des Web Services

Nous avons décrit, dans les sections précédentes, un grand nombre de propositions permettant de décrire les services de manière très détaillé. Néanmoins, il n'existe pas encore aujourd'hui de méthodologie, bien implantée, permettant de vérifier si un service répond réellement à sa spécification. Nous avons besoin de méthodes formelles pour répondre à ce problème.

Il existe plusieurs approches à l'étude formelle des WS et à leur composition. Ces travaux se basent sur différents modèles conceptuels, tel que les réseaux de Petri, les automates (temporisé ou non), les digraphes, etc. (voir par exemple [50, 1, 17, 90, 106, 118, 15]). Il est généralement reconnu que les calculs de processus mobile, du type du π-calcul de Milner, Parrow et Walker [111], sont des candidats naturels au rôle de modèle pour la spécification et la vérification formelle de WS. En effet, les algèbres de processus fournissent un cadre à la fois simple et expressif qui permet de raisonner sur les problèmes lié à la concurrence, la distribution et la mobilité des systèmes. Par exemple, les techniques liées aux *systèmes de types* (dans le style des types comportementaux) peuvent être adaptées pour permettre de réguler les échanges de messages entre services. Ces mêmes techniques peuvent aussi être utilisées pour contraindre le comportement des processus, et des outils tel que les *équivalence comportementales* peuvent être employés pour établir des correspondances entre différents niveaux d'abstractions dans la description des services.

Parmi les nombreuses algèbres de processus pouvant être utilisé pour cette tâche, on peut mettre en avant le π-calcul, qui se caractérise par son approche basée sur le passage de noms. On peut d'ailleurs noter que le π-calcul est à la base de plusieurs langages visant différents aspects des systèmes concurrents et distribués. Par exemple la sécurité et les protocoles cryptographique, avec le spi-calcul de Abadi et Gordon [3] ou le π-calcul appliqué [2], ou encore le contrôle d'accès pour les ressources distribués, avec le π-calcul distribué de Hennessy et Riely ou safeDpi [82].

L'idée sous-tendant une approche "algèbre de processus" pour l'étude des WS est de décrire les services et les clients comme des termes d'un calcul de processus et d'utiliser (en les adaptant) les techniques de vérification existantes pour garantir les propriétés des services. Nous sommes loin d'être les premiers à invoquer cette approche et un grand nombre de propositions récentes vont dans le sens que nous poursuivons ici (voir la Section 1.4 pour une discussion sur l'état de l'art correspondant). Dans cette thèse, nous représentons les clients et les services par des processus et nous utilisons des notions adéquates de systèmes de typage et d'équivalences comportementale pour vérifier que certaines propriétés sont respectées. (Nous nous intéressons à la fois à des propriétés fonctionnelles et à des propriétés non-fonctionnelles, exprimées de manière contractuelle ou non.)

Suivant la ligne développée dans [27], nous considérons certains des aspects de base des WS et nous proposons des méthodes formelles permettant de raisonner sur ces aspects. La thèse peut se diviser en deux grandes parties.

Dans la première partie, chapitres 3 et 4, nous nous concentrons sur le modèle opérationnel des WS ainsi que sur le modèle architectural, orienté-messages. Nous proposons deux modèles, un pour les Web services *arbitrary* et un pour les services REST-*compliant*, qui se concentrent pour le premier sur la communication entre clients et services, et pour le second sur la distribution des ressources sur le réseaux. Dans la deuxième partie, chapitres 5 et 6, nous nous concentrons sur le modèle orienté-politique et abordons plus spécifiquement les propriétés de *disponibilité* des services. Nous étudions également les aspects transactionnels des processus, un autre exemple de propriétés non-fonctionnelles.

Notre travail est, bien sûr, loin d'être complet et il reste encore beaucoup de propriétés, et de directions, à explorer. Par exemple, nous n'avons pas directement adressé les aspects liés à la coordination ou à l'orchestration des services, ni les problèmes lié à la sécurité.

Nous donnons ci-après un plan plus détaillé du manuscrit de thèse.

Étant donné que les WS sont principalement abordés comme des applications orienté-messages, nous étudions en premier ce dispositif. Dans le chapitre 3, nous proposons un calcul de processus pour la transmission de messages XML, appelé *XPi*. Le calcul XPi peut se voir comme une version asynchrone du π-calcul dans lequel: les "adresses internet" sont représentées par des noms (de canaux); les messages échangés sont des documents XML (et pas uniquement des tuples de noms, comme dans le π-calcul); les clients et les services sont représentés comme des processus qui peuvent

communiquer en envoyant et en interrogeant des messages sur des canaux. Dans XPi, une émission correspond à l'envoie d'un message sur un canal de communication et une réception correspond à interroger (appliquer une requête) le contenu d'un canal. Le calcul est munis d'un système de type (statique) qui permet de s'assurer du schémas des messages contenus dans un canal. Nous définissons une relation d'équivalence comportementale pour XPi basée sur la notion de bisimulation à barbes (*barbed bisimulation*), qui se révèle utile pour raisonner sur le comportement de l'implantation d'un service sous forme de processus.

Le modèle basé sur XPi, que nous proposons au chapitre 3, n'est pas approprié pour modéliser les WS bâti sur l'approche REST, qui se concentrent plus sur la distribution des ressources plutôt que sur la communication entre agents. Dans le chapitre 4, nous essayons de capturer ces aspects différents en développant un modèle basé sur un calcul de pattern distribués. Notre but est de définir un modèle approprié à la représentation et à l'interrogation de grands documents XML distribués, voir même de documents générés dynamiquement, et donc potentiellement infinis. Pour ce faire, nous proposons un calcul de processus, *Astuce*, dans lequel: les données (les documents XML) sont des processus qui peuvent être interrogé au moyen de requêtes concurrentes; les noeuds des documents et l'évaluation des requêtes sont explicitement distribués; un système de typage statique, basé sur la notion d'expressions régulières de types, s'assure que les messages, les requêtes et le résultat des requêtes sont conformes à un schéma définit à l'avance.

Dans la deuxième partie de cette thèse, nous nous concentrons sur le modèle orienté-politique, dans le cadre où les politiques sont employées pour décrire des aspects non fonctionnels des contrats. Dans l'approche que nous suivons, une politique est définie de manière générique comme soit une obligation, soit une permission, de faire une action ou d'atteindre un certains état. Évidemment, des contrats différents peuvent engendrer des politiques différentes, par conséquent, nous essayons ici de capturer une sous-classe de garanties fondamentales, habituellement prévues par les utilisateurs des services. Plus précisément, nous étudions des propriétés liées à la disponibilité et au comportement transactif des services.

La *disponibilité*, ou *responsiveness* en anglais, est la propriété associée aux systèmes tel que toute demande de service est assurée d'être suivie par une réponse (du moins en absence de panne et/ou de perte de messages). Dans le chapitre 5, nous définissons formellement la propriété de disponibilité dans un calcul de processus; nous représentons les services comme des processus du π-calcul; et nous définissons deux systèmes de type de complexité croissante qui assurent statiquement la disponi-

bilité des processus. Le premier système permet de s'assurer de la disponibilité d'un processus mais permet également d'obtenir des précisions sur la latence du service (son temps de réponse). Plus précisément, il permet d'inférer une limite supérieure sur le nombre d'actions précédant une réponse. Le second système de types est moins restrictif que le premier système et permet de coder certaine pattern de programmation intéressante. Par exemple, ce système de types est assez expressif pour typer le codage, sous forme de processus, du choix interne ainsi que les différentes patterns d'interaction qu'on retrouve dans langage d'orchestration ORC de Cook et Misra [59] (voir la section 1.4.1 pour plus de détail sur le langage ORC).

En ce qui concerne le comportement transactif, nous nous intéressons à l'ajout de *transactions* aux services, aux sens des bases de données, c'est-à-dire à des unités minimales d'exécution qui exhibent des propriétés d'atomicité et de durabilité. Plus particulièrement, nous nous intéressons aux propriétés dites ACID : l'atomicité (une transaction doit être soit validée soit avortée); la consistance (une transaction ne peut pas casser les invariants globaux); l'isolation (on ne peut pas observer l'exécution partielle d'une transaction); et la durabilité (les effets d'une transaction validée ne peuvent pas être annulé). Toutes ou parties de ces propriétés ne se retrouvent généralement pas dans les modèles de transactions pour les WS. En effet, on retrouve le plus souvent un modèle de transactions par compensation, plus approprié aux transactions distribuées, de longue durées et faiblement couplées. Nous étudions l'approche originale introduite par Herlihy dans le modèle des Transactions Mémoire Logicielle, ou STM pour *Software Transactional Memory* [84]. Dans cette approche, des séquences d'actions peuvent être groupé au sein d'un "bloc atomique" dont l'effet doit s'exécuter atomiquement. Dans le chapitre 6 nous étudions ce modèle en utilisant encore une fois une approche algèbre de processus. Nous définissons un calcul de processus, AtCCS, qui est une extension de CCS asynchrone [110] avec des blocs atomique. Nous définissons également une notion d'équivalence comportementale et prouvons un ensemble de lois algébriques intéressante pour ce modèle. Nous prouvons par la même que l'ajout des STM à CCS résulte en un calcul très expressif, permettant de coder facilement des primitives de la programmation concurrente; par exemple (une version préemptive du) choix non-déterministe ou la multi-synchronisation à la join-calcul.

2.4 Résumé de nos contributions

Dans cette section, nous résumons les principales contributions de notre travail de thèse.

- Nous avons défini XPi, un calcul de processus asynchrone pour l'échange de message XML, pouvant servir de base à l'étude des aspects communications dans les Web services. Nous avons également défini des systèmes de typage, statique et dynamique, pour XPi et prouvé que les processus bien typé était sauf. Les résultats sur XPi sont un travail commun avec M. Boreale et ont été publié à la conférence FMOODS'05 [5].
- Nous avons développé la théorie du calcul Astuce, un modèle de programmation fonctionnel et concurrent, fortement typé, pour l'interrogation de très grand documents XML distribués. Nous avons défini un système de type pour Astuce qui se base sur l'utilisation d'expression régulière de type, une approche compatible avec les DTD, ainsi que d'autres formalismes de schémas pour XML, et qui est compatible avec des langages de programmation fonctionnel tel que XDuce. Astuce est un travail en commun avec M. Boreale et S. Dal Zilio; les résultats présenté ici ont été publiés à la conférence TGC'06 [6].
- Nous avons proposé une nouvelle technique pour vérifier statiquement la "disponibilité" d'un service décrit comme un processus du π-calcul, et nous avons appliqué cette technique à des exemples non-triviaux. Il s'agit d'un travail avec M. Boreale; ces résultats ont été publié à la conférence ASIAN'06 [7].
- Nous avons défini AtCCS, un calcul de processus qui étend CCS avec des transactions atomique, et avons développé une nouvelle notion d'équivalence comportementale pour raisonner sur ces processus. Le calcul AtCCS se base sur le modèle des STM, qui est une approche optimiste permettant d'ajouter des transactions atomique au niveau des langages de programmation. Il s'agit d'un travail en commun avec M. Boreale et S. Dal Zilio, publié à la conférence ESOP'07 [8].

Part I

Processing models

Chapter

THREE

XPi: a typed process calculus for XML messaging

In this chapter we present XPi*, a core calculus for* XML *messaging. XPi features asynchronous communications, pattern matching, name and code mobility, integration of static and dynamic typing. Flexibility and expressiveness of this calculus are illustrated by a few examples, some concerning description and discovery of web services. In XPi, a type system disciplines* XML *message handling at the level of channels, patterns, and processes. A run-time safety theorem ensures that in well-typed systems no service will ever receive documents it cannot understand, and that the offered services, even if re-defined, will be consistent with the declared channel capacities. A notion of barbed equivalence is defined that takes into account information about service interfaces.*

3.1 Introduction

The message-oriented model [27] describes WS from the point of view of the communication. It focuses on those aspects of the architecture that relate to message exchange and processing. Specifically, this model is not concerned with any semantic significance of the content of a message or its relationship to other messages. However, it focuses on the structure of messages, on the relationship between message senders, message receivers and other message processors. In this chapter, we aim at giving a concise semantic account of XML messaging and of the related typing issues. To this purpose, we present *XPi*, a process language based on the asynchronous π-calculus. Prominent features of XPi are: XML communication, patterns generalizing ordinary inputs, ML-like pattern matching, and integration of static and dynamic typing. Our objective is to study issues raised by these features

in connection with name and code mobility. A more precise plan of the chapter follows.

Syntax and reduction semantics of the calculus are introduced in Section 3.2. In XPi, resource addresses on the net are represented as *names*, which can be generally understood as channels at which services are listening. *Messages* passed around are XML documents, represented as tagged/nested lists, in the vein of XDuce; in what follows we usual refer to messages by using the term *documents*. Services and their clients are *processes*, that may send documents to channels, or query channels to retrieve documents obeying given patterns. Documents may contain names, which are passed around with only the *output capability* [117]. Practically, this means that a client receiving a service address cannot use this address to re-define the service. This assumption is perfectly sensible, simplifies typing issues, and does not affect expressive power (see e.g. [28, 107]). Documents may also contain mobile code in the form of *abstractions*, roughly, functions that take some argument and yield a process as a result. More precisely, abstractions can consume documents through pattern matching, thus supplying actual parameters to the contained code and starting its execution. This mechanism allows for considerable expressiveness. For example, we show that it permits a clean encoding of encryption primitives, hence of the spi-calculus [3], into XPi.

Types (Section 3.3) ensure the validity of documents and discipline their processing at the level of channels, patterns, and processes. At the time of its creation, each channel is given a *capacity*, i.e. a type specifying the format of documents that can travel on it. *Subtyping* arises from the presence of star types (arbitrary length lists) and union types, and by lifting at the level of documents a subtyping relation existing on basic values. The presence of a top type $\mathbf{T}$ enhances flexibility, allowing for such types as "all documents with an external tag $\mathtt{f}$, containing a tag $\mathtt{g}$ and something else", written $\mathsf{T} = \mathtt{f}[\mathtt{g}[\mathbf{T}], \mathbf{T}]$. Subtyping is contravariant on channels: this is natural if one thinks of services, roughly, as functions receiving their arguments through channels. Contravariance calls for a bottom type $\mathbf{\bot}$, which allows one to express such sets of values as "all channels that can transport documents of some type $\mathsf{S} < \mathsf{T}$", written $ch(\mathtt{f}[\mathtt{g}[\mathbf{\bot}], \mathbf{\bot}])$. Interplay between pattern matching, types, and capacities raises a few interesting issues concerning *type safety* (Section 3.4). Stated in terms of services accessible at given channels, our run-time safety theorem ensures that in well-typed systems, first, no service will ever receive documents it cannot understand, and second, that the offered service, even when re-defined, will comply with the statically declared capacities. The first property simply means that no process will ever output documents

violating channel capacities. The second property means that no service will hang due to an input pattern that is not consistent with the channel's capacity (a form of "pattern consistency").

type checking is entirely static, in the sense that no run-time type checking is required, while in πDuce [34] and the language of [52] pattern matching also performs type check on messages. Moreover, in both [34, 52] the type system guarantees a form of absence of deadlock, which however presupposes that basic values do not appear in patterns. We thought it was important to allow basic values in patterns for expressiveness reasons (e.g., they are crucial in the encoding of the spi-calculus presented in Section 3.2).

Our type system is partially inspired by XML Schema [137], but is less rich than, say, the language of [45, 52, 48]. In particular, we have preferred to omit recursive types. While certainly useful in a full-blown language, recursion would raise technicalities that hinder issues concerning name and code mobility. Also, our pattern language is quite basic, partly for similar reasons of simplicity, partly because more sophisticated patterns can be easily encoded.

The calculus described so far enforces a strictly static typing discipline. We also consider an extension of this calculus with *dynamic abstractions* (Section 3.5), which are useful when little or nothing is known about the actual types of incoming documents. Run-time type checks ensure that substitutions arising from pattern matching respect the types statically assigned to variables. Run-time safety carries over. We shall argue that dynamic abstractions, combined with code mobility and subtyping, can provide linguistic support to such tasks as publishing and discovering services.

A *behavioural equivalence* based on barbed bisimulation [126] is introduced in Section 3.6. This equivalence takes into account both type information and the presence of an input interface. The underlying idea is that systems come equipped with an interface, i.e. a set of input channels at which services are offered; on these channels, external observers do not have input capability. The resulting equivalence can be used to validate interesting equations. Section 3.7 concludes this chapter.

3.2 Syntax and semantics of XPi

This section presents syntax and reduction semantics of XPi, and a few derived constructs.

3.2.1 Syntax

We assume a countable set of *variables* $\mathcal{V}$, ranged over $x, y, z, \ldots$, a set of *tags* $\mathcal{F}$, ranged over $\mathtt{f}, \mathtt{g}, \ldots$, and a set of *basic values* $\mathcal{BV}$ $v, w, \ldots$. We leave $\mathcal{BV}$ unspecified (it might contain such values as integers, strings, or Java objects), but assume that $\mathcal{BV}$ contains a countable set of *names* $\mathcal{N}$, ranged over $a, b, c, \ldots$. $\mathcal{N}$ is partitioned into a family of countable sets called *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. We let u range over $\mathcal{N} \cup \mathcal{V}$ and $\widetilde{x}, \widetilde{y}, \ldots$ denote tuples of variables.

Definition 3.1 (documents, patterns and processes). *The set $\mathcal{D}$ of XPi* documents M, N, *..., the set $\mathcal{Q}$ of XPi* patterns Q, Q', *... and the set $\mathcal{P}$ of XPi* processes P, $R, \ldots$ *are defined by the syntax in Table 3.1. In $Q_{\widetilde{x}}$, we impose the following linearity condition: $\widetilde{x}$ is a tuple of distinct variables and each $x_i \in \widetilde{x}$ occurs at most once in Q.*

In the style of XDuce and ℂDuce, XML documents are represented in XPi as tagged, ordered lists that can be arbitrarily nested; these are the documents being exchanged among processes. A document can be either a basic value, a variable, a tagged document, a list of documents, or an abstraction. The last mentioned takes the form $(Q_{\widetilde{x}})P$, where variables $\widetilde{x}$ are a subset of variables in Q representing formal parameters, to be replaced by actual parameters at run-time: variables in $\widetilde{x}$ are used for extracting values from matching documents. A pattern is simply an abstraction-free document. For the sake of simplicity, we have ignored tag-variables that could be easily accommodated. Also, note that patterns do not allow for direct decomposition of documents into sublists, akin to the pattern `p, p'` in XDuce, which can be easily encoded though, as we show later in this section.

Process syntax is a variation on the π-calculus. In particular, asynchronous (non blocking) output on a channel u is written $\overline{u}\langle M \rangle$, and u is said to occur in *output subject position*. Nondeterministic guarded summation $\sum_{i \in I} a_i.A_i$ waits for any document matching A_i's pattern at channel a_i, for some $i \in I$, consumes this document and continues as prescribed by A_i; names a_i are said to occur in *input subject position*. Note that the syntax forbids variables in input subject position, hence a received name cannot be used as an input channel; in other words, names are passed around with the output capability only. Parallel composition $P \,|\, R$ represents concurrent execution of P and R. Process P *else* R behaves like P, if P can do some internal reduction, otherwise reduces to R. This operator will be useful for coding up, e.g., $if - then - else$, without the burden of dealing with explicit negation on patterns. Replication $!P$ represents the parallel composition of arbitrarily many copies of P.

Document	$M ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid \mathtt{f}(M)$	*Tagged Document*
		$\mid LM$	*List*
		$\mid A$	*Abstraction*
List of documents	$LM ::=$	$[\,]$	*Empty list*
		$\mid x$	*Var*
		$\mid M, LM$	*Sequence*
Abstraction	$A ::=$	$(Q_{\widetilde{x}})P$	*Pattern and Continuation*
		$\mid x$	*Var*
Pattern	$Q ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid \mathtt{f}(Q)$	*Tagged Pattern*
		$\mid LQ$	*List*
List of patterns	$LQ ::=$	$[\,]$	*Empty list*
		$\mid x$	*Var*
		$\mid Q, LQ$	*Sequence*
Process	$P ::=$	$\overline{u}\langle M\rangle$	*Output*
		$\mid \sum_{i\in I} a_i.A_i$	*Input Guarded Summation*
		$\mid P \ else \ P$	*Else*
		$\mid P \mid P$	*Parallel Composition*
		$\mid !P$	*Replication*
		$\mid (\nu\, a)P$	*Restriction*

Table 3.1: Syntax of documents, patterns and processes.

Restriction $(\nu\, a)P$ creates a fresh name a, whose initial scope is P.

Binding conventions and notations. We stipulate that in every abstraction $(Q_{\widetilde{x}})P$ the variables in $\widetilde{x}$ bind with scope P, and that in each restriction $(\nu\, a)P$ name a binds with scope P. Accordingly, notions of alpha-equivalence ($=_\alpha$), free and bound names ($\mathrm{fn}(\cdot)$ and $\mathrm{bn}(\cdot)$), free and bound variables ($\mathrm{fv}(\cdot)$ and $\mathrm{bv}(\cdot)$) arise as expected for documents, patterns and processes. We assume that $=_\alpha$ is sort-respecting, in the

sense that a bound name can be alpha-renamed only to a name of the same sort. Whenever needed, we shall implicitly assume all binding occurrences of names (resp. variables) are distinct and disjoint from free names (resp. variables). Moreover, we identify processes up to alpha-equivalence.

The following abbreviations for documents and patterns are used: $[M_1, M_2, \ldots, M_{k-1}, M_k]$ stands for $M_1, (M_2, (\ldots (M_{k-1}, (M_k, [])) \ldots))$, while $\mathtt{f}[M_1, \ldots, M_k]$ stands for $\mathtt{f}([M_1, \ldots, M_k])$. The following abbreviations for processes are used: $\mathbf{0}$, $a_1.A_1$ and $a_1.A_1 + a_2.A_2 + \cdots + a_n.A_n$ stand for $\sum_{\{i \in I\}} a_i.A_i$ when $|I| = 0$, $|I| = 1$, and $|I| = n$, respectively; $(\nu\, a_1, \ldots, a_n)P = (\nu\, \widetilde{a})P$ stands for $(\nu\, a_1) \ldots (\nu\, a_n)P$; and a stands for $a.\mathbf{0}$. We sometimes save on subscripts by marking binding occurrences of variables in patterns by a "?" symbol, or by replacing a binding occurrence of a variable by a don't care symbol, "_", if that variable does not occur in the continuation process. E.g. $([\mathtt{f}[?x], \mathtt{g}[\_]])P$ stands for $([\mathtt{f}[x], \mathtt{g}[y]]_{\{x,y\}})P$ where $y \notin \mathrm{fv}(P)$.

Our list representation of XML ignores algebraic properties of concatenation (such as associativity, see [89]). We simply take for granted some translation from actual XML documents to our syntax. The following example illustrates informally what this translation might look like.

Example 3.2.1. An XML document encoding an address book (on the left) and its representation in XPi (on the right):

```
< addrbook >                                  addrbook[
  < person >                                    person[
    < name > John Smith < /name >                 name(John Smith),
    < tel > 12345 < /tel >                        tel(12345),
    < emailaddrs >                                emailaddrs[
      < email > john@smith < /email >               email(john@smith),
      < email > smith@john < /email >               email(smith@john)
    < /emailaddrs >                               ],
  < /person >                                   ],
  < person >                                    person[
    < name > Eric Brown < /name >                 name(Eric Brown),
    < tel > 678910 < /tel >                       tel(678910),
    < emailaddrs >< /emailaddrs >                 emailaddrs[]
  < /person >                                   ]
< /addrbook >                                 ]
```

Note that a sequence of tagged documents such as $< \mathtt{tag1} > M < /\mathtt{tag1} >< \mathtt{tag2} > N < /\mathtt{tag2} > \cdots$ is rendered as an ordered list $[\mathtt{tag1}(M), \mathtt{tag2}(N), \cdots]$. A pattern that extracts name and tele-

$P \mid R \equiv R \mid P$	$P \mid \mathbf{0} \equiv P$
$(P \mid R) \mid S \equiv P \mid (R \mid S)$	$!P \equiv P \mid !P$
$(\nu\, a)\mathbf{0} \equiv \mathbf{0}$	$(\nu\, a)(\nu\, b)P \equiv (\nu\, b)(\nu\, a)P$
$(\nu\, a)(P \mid R) \equiv P \mid (\nu\, a)R$ if $a \notin \mathrm{fn}(P)$	

Table 3.2: Structural congruence.

phone number of the first person of the address book above is: $Q = \texttt{addrbook}[\texttt{person}[\texttt{name}(?x), \texttt{tel}(?y), \_], \_]$.

Definition 3.2 (closed processes and documents). *We denote with $\mathcal{P}_{cl}$ (resp. $\mathcal{D}_{cl}$) the set containing all processes $P \in \mathcal{P}$ (resp. documents $M \in \mathcal{D}$) such that* $\mathrm{fv}(P) = \emptyset$ *(resp.* $\mathrm{fv}(M) = \emptyset$*), that is all closed processes (resp. documents).*

3.2.2 Reduction semantics

A *reduction relation* describes system evolution via internal communications. Following [109], XPi reduction semantics is based on *structural congruence* $\equiv$, which permits certain rearrangements of parallel composition, replication, and restriction.

Definition 3.3 (structural congruence). *The* structural congruence $\equiv$ *is the least congruence satisfying the rules in Table 3.2.*

The relation $\equiv$ extends to abstractions, hence to documents, in the expected manner. The reduction semantics also relies on a standard *matching predicate*, that matches a (linear) pattern against a closed document and yields a *substitution*.

Definition 3.4 (substitutions). Substitutions $\sigma, \sigma', \ldots$ *are finite partial maps from the set $\mathcal{V}$ of variables to the set $\mathcal{D}_{cl}$ of closed documents. We denote by ε the empty substitution. For any term t, $t\sigma$ denotes the result of applying σ onto t (with alpha-renaming of bound names and variables if needed.)*

In what follows, we denote by $\mathrm{dom}(\sigma)$ the domain of the substitution σ.

Definition 3.5 (match document-pattern). *Let M be a closed document and Q be a linear pattern:* $\mathrm{match}(M, Q, \sigma)$ *holds true if and only if* $\mathrm{dom}(\sigma) = \mathrm{fv}(Q)$ *and $Q\sigma = M$; in this case, we also say that M* matches *Q.*

We can now introduce the reduction relation.

Definition 3.6 (reduction). *The* reduction relation*, $\rightarrow \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$, is the least binary relation on closed processes satisfying the rules in Table 3.3.*

$$(\text{COM})\quad \frac{j \in I \quad a_j = a \quad A_j = (Q_{\widetilde{x}})P \quad \text{match}(M, Q, \sigma)}{\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i \to P\sigma}$$

$$(\text{STRUCT})\quad \frac{P \equiv R \quad R \to R' \quad R' \equiv P'}{P \to P'} \qquad (\text{CTX})\quad \frac{P \to P'}{(\nu\, \widetilde{a})(P \mid R) \to (\nu\, \widetilde{a})(P' \mid R)}$$

$$(\text{ELSE}_1)\quad \frac{P \to P'}{P \ \mathit{else}\ R \to P'} \qquad (\text{ELSE}_2)\quad \frac{P \not\to}{P \ \mathit{else}\ R \to R}$$

Table 3.3: Reduction semantics.

According to (COM), a communication between $\overline{a}\langle M\rangle$ and $\sum_{\{i\in I\}} a_i.A_i$ occurs provided that $a = a_j$ for some $j \in I$ and the match predicate between the pattern Q and M yields a substitution σ. The other rules account for structural congruence, else, parallel composition and restriction as expected. As usual, we denote by $\to^*$ the reflexive and transitive closure of $\to$.

Example 3.2.2. Consider the document M and the pattern Q defined in Example 3.2.1. $\text{match}(M, Q, \sigma)$ holds true, with $\text{dom}(\sigma) = \{x, y\}$, $\sigma(x) = \texttt{John Smith}$ and $\sigma(y) = \texttt{12345}$; according to (COM):

$$\overline{a}\langle M\rangle \mid a.(Q)(\overline{b}\langle[\texttt{n}(x), \texttt{t}(y)]\rangle \mid P) \to \overline{b}\langle[\texttt{n}(\texttt{John Smith}), \texttt{t}(\texttt{12345})]\,\rangle \mid P\sigma\ .$$

3.2.3 Derived constructs and examples

XPi allows for straightforward definition of a few powerful constructs, that will be used in later examples. In the following, we shall freely use recursive definitions of processes of the form $A(\widetilde{x}) \stackrel{\triangle}{=} P$ (with $\text{fv}(P) \subseteq \widetilde{x}$), that can be coded up using replication [109].

Application. A functional-like application for abstractions, $A \bullet M$, can be defined as $(\nu\, c)(\overline{c}\langle M\rangle \mid c.A)$, for any $c \notin \text{fn}(M, A)$.

Case. A pattern matching construct relying on a *first match* policy, written

$$\begin{array}{rlcl} \mathit{case}\ M\ \mathit{of} & (Q_1)_{\widetilde{x}_1} & \Rightarrow & P_1, \\ & (Q_2)_{\widetilde{x}_2} & \Rightarrow & P_2, \\ & & \vdots & \\ & (Q_k)_{\widetilde{x}_k} & \Rightarrow & P_k \end{array}$$

evolves into P_1 if M matches Q_1 (with substitutions involved), otherwise evolves into P_2 if M matches Q_2, and so on; if there is no match, the process is stuck. This construct can be defined in XPi as follows (assuming precedence of $\bullet$ on else and right-associativity for else):

$$(Q_1)_{\widetilde{x}_1} P_1 \bullet M \quad else \quad (Q_2)_{\widetilde{x}_2} P_2 \bullet M \quad else \quad \cdots \quad else \quad (Q_k)_{\widetilde{x}_k} P_k \bullet M \ .$$

Example 3.2.3. Consider the document M defined in Example 3.2.1. Suppose that we want to extract and send along b the name of all persons that have at least an email, and along c the name of all persons that do not have an email. Assume M is available on channel a. A process that performs this task is: $a.(\texttt{addrbook}[?x])R(x)$, where $R(x)$ is:

$$R(x) = case\ x\ of\ \texttt{person}[\texttt{name}(?y), \_, \texttt{emailaddrs}[\texttt{email}(\_), \_]], ?w \Rightarrow \overline{b}\langle y\rangle \mid R(w)$$
$$\texttt{person}[\texttt{name}(?z), \_], ?j \Rightarrow \overline{c}\langle z\rangle \mid R(j) \ .$$

Decomposition. A process that attempts to *decompose* a document M into two sublists that satisfy the patterns $Q_{\widetilde{x}}$ and $Q'_{\widetilde{y}}$ and proceeds like P (with substitutions for $\widetilde{x}$ and $\widetilde{y}$ involved), if possible, otherwise is stuck, written:

$$M\ as\ Q_{\widetilde{x}}, Q'_{\widetilde{y}}\ \Rightarrow P$$

can be defined as the recursive process $Dec([\,], M)$, where:

$$Dec(l, x) \stackrel{\triangle}{=} case\ x\ of\ ?\,y\,, ?\,w\ \Rightarrow \Big(case\ l@y\ of\ Q_{\widetilde{x}}\ \Rightarrow \big(case\ w\ of$$
$$Q'_{\widetilde{y}} \Rightarrow P,$$
$$\_ \Rightarrow Dec(l@y,\ w)\big),$$
$$\_\ \Rightarrow\ Dec(l@y, w)\Big) \ .$$

Here we have used a list-append function @, which can be easily defined via a call $Append(l_1, l_2, r)$ where l_1 and l_2 are two lists, r is the channel where the result of the append $l_1@l_2$ will be sent and $Append$ is the following recursive process:

$$Append(x, y, r) \stackrel{\triangle}{=} case\ x\ of\ ?w, [\,] \Rightarrow \overline{r}\langle [w, y]\rangle$$
$$?w_1, ?w_2 \Rightarrow (\nu\, r')(Append(w_2, y, r') \mid r'.(?z)\,\overline{r}\langle [w_1, z]\rangle) \ .$$

Example 3.2.4. Consider $M = [\mathtt{int(1)}, \mathtt{int(2)}, \mathtt{int(3)}, \mathtt{char(a)}, \mathtt{char(b)}, \mathtt{char(c)}]$ and the patterns $Q_{\{x\}} = ?x$ and $Q'_{\{y,w\}} = \mathtt{char}(?y), ?w$. Then

$$\begin{aligned} &\overline{a}\langle M\rangle \mid a.(?z)\ z\ as\ Q,\ Q' \Rightarrow \overline{b}\langle x\rangle \mid \overline{c}\langle[\mathtt{char}(y), w]\rangle \\ \longrightarrow^* \ &\overline{b}\langle[\mathtt{int(1)}, \mathtt{int(2)}, \mathtt{int(3)}]\rangle \mid \overline{c}\langle[\mathtt{char(a)}, \mathtt{char(b)}, \mathtt{char(c)}]\rangle\ . \end{aligned}$$

Map. A process that, from a list LM, generates another list containing all documents of the original list satisfying a certain (closed) pattern Q, assigns this list to a variable y and proceeds like P:

$$let\ \ y = map\ Q, LM\ \ in\ \ P$$

can be defined as the process $Map([\,], LM)$, where the following recursive definition is assumed:

$$\begin{aligned} Map(l, x) \stackrel{\triangle}{=}\ \ case\ x\ of\ \ ?z, ?w\ \ \Rightarrow (\ case\ z\ of\ \ Q\ \ &\Rightarrow Map(l@z, w), \\ \_\ \ &\Rightarrow Map(l, w)\ ), \\ \_\ \ \Rightarrow\ P[l/y]\ .& \end{aligned}$$

Example 3.2.5. Suppose the document M of Example 3.2.1 is available at a. Here is a process that consumes M, creates a list of all persons that have at least an email and sends this list along b:

$$a.(\mathtt{addrbook}[?x])(let\ \ y = map\ \ \mathtt{person}[\_, \_, \mathtt{emailaddrs}[\mathtt{email}(\_), \_]],\ x\ \ in\ \ \overline{b}\langle y\rangle)\ .$$

Path expressions. A process that evaluates the path expression /f, that is extracts all top-level elements tagged f from a document M is abbreviated as

$$let\ \ x = path\ \ /\mathtt{f},\ M\ \ in\ \ P$$

and can be defined as the process $Path([\,], M)$, where the following recursive definition is assumed:

$$\begin{aligned} Path(l, y) \stackrel{\triangle}{=} case\ \ y\ \ of\ \mathtt{f}[?w_1], ?w_2 &\Rightarrow Path(l@\mathtt{f}[w_1], w_2) \\ \_, ?w &\Rightarrow Path(l, w) \\ [\,] &\Rightarrow P[l/x]\ . \end{aligned}$$

More involved is the definition of a recursive process evaluating //f, which extracts all elements tagged f from a document M. We abbreviate with

$$let \;\; x = path \;\; //\mathtt{f},\, M \;\; in \;\; P$$

the process $DeepPath([\,], M)$ below:

$$\begin{aligned} DeepPath(l, y) \stackrel{\triangle}{=}\; & (\nu\, r)(r(x).P \mid DP'([\,], y, r)) \\ DP'(l, y, r) \stackrel{\triangle}{=} case \;\; y \;\; of\; & \mathtt{f}[?w_1], ?w_2 \Rightarrow (\nu\, r_1,\, r_2)\big(\, DP'([\,], w_1, r_1) \mid DP'([\,], w_2, r_2) \\ & \qquad \mid r_1(x_1).r_2(x_2).\overline{r}\langle[\mathtt{f}[w_1]]@x_1@x_2\rangle\big) \\ & \_[?w_1], ?w_2 \Rightarrow (\nu\, r_1,\, r_2)\big(\, DP'([\,], w_1, r_1) \mid DP'([\,], w_2, r_2) \\ & \qquad \mid r_1(x_1).r_2(x_2).\overline{r}\langle x_1@x_2\rangle\big) \\ & [\,] \Rightarrow \overline{r}\langle l\rangle \;. \end{aligned}$$

Obviously the previous processes can be composed for evaluating expressions like //f/g. Other common path expressions can be easily coded up in this style. We shall not pursue this direction any further.

Example 3.2.6 (a web service). Consider a web service *WS* that offers two different services: an audio streaming service, offered at channel *stream*, and a download service, offered at channel *download*. Clients that request the first kind of service must specify a streaming channel and its bandwidth (high or low), so that *WS* can stream one of two audio files ($\mathtt{v}_{\mathtt{low}}$ or $\mathtt{v}_{\mathtt{high}}$), as appropriate. Clients that request to download must specify a channel at which the player will be received. A client can run the downloaded player locally, supplying it appropriate parameters (a local streaming channel and its bandwidth). We represent streaming on a channel simply as an output action along that channel:

$$\begin{aligned} WS \stackrel{\triangle}{=} !(\quad & stream.(\mathtt{req\_stream}[\mathtt{bandwidth}(\mathtt{low}), \mathtt{channel}(?x)])\, \overline{x}\langle \mathtt{v}_{\mathtt{low}}\rangle \\ + \; & stream.(\mathtt{req\_stream}[\mathtt{bandwidth}(\mathtt{high}), \mathtt{channel}(?y)])\, \overline{y}\langle \mathtt{v}_{\mathtt{high}}\rangle \\ + \; & download.(\mathtt{req\_down}(?z))\overline{z}\langle Player\rangle\,) \;. \end{aligned}$$

Player is an abstraction:

$$\begin{aligned} Player \stackrel{\triangle}{=} (\mathtt{req\_stream}[\mathtt{bandwidth}(?y), \mathtt{channel}(?z)])(\; case \;\; y \;\; of \;\; & \mathtt{low} \Rightarrow \overline{z}\langle \mathtt{v}_{\mathtt{low}}\rangle \\ & \mathtt{high} \Rightarrow \overline{z}\langle \mathtt{v}_{\mathtt{high}}\rangle\,) \;. \end{aligned}$$

Note that the first two addends of *WS* are equivalent to *stream.Player*. However, the extended form written above makes it possible a static optimization of channels (see Example 3.3.3).

A client that asks for low bandwidth streaming, listens at s and then proceeds like C is:

$$C_1 \triangleq (\nu\, s)(\overline{stream}\langle \texttt{req\_stream}[\texttt{bandwidth}(\texttt{low}), \texttt{channel}(s)]\rangle \mid s.(?v)C) \ .$$

Another client that asks for download, then runs the player locally, listening at a local high bandwidth channel s is defined as:

$$C_2 \triangleq (\nu\, d,\, s)\big(\overline{download}\langle \texttt{req\_down}(d)\rangle \mid d.(?x)(x\bullet \\ \texttt{req\_stream}[\texttt{bandwidth}(\texttt{high}), \texttt{channel}(s)] \mid s.(?v)C)\big) \ .$$

3.2.4 Encryption and decryption

Cryptographic primitives are sometimes used in distributed applications to guarantee secrecy and authentication of transmitted data. As a testbed for expressiveness, we show how to encode shared-key encryption and decryption primitives à la spi-calculus [3] into XPi. In Example 3.5.2, we shall see an example of application of these encodings. We first introduce XPi$^{\text{cr}}$, a cryptographic extension of XPi that subsumes shared-key spi-calculus, and then show how to encode XPi$^{\text{cr}}$ into XPi. Document syntax is extended with the following clause, that represents encryption of M using N as a key:

$$M \quad ::= \quad \cdots \mid \{M\}_N \qquad \textit{Encryption}$$

where N does contain neither abstractions nor encryptions. Process syntax is extended with a case operator, that attempts decryption of M using N as a key and if successful binds the result to a variable x :

$$P \quad ::= \quad \cdots \mid \mathit{case}\ M\ \mathit{of}\ \{x\}_N\ \ \mathit{in}\ \ P \qquad \textit{Decryption}$$

where N does contain neither abstractions nor encryptions, M is a variable or a document of the form $\{M'\}_{N'}$ and x binds in P. Patterns remain unchanged, in particular they may not contain encryptions or abstractions. The additional reduction rule is:

$$(\textsc{dec}) \quad \mathit{case}\ \{M\}_N\ \mathit{of}\ \{x\}_N\ \ \mathit{in}\ \ P \to P[M/x] \ .$$

Next, two translation functions, one for documents ($[\![\cdot]\!]$) and one for processes ($(\!|\cdot|\!)$), are defined from XPi$^{\text{cr}}$ to XPi. The translations of document follow a familiar continuation-passing style (see Table 3.4).

Following [126], let us define the barb predicate $P \Downarrow a$ as follows: there is P' s.t. $P \to^* P'$ and P' has either an input addend $a.A$ or an output $\overline{a}\langle M\rangle$, which are not

$$
\begin{array}{rr}
[\![u]\!] = u & [\![\mathtt{f}(M)]\!] = \mathtt{f}([\![M]\!]) \\
[\![\{M\}_N]\!] = ([N, ?x])\,\overline{x}\langle [\![M]\!]\rangle & [\![(Q_{\tilde{x}})P]\!] = (Q_{\tilde{x}})(\!|P|\!) \\
\multicolumn{2}{c}{[\![M, LM]\!] \;=\; [\![M]\!], [\![LM]\!]}
\end{array}
$$

$$
\begin{array}{rr}
(\!|\overline{u}\langle M\rangle|\!) = \overline{u}\langle [\![M]\!]\rangle & (\!|P_1 \mid P_2|\!) = (\!|P_1|\!) \mid (\!|P_2|\!) \\
(\!|\textstyle\sum_{i\in I} a_i.A_i|\!) = \sum_{i\in I} a_i.[\![A_i]\!] & (\!|!P|\!) = !(\!|P|\!) \\
(\!|P \;\mathit{else}\; R|\!) = (\!|P|\!) \;\mathit{else}\; (\!|R|\!) & (\!|(\nu\, a)P|\!) = (\nu\, a)(\!|P|\!) \\
\multicolumn{2}{c}{(\!|\,\mathit{case}\; M \;\mathit{of}\; \{x\}_N \;\; \mathit{in} \;\; P|\!) \;=\; (\nu\, r)\,([\![M]\!] \bullet [N, r] \mid r.(?x)\,(\!|P|\!))}
\end{array}
$$

Table 3.4: Translating functions from XPi$^{\mathrm{cr}}$ to XPi.

in the scope of a $(\nu\, a)$, an else or guarded summation. The encoding defined above is correct, in the sense that it preserves reductions and barbs in both directions, as stated by the proposition below. Note that, by compositionality, this implies the encoding is fully abstract w.r.t. barbed equivalence (see e.g. [28]).

Proposition 3.1. *Let P be a closed process in XPi$^{\mathrm{cr}}$.*

(1) *if $P \to P'$ then $(\!|P|\!) \to^* (\!|P'|\!)$;*

(2) *if $(\!|P|\!) \to P'$ then $\exists P'' \in$ XPi$^{\mathrm{cr}}$ s.t. $P' \to^* (\!|P''|\!)$;*

(3) *$P \Downarrow a$ if and only if $(\!|P|\!) \Downarrow a$.*

Proof:

(1) The proof is straightforward by induction on the derivation of $P \to P'$. We consider the last reduction rule applied; the most interesting case is rule (DEC), in the other cases $(\!|P|\!)$ reduces in one step into $(\!|P'|\!)$. $\mathit{case}\ \{M\}_N\ \mathit{of}\ \{x\}_N\ \mathit{in}\ P \to P[M/x]$ and

$$
\begin{array}{ll}
(\!|\,\mathit{case}\ \{M\}_N\ \mathit{of}\ \{x\}_N \;\; \mathit{in} \;\; P|\!) & \\
= (\nu\, r)\big([\![\{M\}_N]\!] \bullet [N, r] \mid r.(?x)\,(\!|P|\!)\big) & \text{(by def. of } (\!|\cdot|\!),\ r \text{ fresh)} \\
= (\nu\, r)\big(([N, ?y])\overline{y}\langle [\![M]\!]\rangle \bullet [N, r] \mid r.(?x)\,(\!|P|\!)\big) & \text{(by def. of } [\![\cdot]\!],\ y \text{ fresh)} \\
= (\nu\, r)\big((\nu\, c)(c.([N, ?y])\overline{y}\langle [\![M]\!]\rangle \mid \overline{c}\langle [N, r]\rangle) \mid r.(?x)\,(\!|P|\!)\big) & \text{(by def. of } \bullet,\ c \text{ fresh)} \\
\to (\nu\, r)\big(\overline{r}\langle [\![M]\!]\rangle \mid r.(?x)\,(\!|P|\!)\big) & \text{(by (COM))} \\
\to (\!|P|\!)[[\![M]\!]/x] & \text{(by (COM))} \\
= (\!|P[M/x]|\!) & \text{(by def. of } (\!|\cdot|\!)\text{)}\ .
\end{array}
$$

(2) The proof is straightforward by induction on $(\!|P|\!) \to P'$, we proceed by distinguishing the various cases by looking at the structure of P. The most interesting is the case $P = \mathit{case}\ \{M\}_N\ \mathit{of}\ \{x\}_{N'}\ \mathit{in}\ R$. Recall that P is closed, hence M,

N and N' are closed documents.

$$\langle\!|P|\!\rangle \;=\; (\nu\, r)\big((\nu\, c)(c.([N, ?y])\overline{y}\langle[\![M]\!]\rangle \mid \overline{c}\langle[N', r]\rangle) \mid r.(?x)\,\langle\!|R|\!\rangle\big) \to P'.$$

A communication on c takes place, thus rules (CTX) and (COM) are applied. This means that $\mathrm{match}([N', r], [N, y], [r/y])$, $N = N'$ and $P' = (\nu\, r)\big(\overline{r}\langle[\![M]\!]\rangle \mid r.(?x)\,\langle\!|R|\!\rangle\big)$. Moreover, $\mathrm{match}([\![M]\!], x, [[\![M]\!]/x])$ and by rule (COM), $P' \to \langle\!|R|\!\rangle[[\![M]\!]/x]$. By rule (DEC), $\mathit{case}\ \{M\}_N\ \mathit{of}\ \{x\}_N\ \mathit{in}\ R \to R[M/x] = P''$, and $\langle\!|P''|\!\rangle = \langle\!|R|\!\rangle[[\![M]\!]/x]$ by definition of $\langle\!|\cdot|\!\rangle$.

(3) The result follows from (1) and (2).

□

3.3 A type system

In this section, we define a type system for XPi that disciplines messaging at the level of channels, patterns and processes. The system guarantees that well-typed processes respect channels capacities at run-time. In other words, services are guaranteed to receive only requests they can understand, and conversely, services offered at a given channel are consistent with the type declared for that channel. Concerning the structure of messages, XPi's type system draws its inspiration from, but is less rich than, XML-Schema [137]. Our system permits to specify types for basic values (such as string or int) and provides tuple types (fixed-length lists) and star types (arbitrary-length lists). Moreover, it provides abstraction types for channel and code mobility. For the sake of simplicity, we have omitted attributes and recursive types.

3.3.1 Document types

We assume an unspecified set of *basic types* $\mathcal{BT}$, ranged over by bt, bt', ..., that might include int, string, boolean, or even Java classes. We assume that $\mathcal{BT}$ contains a countable set of *sort names* in one-to-one correspondence with the sorts $\mathcal{S}, \mathcal{S}', \ldots$ of $\mathcal{N}$; by slight abuse of notation, we denote sort names by the corresponding sorts.

Definition 3.7 (types). *The set $\mathcal{T}$ of* types*, ranged over by T, S, ..., is defined by the syntax in Table 3.5.*

Note the presence of the union type $\mathsf{T} + \mathsf{S}$, that is the type of all documents of type T or S, and of the star type $*\mathsf{T}$, that is the type of all (possibly empty) lists of elements of type T. (T)Abs is the type of all abstractions that can consume documents of type T. Finally, note the presence of $\mathbf{T}$ and $\mathbf{\bot}$ types. $\mathbf{T}$ is simply the type of all

Type	$\mathsf{T} ::=$	bt	*Basic type*
		$\mid \mathbf{T}$	*Top*
		$\mid \mathbf{\perp}$	*Bottom*
		$\mid \mathtt{f}(\mathsf{T})$	*Tagged Type*
		$\mid \mathsf{LT}$	*List*
		$\mid \mathsf{T} + \mathsf{T}$	*Union*
		$\mid (\mathsf{T})\mathsf{Abs}$	*Abstraction*
List type	$\mathsf{LT} ::=$	$[\,]$	*Empty*
		$\mid *\,\mathsf{T}$	*Star*
		$\mid \mathsf{T}, \mathsf{LT}$	*Sequence*

Table 3.5: Syntax of types.

documents. On the contrary, no document has type $\mathbf{\perp}$, but this type is extremely useful for the purpose of defining channel types, as we shall see below.

Notations. The following abbreviations for types are used: $[\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_{k-1}, \mathsf{T}_k]$ stands for $\mathsf{T}_1, (\mathsf{T}_2, (\ldots (\mathsf{T}_{k-1}, (\mathsf{T}_k, [\,]))\ldots))$, while $\mathtt{f}[\mathsf{T}_1, \ldots, \mathsf{T}_k]$ stands for $\mathtt{f}([\mathsf{T}_1, \ldots, \mathsf{T}_k])$.

Example 3.3.1. A type for address books (see document M in Example 3.2.1) can be the following:

$$\begin{aligned}\mathtt{addrbook}[*\mathtt{person}[&\mathtt{name}(\mathtt{string}),\\ &\mathtt{tel}(\mathtt{int}),\\ &\mathtt{emailaddrs}(*\mathtt{email}(\mathtt{string}))]]\ .\end{aligned}$$

Next, we associate types with channels, or more precisely with sorts. This is done by introducing a capacity function.

Definition 3.8 (capacity function). *A* capacity *function is a surjective map from the set of sorts to the set of types.*

In the sequel, we fix a generic capacity function. We shall denote by $ch(\mathsf{T})$ a generic sort $\mathcal{S}$ that is mapped to T. The meaning of this being that channels of sort $ch(\mathsf{T})$ can only carry documents of type T. Note that, by surjectivity of the capacity function, for each type T there is a sort $ch(\mathsf{T})$. In particular, $ch(\mathbf{T})$ is the sort of channels that can transport anything. In practice, determining capacity T of a given channel a, i.e.

	(Sub-Sort) $\dfrac{\mathsf{T} < \mathsf{S}}{ch(\mathsf{S}) < ch(\mathsf{T})}$		
(Sub-Top)	$\dfrac{}{\mathsf{T} < \mathbf{T}}$	(Sub-Bottom)	$\dfrac{}{\mathbf{\bot} < \mathsf{T}}$
(Sub-Basic)	$\dfrac{\mathsf{bt}_1 \prec \mathsf{bt}_2}{\mathsf{bt}_1 < \mathsf{bt}_2}$	(Sub-Tag)	$\dfrac{\mathsf{S} < \mathsf{T}}{\mathtt{f}(\mathsf{S}) < \mathtt{f}(\mathsf{T})}$
(Sub-Star$_1$)	$\dfrac{}{[\,] < *\mathsf{T}}$	(Sub-Star$_2$)	$\dfrac{\mathsf{S} < \mathsf{T} \quad \mathsf{LT} < *\mathsf{T}}{\mathsf{S}, \mathsf{LT} < *\mathsf{T}}$
(Sub-Star$_3$)	$\dfrac{\mathsf{S} < \mathsf{T}}{*\mathsf{S} < *\mathsf{T}}$	(Sub-List)	$\dfrac{\mathsf{T} < \mathsf{S} \quad \mathsf{LT} < \mathsf{LS}}{\mathsf{T}, \mathsf{LT} < \mathsf{S}, \mathsf{LS}}$
(Sub-Union$_1$)	$\dfrac{\mathsf{T} < \mathsf{S} \text{ or } \mathsf{T} < \mathsf{S}'}{\mathsf{T} < \mathsf{S} + \mathsf{S}'}$	(Sub-Union$_2$)	$\dfrac{\mathsf{S} < \mathsf{T} \quad \mathsf{S}' < \mathsf{T}}{\mathsf{S} + \mathsf{S}' < \mathsf{T}}$

Table 3.6: Subtyping relation.

that a belongs to $ch(\mathsf{T})$, might be implemented with a variety of mechanisms, such as attaching to a an explicit reference to T's definition. We abstract away from these details.

3.3.2 Subtyping relation

List and star types and the presence of $\mathbf{T}$ and $\mathbf{\bot}$ naturally induce a subtyping relation. For example, a service capable of processing documents of type $\mathsf{T} = \mathtt{f}(* \, \mathsf{int})$ must be capable of processing documents of type $\mathsf{T}' = \mathtt{f}[\mathsf{int}, \mathsf{int}]$, i.e. T' is a subtype of T. Subtyping also serves to lift a generic subtyping preorder on basic types, $\prec$, to all types.

Definition 3.9 (subtyping). *The* subtyping relation $< \subseteq \mathcal{T} \times \mathcal{T}$ *is the least reflexive and transitive relation closed under the rules of Table 3.6.*

Note that we disallow subtyping on abstractions. The reason for this limitation will be discussed shortly after presenting the type checking system (see Remark 3.1). Also note that subtyping is contravariant on sorts capacities (rule (Sub-Sort)): this is natural if one thinks of a name of capacity T as, roughly, a function that can take arguments of type T. As a consequence of contravariance, for any T, we have $ch(\mathsf{T}) < ch(\mathbf{\bot})$, that is, $ch(\mathbf{\bot})$ is the type of all channels.

3.3.3 Type checking

A *basic typing* relation $v : \mathsf{bt}$ on basic values and basic types is presupposed, which is required to respect subtyping, i.e. whenever $\mathsf{bt} \prec \mathsf{bt}'$ and $v : \mathsf{bt}$ then $v : \mathsf{bt}'$. We demand that for each bt there is at least one $v : \mathsf{bt}$. Moreover, we require that for each v the set of bt's such that $v : \mathsf{bt}$ has a minimal element bt' such that $\mathsf{bt}' \prec \mathsf{bt}$ for each bt in the set. On names and sort names the basic typing relation is the following:

$$a : \mathcal{S} \text{ if and only if } a \in \mathcal{S}' \text{ for some } \mathcal{S}' < \mathcal{S} \ .$$

Contexts $\Gamma, \Gamma', \ldots$ are finite partial maps from variables $\mathcal{V}$ to types $\mathcal{T}$, sometimes denoted as sets of variable bindings $\{x_i : \mathsf{T}_i\}_{i \in I}$ with x_i distinct from x_j for each i different from j.

Notations. We denote the empty context by $\emptyset$. We denote by $\Gamma_{-\widetilde{x}}$ the context obtained from Γ by removing the bindings for the variables in $\widetilde{x}$, and by $\Gamma_{|\widetilde{x}}$ the context obtained by restricting Γ to the bindings for the variables in $\widetilde{x}$. The subtyping relation is extended to contexts by letting $\Gamma_1 < \Gamma_2$ iff $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ and $\forall x \in \mathrm{dom}(\Gamma_1)$ it holds that $\Gamma_1(x) < \Gamma_2(x)$. Union of contexts Γ_1 and Γ_2 having disjoint domains is written as $\Gamma_1 \cup \Gamma_2$ or as Γ_1, Γ_2 if no ambiguity arises. Sum of contexts Γ_1 and Γ_2 is written as $\Gamma_1 + \Gamma_2$ and is defined as $(\Gamma_1 + \Gamma_2)(x) = \Gamma_1(x) + \Gamma_2(x)$ if $x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$, otherwise $(\Gamma_1 + \Gamma_2)(x) = \Gamma_i(x)$ if $x \in \mathrm{dom}(\Gamma_i)$ for $i = 1, 2$.

Type checking relies on a type-pattern matching predicate, $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$, whose role is twofold: (1) it extracts from T the types expected for variables in Q after matching against documents of type T, yielding the context Γ, (2) it checks that Q is consistent with type T, i.e. that the type of Q is a subtype of T under Γ.

Definition 3.10 (type-pattern match). *The predicate* $\mathrm{tpm}(\mathcal{T}, Q, \Gamma)$ *is defined by the rules in Table 3.7.*

The matching predicate between types and patterns is univocal:

Lemma 3.1. *For every* $\mathcal{T}$, Q, Γ *and* Γ' *if* $\mathrm{tpm}(\mathcal{T}, Q, \Gamma)$ *and* $\mathrm{tpm}(\mathcal{T}, Q, \Gamma')$ *then* $\Gamma = \Gamma'$.

Proof: The proof is straightforward by induction on the derivation of $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$. The base cases are (tpm-Empty), (tpm-Value) and (tpm-Star$_1$), where $\Gamma = \Gamma' = \emptyset$, (tpm-Var), where $\Gamma = \Gamma' = \{x : \mathsf{T}\}$ and (tpm-Top), where for each $x \in \mathrm{fv}(Q)$ we have $\Gamma(x) = \Gamma'(x) = \mathbf{T}$. The other cases can be proved by applying the inductive hypothesis (recall the linearity of Q). □

As expected, type checking works on an *annotated syntax*, where each $Q_{\widetilde{x}}$ is decorated by a context Γ for its binding variables $\widetilde{x}$, written $Q_{\widetilde{x}} : \Gamma$, with $\widetilde{x} = \mathrm{dom}(\Gamma)$, or

$$(\text{TPM-TOP}) \quad \frac{Q \neq x}{\text{tpm}(\mathbf{T}, Q, \Gamma)},\ \forall x \in \text{fv}(Q) :\ \Gamma(x) = \mathbf{T}$$

$$(\text{TPM-EMPTY}) \quad \frac{}{\text{tpm}([\,],[\,],\emptyset)} \qquad (\text{TPM-VAR}) \quad \frac{}{\text{tpm}(\mathsf{T}, x, \{x : \mathsf{T}\})}$$

$$(\text{TPM-VALUE}) \quad \frac{v : \mathsf{bt}}{\text{tpm}(\mathsf{bt}, v, \emptyset)} \qquad (\text{TPM-TAG}) \quad \frac{\text{tpm}(\mathsf{T}, Q, \Gamma)}{\text{tpm}(\mathtt{f}(\mathsf{T}), \mathtt{f}(Q), \Gamma)}$$

$$(\text{TPM-STAR}_1) \quad \frac{}{\text{tpm}(*\mathsf{T}, [\,], \emptyset)} \qquad (\text{TPM-STAR}_2) \quad \frac{\text{tpm}(\mathsf{T}, Q, \Gamma_1)\ \ \text{tpm}(*\mathsf{T}, LQ, \Gamma_2)}{\text{tpm}(*\mathsf{T}, (Q, LQ), \Gamma_1 \cup \Gamma_2)}$$

$$(\text{TPM-LIST}) \quad \frac{\text{tpm}(\mathsf{T}, Q, \Gamma_1)\ \ \text{tpm}(\mathsf{LT}, LQ, \Gamma_2)}{\text{tpm}((\mathsf{T}, \mathsf{LT}), (Q, LQ), \Gamma_1 \cup \Gamma_2)}$$

$$(\text{TPM-UNION}) \quad \frac{\text{tpm}(\mathsf{T}_0, Q, \Gamma_0)\ \ \text{or}\ \ \text{tpm}(\mathsf{T}_1, Q, \Gamma_1)}{\text{tpm}(\mathsf{T}_0 + \mathsf{T}_1, Q, \Gamma)} \quad \text{where:}$$

$$\Gamma = \begin{cases} \Gamma_0 + \Gamma_1 \text{ if } \text{tpm}(\mathsf{T}_0, Q, \Gamma_0) \text{ and } \text{tpm}(\mathsf{T}_1, Q, \Gamma_1) \\ \Gamma_i \qquad \text{if } \text{tpm}(\mathsf{T}_i, Q, \Gamma_i) \text{ and for no } \Gamma' \ \text{tpm}(\mathsf{T}_{i+1 \bmod 2}, Q, \Gamma'), \text{ for } i = 0 \text{ or } i = 1 \end{cases}$$

Table 3.7: Matching between types and patterns.

simply $Q : \Gamma$, where it is understood that the binding variables of Q are $\text{dom}(\Gamma)$. For notational simplicity, we shall use such abbreviations as $a.(\mathtt{f}[?x : \mathsf{T}, ?y : \mathsf{S}])P$ instead of $a.(\mathtt{f}[x, y] : \{x : \mathsf{T}, y : \mathsf{S}\})P$, and assume don't care variables "_" are always annotated with $\mathbf{T}$. Reduction semantics carries over to annotated closed processes formally unchanged.

Notations. We say that a type T is *abstraction-free* if T contains no subterms of the form $(\mathsf{S})\mathsf{Abs}$. A context Γ is abstraction-free if for each $x \in \text{dom}(\Gamma)$, $\Gamma(x)$ is abstraction-free. We use $\Gamma \vdash u \in ch(\mathsf{T})$ as an abbreviation for: either $u = a \in ch(\mathsf{T})$ or $u = x \in \mathcal{V}$ and $\Gamma(x) = ch(\mathsf{T})$.

The type system, defined on open terms, consists of two sets of inference rules, one for documents and one for processes, displayed in Table 3.8 and 3.9, respectively. These two systems are mutually dependent, since abstractions may contain processes, and processes may contain abstractions. Note that the system is entirely syntax driven, i.e. the process P (resp. the pair (M, T)) determines the rule that should be applied to check $\Gamma \vdash P : \mathsf{ok}$ (resp. $\Gamma \vdash M : \mathsf{T}$), where ok is the type associated to well-typed processes.

The most interesting of the typing rules for documents is (TM-ABS). Informally, $\Gamma \vdash A : (\mathsf{T})\mathsf{Abs}$ ensures that under Γ the following is true: (1) abstraction $A =$

$$\text{(Tm-Empty)}\ \frac{}{\Gamma \vdash\ [\,] : [\,]} \qquad \text{(Tm-Top)}\ \frac{}{\Gamma \vdash\ M : \mathbf{T}}$$

$$\text{(Tm-Value)}\ \frac{v : \mathsf{bt}}{\Gamma \vdash\ v : \mathsf{bt}} \qquad \text{(Tm-Var)}\ \frac{\Gamma(x) < \mathsf{T}}{\Gamma \vdash\ x : \mathsf{T}}$$

$$\text{(Tm-Tag)}\ \frac{\Gamma \vdash\ M : \mathsf{T}}{\Gamma \vdash \mathtt{f}(M) : \mathtt{f}(\mathsf{T})} \qquad \text{(Tm-List)}\ \frac{\Gamma \vdash M : \mathsf{T} \quad \Gamma \vdash LM : \mathsf{LT}}{\Gamma \vdash\ M, LM\ :\ \mathsf{T}, \mathsf{LT}}$$

$$\text{(Tm-Star}_1\text{)}\ \frac{}{\Gamma \vdash\ [\,] : *\mathsf{T}} \qquad \text{(Tm-Star}_2\text{)}\ \frac{\Gamma \vdash M : \mathsf{T} \quad \Gamma \vdash LM : *\mathsf{T}}{\Gamma \vdash\ M, LM\ :\ *\mathsf{T}}$$

$$\text{(Tm-Union)}\ \frac{\Gamma \vdash M : \mathsf{T} \quad \text{or} \quad \Gamma \vdash M : \mathsf{S}}{\Gamma \vdash M : \mathsf{T} + \mathsf{S}}$$

$$\text{(Tm-Abs)}\ \frac{\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1) \quad (\Gamma_1)_{|\widetilde{x}} < \Gamma_Q \quad (\Gamma_1)_{|\widetilde{y}} > \Gamma_{|\widetilde{y}} \quad \Gamma, \Gamma_Q \vdash P : \mathsf{ok}}{\Gamma \vdash (Q : \Gamma_Q)P : (\mathsf{T})\mathsf{Abs}}$$

where $\widetilde{x} = \mathrm{dom}(\Gamma_Q)$, $\widetilde{y} = \mathrm{fv}(Q) \setminus \widetilde{x}$ and $(\Gamma_1)_{|\widetilde{y}}$ is abstraction-free

Table 3.8: Type system for documents.

$$\text{(T-In)}\ \frac{a \in ch(\mathsf{T}) \quad \Gamma \vdash A : (\mathsf{T})\mathsf{Abs}}{\Gamma \vdash a.A : \mathsf{ok}}$$

$$\text{(T-Out)}\ \frac{\Gamma \vdash u \in ch(\mathsf{T}) \quad \Gamma \vdash M : \mathsf{T}}{\Gamma \vdash \overline{u}\langle M \rangle : \mathsf{ok}} \qquad \text{(T-Sum)}\ \frac{\forall i \in I \quad \Gamma \vdash a_i.A_i : \mathsf{ok} \quad |I| \neq 1}{\Gamma \vdash \sum_{i \in I} a_i.A_i : \mathsf{ok}}$$

$$\text{(T-Rep)}\ \frac{\Gamma \vdash P : \mathsf{ok}}{\Gamma \vdash\ !P : \mathsf{ok}} \qquad \text{(T-Par)}\ \frac{\Gamma \vdash P : \mathsf{ok} \quad \Gamma \vdash R : \mathsf{ok}}{\Gamma \vdash\ (P \,|\, R) : \mathsf{ok}}$$

$$\text{(T-Res)}\ \frac{\Gamma \vdash P : \mathsf{ok}}{\Gamma \vdash\ (\nu\, a)P : \mathsf{ok}} \qquad \text{(T-Else)}\ \frac{\Gamma \vdash P : \mathsf{ok} \quad \Gamma \vdash R : \mathsf{ok}}{\Gamma \vdash\ P\ else\ R : \mathsf{ok}}$$

Table 3.9: Type system for processes.

$(Q_{\widetilde{x}} : \Gamma_Q)P$ behaves safely upon consuming documents of type T (because the type at which the actual parameters will be received is a subtype of the type declared for formal parameters, $(\Gamma_1)_{|\widetilde{x}} < \Gamma_Q$, and because of $\Gamma, \Gamma_Q \vdash P : \mathsf{ok}$); (2) the pattern Q is consistent with type T, i.e. essentially, the run-time type of Q is a subtype of T (because of type-pattern match and of $\Gamma_{|\widetilde{y}} < (\Gamma_1)_{|\widetilde{y}}$). This guarantees existence of a document of type T that matches the pattern, that is, patterns with no chance of being matched are forbidden. Moreover, no ill-formed pattern will arise from Q thanks to the abstraction-freeness of $(\Gamma_1)_{|\widetilde{y}}$. Of course, $\Gamma \vdash A : (\mathsf{T})\mathsf{Abs}$ does not guarantee that any documents of type T can be consumed, as this depends on pattern matching, hence on the value associated at run-time to the free variables in A's pattern.

Rule (T-In) checks that an abstraction A residing at channel $a \in ch(\mathsf{T})$ can safely

consume documents of type T, and that there do exist documents of type T that match the pattern of A. Conversely (T-Out) checks that documents sent at u be of type T. Input and summation (rule (T-Sum)) are dealt with separately only for notational convenience. Finally, it is worth to notice that, by definition of $a : \mathcal{S}$, rule (Tm-Value) entails subsumption on channels (i.e. $\Gamma \vdash a : \mathcal{S}$ and $\mathcal{S} < \mathcal{S}'$ implies $\Gamma \vdash a : \mathcal{S}'$). The remaining rules should be self-explanatory.

In the sequel, for closed annotated processes P, we shall write $P : \mathsf{ok}$ for $\emptyset \vdash P : \mathsf{ok}$, and say that P is well-typed. Similarly we write $M : \mathsf{T}$ for closed annotated documents M.

Example 3.3.2. Assume $a \in ch(*\mathsf{int})$ and $b \in ch(\mathtt{f}[\mathsf{int}, *\mathsf{int}])$. Then $P : \mathsf{ok}$, where:

$$P = a.(?y : *\mathsf{int})b.(\mathtt{f}[?x : \mathsf{int}, y])\overline{a}\langle x, y\rangle \mid \overline{a}\langle[4, 5]\rangle \mid \overline{a}\langle[4, 5, 6]\rangle \ .$$

Note that, if we change the sort of b into $ch(\mathtt{f}[\mathsf{int}, [\mathsf{int}, \mathsf{int}]])$, then P is not well-typed, as rule (Tm-Abs) fails on $A = (\mathtt{f}[?x : \mathsf{int}, y])\overline{a}\langle x, y\rangle$. This is intuitively correct, because a possible run-time type of A is $(\mathtt{f}[\mathsf{int}, [\mathsf{int}, \mathsf{int}, \mathsf{int}]])\mathsf{Abs}$, which is not consistent with the capacity associated to b, that is $\mathtt{f}[\mathsf{int}, [\mathsf{int}, \mathsf{int}]]$.

To illustrate the use of $ch(\mathbf{T})$ and $ch(\mathbf{\bot})$, and contravariance on sort names, consider a "link process" ([28]) that constantly receives any *name* on a and sends it along b. This can be written as $!a.(?x : ch(\mathbf{\bot}))\overline{b}\langle x\rangle$. This process is well-typed provided $a \in ch(ch(\mathsf{T}))$, for some T, and that $b \in ch(ch(\mathbf{\bot}))$.

Remark 3.1 (on abstractions and subtyping). To see why we disallow subtyping on abstractions, consider the types $\mathsf{T} = [\mathtt{f}(\mathsf{int}), \mathtt{f}(\mathsf{int})]$ and $*\mathtt{f}(\mathsf{int}) = \mathsf{S}$. Clearly $\mathsf{T} < \mathsf{S}$. Assume we had defined subtyping *covariant* on abstractions, so that $(\mathsf{T})\mathsf{Abs} < (\mathsf{S})\mathsf{Abs}$. Now, clearly $A = (?x : \mathsf{T})\mathbf{0} : (\mathsf{T})\mathsf{Abs}$, but *not* $A : (\mathsf{S})\mathsf{Abs}$ (the condition $(\Gamma_1)_{|\widetilde{x}} < \Gamma_Q$ of (Tm-Abs) fails). In other words, a crucial subtyping property would be violated.

On the other hand, assume we had defined subtyping *contravariant* on abstractions, so that $(\mathsf{S})\mathsf{Abs} < (\mathsf{T})\mathsf{Abs}$. Consider $A' = (Q : \Gamma_Q)\mathbf{0}$, where $Q : \Gamma_Q = [\mathtt{f}(?x : \mathsf{int}), \mathtt{f}(?y : \mathsf{int}), \mathtt{f}(?z : \mathsf{int})]$; clearly $A' : (\mathsf{S})\mathsf{Abs}$, but *not* $A' : (\mathsf{T})\mathsf{Abs}$ (simply because there is no type-pattern match between T and Q.) This would violate again the subtyping property.

3.3.4 Typing rules for Application and Case

Previously we presented the typing rules for the standard syntax of the calculus, but also the processes that use derived constructs and recursive processes are subject to type checking. We can do this by traducing the constructs in standard syntax and

using the basic rules. To facilitate type checking we define some rules that we can directly apply on derived constructs.

Application. In the following, we let $\mathsf{T}_{M,\Gamma}$ denote the *exact type* of M under Γ, obtained from M by replacing each x by $\Gamma(x)$, each name $a \in ch(\mathsf{T})$ by $ch(\mathsf{T})$, each other v by the least type bt s.t. $v : \mathsf{bt}$, and, recursively, each abstraction subterm $(Q : \Gamma_Q)P$ by $(\mathsf{T}_{Q,\,\Gamma\cup\Gamma_Q})\mathsf{Abs}$. The rule for application is derived by rules (T-Par), (T-Out) and (T-In) and is the following:

$$(\text{T-Appl}) \quad \frac{\Gamma \vdash A : (\mathsf{T}_{M,\Gamma})\mathsf{Abs}}{\Gamma \vdash A \bullet M : \mathsf{ok}}$$

that is easily proved sound recalling that $A \bullet M = (\nu\, c)(c.A \,|\, \overline{c}\langle M\rangle)$ (c fresh) and assuming that c is chosen s.t. $c \in ch(\mathsf{T}_{M,\Gamma})$.

Case. Concerning *case*, first note that the typed version of this construct contemplates annotated patterns, thus:

$$\begin{aligned} case\ M\ of \quad & Q_1 : \Gamma_{Q_1} \ \Rightarrow\ P_1, \\ & Q_2 : \Gamma_{Q_2} \ \Rightarrow\ P_2, \\ & \qquad \vdots \\ & Q_k : \Gamma_{Q_k} \ \Rightarrow\ P_k\ . \end{aligned}$$

Then, relying on the rule for application, the typing rule can be written as:

$$(\text{T-Case}) \quad \frac{\forall i = 1, \ldots, k: \ \ \Gamma \vdash (Q_i : \Gamma_{Q_i})P_i \bullet M : \mathsf{ok}}{\Gamma \vdash case\ M\ of\ \ Q_1 : \Gamma_{Q_1} \Rightarrow P_1, \ldots, Q_k : \Gamma_{Q_k} \Rightarrow P_k : \mathsf{ok}}\ .$$

Example 3.3.3 (a web service, continued). Consider the service defined in Example 3.2.6. Assume a basic type $\mathsf{mp3}$ of all mp3 files, such that $\mathtt{v_{low}}$, $\mathtt{v_{high}} : \mathsf{mp3}$, and a basic type $\mathsf{l\text{-}mp3}$ of low quality mp3 files, s.t. $\mathtt{v_{low}} : \mathsf{l\text{-}mp3}$, but *not* $\mathtt{v_{high}} : \mathsf{l\text{-}mp3}$. Assume $\mathsf{l\text{-}mp3} < \mathsf{mp3}$; note that this implies that $ch(\mathsf{mp3}) < ch(\mathsf{l\text{-}mp3})$, i.e. if a channel can be used for streaming generic files, it can also be used for streaming low-quality files, which fits intuition. Let T be $\mathtt{req\_stream}[\mathtt{bandwidth}(\mathsf{string}), \mathtt{channel}(ch(\mathsf{mp3}))]$ and fix the following capacities for channels *stream* and *download*: $stream \in ch(\mathsf{T})$ and $download \in ch(\mathtt{req\_down}(ch((\mathsf{T})\mathsf{Abs})))$. An annotated version of *WS*, which permits in principle a static optimization of channels (assuming allocation of low-quality

channels is less expensive than generic channels'):

$$\begin{aligned} WS = !\big(\ & \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\texttt{low}), \texttt{channel}(?x : \mathit{ch}(\mathsf{l\text{-}mp3}))])\overline{x}\langle \texttt{v}_{\texttt{low}} \rangle \\ & + \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\texttt{high}), \texttt{channel}(?y : \mathit{ch}(\mathsf{mp3}))])\overline{y}\langle \texttt{v}_{\texttt{high}} \rangle \\ & + \mathit{download}.(\texttt{req\_down}[?z : \mathit{ch}((\mathsf{T})\mathsf{Abs})])\overline{z}\langle \mathit{Player} \rangle\big) \end{aligned}$$

where *Player* is the obvious annotated version of the player of Example 3.2.6. It is easy to check that $\mathit{Player} : (\mathsf{T})\mathsf{Abs}$ and that $WS : \mathsf{ok}$.

3.4 Properties of typing

A type system aims at providing static guarantees that certain properties hold at run-time. The safety property of our interest can be defined in terms of channel capacities, document types and consistency. First, a formal definition of consistent types and patterns.

Definition 3.11 (T-consistency). *A type T is* consistent *if $\mathbf{\bot}$ does not occur in T. A pattern Q is* T-consistent *if there is a document $M : \mathsf{T}$ that matches Q.*

Note that all sort names, including $\mathit{ch}(\mathbf{\bot})$, are consistent types by definition. A safe process is one whose output and input actions are in agreement with channel capacities, as stated by the definition below. Of course, for input actions it makes sense to require consistency (condition (2)) only if the input channel has in turn a consistent capacity.

Definition 3.12 (safety). *Let P be an annotated closed process. P is* safe *if and only if for each name $a \in \mathit{ch}(\mathsf{T})$:*

(1) *whenever $P \equiv (\nu\, \widetilde{h})(\overline{a}\langle M \rangle \mid R)$ then $M : \mathsf{T}$;*

(2) *suppose T is consistent. Whenever $P \equiv (\nu\, \widetilde{h})(S \mid R)$, where S is a guarded summation, $a.A$ an addend of S and Q is A's pattern, then Q is T-consistent.*

A first expected result about the type system is type safety which relies on the following lemma.

Lemma 3.2. *Suppose T is consistent. If* $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$ *for some Γ then Q is T-consistent.*

Proof: The proof is straightforward by induction on the derivation of $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$. The most interesting cases are (tpm-Top) and (tpm-Var):

(tpm-Top): consider the document M obtained by replacing every variable in Q with a value; by (Tm-Top) $M : \mathbf{T}$ and obviously M matches Q, thus Q is $\mathbf{T}$-consistent;

(TPM-VAR): $\mathrm{tpm}(\mathsf{T}, x, \{x : \mathsf{T}\})$ implies that $Q = x$ and Q is T-consistent because we can prove that, for every consistent type T, there is a document $M : \mathsf{T}$ (the proof is immediate assuming that for each basic type bt there is at least one $v : \mathsf{bt}$);

(TPM-EMPTY), (TPM-STAR$_1$): it is enough to choose $M = []$;

(TPM-VALUE): given that for each basic type bt there is at least one $v : \mathsf{bt}$ it is enough to choose $M = v$;

(TPM-TAG), (TPM-STAR$_2$), (TPM-LIST) and (TPM-UNION): the proof proceeds by applying the inductive hypothesis.

□

Theorem 3.1 (type safety). *Let P be an annotated closed process and suppose $P : \mathsf{ok}$, then P is safe.*

PROOF: The proof is straightforward by induction on the derivation of $P : \mathsf{ok}$, by distinguishing the last typing rule applied. Case (T-IN) relies on Lemma 3.2. □

Subject reduction relies on the following lemmata. The first states that typing does respect the subtyping relation:

Proposition 3.2 (subtyping). *If $\mathsf{S} < \mathsf{T}$ then for any document M such that $\Gamma \vdash M : \mathsf{S}$ we have $\Gamma \vdash M : \mathsf{T}$.*

PROOF: We distinguish two cases:

$M = x$**:** $\Gamma \vdash M = x : \mathsf{S}$ implies, by rule (TM-VAR), $\Gamma(x) < \mathsf{S}$; but $\mathsf{S} < \mathsf{T}$, therefore, by the transitivity of $<$, $\Gamma(x) < \mathsf{T}$ and, by rule (TM-VAR), $\Gamma \vdash M = x : \mathsf{T}$.

$M \neq x$**:** The proof is straightforward by induction on the sum of the depths of the derivations of $\mathsf{S} < \mathsf{T}$ and $\Gamma \vdash M : \mathsf{S}$ and proceeds by distinguishing the last subtyping rule applied. The base cases are (SUB-BASIC), which relies on the fact that subtyping on basic values entails subtyping, (SUB-SORT), which relies on definition of $a : \mathcal{S}$, (SUB-TOP) and (SUB-STAR$_1$). The other cases can be proved by applying the inductive hypothesis.

□

The following lemma ensures, roughly, that type-pattern match agrees with document-pattern match. In particular, if a closed document of type T matches a pattern Q, then the values taken on by Q's variables after matching will be of the type predicted by tpm. Moreover, the lemma states that type-pattern match is preserved by type-respecting substitutions.

Proposition 3.3 (matching). *Let M be a closed document.*

(1) *If $M : \mathsf{T}$ and* $\mathrm{match}(M, Q, \sigma)$ *then* $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$ *and $\sigma(x) : \Gamma(x)$ for each x in* $\mathrm{dom}(\sigma)$.

(2) *If* M *is abstraction-free and* $M : \mathsf{T}$ *then* $\mathrm{tpm}(\mathsf{T}, M, \emptyset)$.

(3) *If* M *is abstraction free,* $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$ *and* $M : \Gamma(x)$ *then* $\mathrm{tpm}(\mathsf{T}, Q[{}^{M}\!/_{x}], \Gamma_{-\{x\}})$.

PROOF:

(1) We distinguish two cases:

$Q = x$**:** $\sigma = [{}^{M}\!/_{x}]$ and, by (TPM-VAR), $\mathrm{tpm}(\mathsf{T}, x, \{x : \mathsf{T}\})$;

$Q \neq x$**:** the proof is straightforward by induction on the derivation of $M : \mathsf{T}$. Note that it can be neither the case $M = x$, because M is closed, nor $M = A$, because $\mathrm{match}(M, Q, \sigma)$ with $Q \neq x, A$.

(TM-EMPTY): $M = [\,] : [\,] = \mathsf{T}$ and $\mathrm{match}(M, Q, \sigma)$ implies $Q = [\,]$ and $\sigma = \epsilon$; moreover, by (TPM-EMPTY), $\mathrm{tpm}([\,], [\,], \emptyset)$;

(TM-STAR$_1$): the proof proceeds as in the previous case;

(TM-TOP): $\mathrm{tpm}(\mathbf{T}, Q, \Gamma)$ with $\Gamma(x) : \mathbf{T}$ for each $x \in \mathrm{fv}(Q)$. By (TM-TOP), each document is of type $\mathbf{T}$, hence $\sigma(x) : \mathbf{T} = \Gamma(x)$ for each $x \in \mathrm{fv}(Q) = \mathrm{dom}(\sigma)$;

(TM-VALUE): $v : \mathsf{bt}$ and $\mathrm{match}(v, Q, \sigma)$, with $Q \neq x$, implies $Q = v$ and $\sigma = \epsilon$. Moreover, by (TPM-VALUE), $\mathrm{tpm}(\mathsf{bt}, v, \emptyset)$;

(TM-TAG), (TM-STAR$_2$), (TM-LIST), (TM-UNION): the proof relies on linearity of patterns and proceeds by applying the inductive hypothesis.

(2) The proof is straightforward by induction on the derivation of $M : \mathsf{T}$.

(3) The proof is straightforward by induction on the derivation of $\mathrm{tpm}(\mathsf{T}, Q, \Gamma)$, the base case (TPM-VAR) relies on Proposition 3.3 (2).

□

The next three propositions ensure that typing is preserved by substitution, weakening and structural congruence.

Proposition 3.4 (substitution). *Let* M *be a closed document and* $M : \mathsf{T}$. *If* $\Gamma,\ x : \mathsf{T} \vdash P : \mathsf{ok}$ *then* $\Gamma \vdash P[{}^{M}\!/_{x}] : \mathsf{ok}$. *If* $\Gamma,\ x : \mathsf{T} \vdash N : \mathsf{S}$ *then* $\Gamma \vdash N[{}^{M}\!/_{x}] : \mathsf{S}$.

PROOF: The proof proceeds by mutual induction on the derivation of $\Gamma,\ x : \mathsf{T} \vdash P : \mathsf{ok}$ and $\Gamma,\ x : \mathsf{T} \vdash N : \mathsf{S}$. We distinguish the last typing rule applied. The most interesting cases are:

(T-OUT): $\Gamma,\ x : \mathsf{T} \vdash \overline{u}\langle M' \rangle : \mathsf{ok}$ implies $\Gamma, x : \mathsf{T} \vdash u \in ch(\mathsf{S})$ and $\Gamma, x : \mathsf{T} \vdash M' : \mathsf{S}$. By induction $\Gamma \vdash M'[{}^{M}\!/_{x}] : \mathsf{S}$. If $u \neq x$ then $\Gamma \vdash u[{}^{M}\!/_{x}] = u \in ch(\mathsf{S})$. If $u = x$ then $\mathsf{T} = ch(\mathsf{S})$ and $M = a$ for some $a \in \mathcal{N}$. $a : \mathsf{T}$ implies $a \in ch(\mathsf{S}') < ch(\mathsf{S})$, for a S' such that $\mathsf{S} < \mathsf{S}'$. By Proposition 3.2 (subtyping), $\Gamma \vdash M'[{}^{M}\!/_{x}] : \mathsf{S}$ and $\mathsf{S} < \mathsf{S}'$ imply $\Gamma \vdash M'[{}^{M}\!/_{x}] : \mathsf{S}'$. In both cases, by rule (T-OUT), $\Gamma \vdash (\overline{u}\langle M' \rangle)[{}^{M}\!/_{x}] : \mathsf{ok}$.

(TM-VAR): $\Gamma, x : \mathsf{T} \vdash y : \mathsf{S}$; the proof proceeds by observing that, in case $x = y$, it holds that $\mathsf{T} < \mathsf{S}$ and, by applying Proposition 3.2 (subtyping), $M : \mathsf{S}$.

(TM-ABS): $\Gamma, x : \mathsf{T} \vdash (Q_{\widetilde{x}} : \Gamma_Q)P : (\mathsf{S})\mathsf{Abs}$ implies $\Gamma, \Gamma_Q, x : \mathsf{T} \vdash P : \mathsf{ok}$ and, by induction, $\Gamma, \Gamma_Q \vdash P[M/x] : \mathsf{ok}$.

If $x \notin \mathrm{fv}(Q)$ then $Q[M/x] = Q$. Otherwise, by (TM-ABS), $(\Gamma_1)_{|\widetilde{y}} > (\Gamma, x : \mathsf{T})_{|\widetilde{y}}$. Hence, from $M : \mathsf{T}$ and Proposition 3.2 (subtyping), it follows that $M : \Gamma_1(x)$. Therefore, by Proposition 3.3 (3) (matching), we have $\mathrm{tpm}(\mathsf{S}, Q[M/x], (\Gamma_1)_{-\{x\}})$. Finally, by rule (TM-ABS), $\Gamma \vdash ((Q[M/x])_{\widetilde{x}} : \Gamma_Q)P[M/x] = ((Q_{\widetilde{x}} : \Gamma_Q)P)[M/x] : (\mathsf{S})\mathsf{Abs}$.

The other cases can be proved by applying the inductive hypothesis. □

Proposition 3.5 (weakening). *Let P be an annotated process. If $\Gamma, x : \mathsf{T} \vdash P : \mathsf{ok}$ and $x \notin \mathrm{fv}(P)$ then $\Gamma \vdash P : \mathsf{ok}$.*

PROOF: The proof is straightforward by induction on the derivation of $\Gamma, x : \mathsf{T} \vdash P : \mathsf{ok}$. □

Proposition 3.6 (subject congruence). *Let P and R be annotated processes. If $\Gamma \vdash P : \mathsf{ok}$ and $P \equiv R$ then $\Gamma \vdash R : \mathsf{ok}$.*

PROOF: The proof is straightforward by induction on the derivation of $P \equiv R$, and in case $(\nu\, a)(P \mid R) \equiv P \mid (\nu\, a)R$, with $a \notin \mathrm{fn}(P)$, relies on Proposition 3.5. □

Theorem 3.2 (subject reduction). *Let P be an annotated closed process. If $P : \mathsf{ok}$ and $P \to P'$ then $P' : \mathsf{ok}$.*

PROOF: By induction on the derivation of $P \to P'$. We distinguish the last reduction rule applied; the most interesting case is (COM):

(COM): $\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i \to P\sigma$ where, for some $j \in I$: $a = a_j$, $A_j = (Q_{\widetilde{x}} : \Gamma_Q)P$ and $\mathrm{match}(M, Q, \sigma)$. We have to prove that $P\sigma : \mathsf{ok}$; recall that P is closed, hence M is closed.

From $\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i : ok$ and rule (T-PAR): $\overline{a}\langle M\rangle : \mathsf{ok}$ and $\sum_{i\in I} a_i.A_i : \mathsf{ok}$. Hence, for some T it holds that $a \in ch(\mathsf{T})$, $M : \mathsf{T}$ and $A_j : (\mathsf{T})\mathsf{Abs}$, by (T-OUT), (T-SUM) and (T-IN).

From $A_j : (\mathsf{T})\mathsf{Abs}$ and rule (TM-ABS), we deduce that $\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1)$, $(\Gamma_1)_{|\widetilde{x}} < \Gamma_Q$ and $\Gamma_Q \vdash P : \mathsf{ok}$.

By Proposition 3.3 (1) (matching), $M : \mathsf{T}$, $\mathrm{match}(M, Q, \sigma)$ and $\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1)$ imply that $\forall x \in \mathrm{dom}(\sigma)$ we have $\sigma(x) : \Gamma_1(x)$ and, by Proposition 3.2 (subtyping), $\sigma(x) : \Gamma_Q(x)$. Moreover, M closed implies that $\sigma(x)$ is closed for each $x \in \mathrm{dom}(\sigma)$. In conclusion, by $\Gamma_Q \vdash P : \mathsf{ok}$ and Proposition 3.4 (substitution), $P\sigma : \mathsf{ok}$;

(STRUCT): $P \to P'$ implies $P \equiv R$ and $R \to R'$ with $R' \equiv P'$. By Proposition 3.6 (subject congruence), $P : \mathsf{ok}$ implies $R : \mathsf{ok}$ and, by inductive hypothesis, $R' : \mathsf{ok}$. Hence, again by Proposition 3.6 (subject congruence), $P' : \mathsf{ok}$;

(CTX): $(\nu\, \widetilde{a})(P|R) \to (\nu\, \widetilde{a})(P'|R)$ implies $P \to P'$. By (T-RES) and (T-PAR), $(\nu\, \widetilde{a})(P|R) : \mathsf{ok}$ implies $P : \mathsf{ok}$ and $R : \mathsf{ok}$. By inductive hypothesis, $P' : \mathsf{ok}$ and, again by (T-RES) and (T-PAR), $(\nu\, \widetilde{a})(P'|R) : \mathsf{ok}$;

(ELSE$_1$): $P \ else \ R \to P'$ implies $P \to P'$. By (T-ELSE), $P \ else \ R : \mathsf{ok}$ implies $P : \mathsf{ok}$ and by inductive hypothesis $P' : \mathsf{ok}$;

(ELSE$_2$): $P \ else \ R \to R$ and, by (T-ELSE), $P \ else \ R : \mathsf{ok}$ implies $R : \mathsf{ok}$.

□

As a consequence of subject reduction and type safety we get run-time safety.

Corollary 3.1 (run-time safety). *Let P be a closed annotated process. If $P : ok$ and $P \to^* P'$ then P' is safe.*

PROOF: By Theorem 3.1 (type safety) and 3.2 (subject reduction). □

3.5 An extension: dynamic abstractions

Although satisfactory in most situations, a static typing scenario does not seem appropriate in those cases where little is known in advance on actual types of data that will be received from the network.

Example 3.5.1 (a directory of services). Suppose one has to program an on-line directory of (references to) services. Upon request of a service of type T, for *any* T, the directory should look-up its catalog and respond by sending a channel of type $ch(\mathsf{T})$ along a reply channel. If the reply channel is fixed statically, it must be given capacity $ch(\mathbb{1})$, that is, any channel. Then, a client that receives a name at this channel must have some mechanism to cast at run-time this generic type to the subtype $ch(\mathsf{T})$, which means going beyond static typing. If the reply channel is provided by clients the situation does not get any better. E.g. consider the following service (here we use some syntactic sugar for the sake of readability):

$$!\, request.(\mathtt{req}[?t : \mathsf{Td}, ?x : ch(\mathsf{Tr})])\ let\ \ y = lookup(t)\ \ in\ \ \overline{x}\langle y\rangle$$

where *lookup* is a function from some type Td of type-descriptors to the type of *all* channels, $ch(\mathbb{1})$. It is not clear what capacity Tr the return channel variable x should be assigned. The only choice that makes the above process well typed is to set $\mathsf{Tr} = ch(\mathbb{1})$, that is, x can transport any channel. But then, a client's call to this

service like $\overline{request}\langle \mathtt{req}[v_{td}, r]\rangle$, where r has capacity $ch(\mathsf{T})$, is not well typed (because $r \in ch(ch(\mathsf{T}))$ and $ch(ch(\mathsf{T}))$ is not a subtype of $ch(\mathsf{Tr}) = ch(ch(\mathbf{1}))$).

Even ignoring the static vs. dynamic issue, the schemes sketched above would imply some form encoding of types and subtyping into XML, which is undesirable if one wishes to reason at an abstract level. As we shall see below, dynamic abstractions can solve these difficulties.

The scenario illustrated in the above example motivates the extension of the calculus presented in the preceding sections with a form of dynamic abstraction. The main difference from ordinary abstractions is that type checking for pattern variables is moved to run-time. This is reflected into an additional communication rule, that explicitly invokes type checking. We describe below the necessary extensions to syntax and semantics. We extend the syntactic category of Abstractions thus:

$$A \ ::= \ \cdots \mid (\!|Q_{\widetilde{x}} : \Gamma|\!)P \quad \textit{Dynamic abstraction}$$

with $\widetilde{x} = \mathrm{dom}(\Gamma)$. We let D range over dynamic abstractions and A over all abstractions. We add a new reduction rule:

$$(\textsc{com-d})\ \frac{j \in I \ \ a_j = a \ \ A_j = (\!|Q_{\widetilde{x}} : \Gamma|\!)P \ \ \mathrm{match}(M, Q, \sigma) \ \ \forall y \in \mathrm{dom}(\sigma) : \ \sigma(y) : \Gamma(y)}{\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i \to P\sigma}.$$

We finally add a new type checking rule. For this, we need the following additional notations. Given Γ_1 and Γ_2, we write $\Gamma_1 \lesseqgtr \Gamma_2$ if $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ and $\forall x \in \mathrm{dom}(\Gamma_1)$ there is a consistent type T such that $\mathsf{T} < \Gamma_1(x)$ and $\mathsf{T} < \Gamma_2(x)$.

$$(\textsc{Tm-Abs-d})\quad \frac{\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1) \quad (\Gamma_1)_{|\widetilde{x}} \lesseqgtr \Gamma_Q \quad (\Gamma_1)_{|\widetilde{y}} > \Gamma_{|\widetilde{y}} \quad \Gamma, \Gamma_Q \vdash P : \mathsf{ok}}{\Gamma \vdash (\!|Q_{\widetilde{x}} : \Gamma_Q|\!)P : (\mathsf{T})\mathsf{Abs}}$$

where $\widetilde{y} = \mathrm{fv}(Q) \setminus \widetilde{x}$ and $(\Gamma_1)_{|\widetilde{y}}$ is abstraction free. The existence of a common consistent subtype for Γ_Q and $(\Gamma_1)_{|\widetilde{x}}$ ensures a form of dynamic consistency for Q, detailed below.

We discuss now the extension of run-time safety. The safety property needs to be extended to inputs formed with dynamic abstractions. A stronger form of pattern consistency is needed.

Definition 3.13 (dynamic T-consistency). *An annotated pattern* $Q : \Gamma$ ($\mathrm{fv}(Q) = \mathrm{dom}(\Gamma)$) *is* dynamically T-consistent *if there is a document* $M : T$ *such that* $\mathrm{match}(M, Q, \sigma)$ *and* $\forall x \in \mathrm{dom}(\sigma)$ *we have* $\sigma(x) : \Gamma(x)$.

Definition 3.14 (dynamic safety). *Let P be an annotated closed process. P is* dynamically safe *if for each name $a \in ch(\mathsf{T})$ conditions (1) and (2) of Definition 3.12 hold, and moreover the following condition is true. Suppose T is consistent. Whenever $P \equiv (\nu\, \widetilde{h})(S \mid R)$, where S is a guarded summation, $a.D$ is an addend of S and $Q : \Gamma$ is D's annotated pattern, then $Q : \Gamma$ is dynamically T-consistent.*

It is straightforward to prove the extensions of Theorem 3.1 (type safety) and Corollary 3.1 (run-time safety) to the dynamic case, i.e.: every closed annotated well-typed process P is dynamically safe, and dynamic safety is preserved by reduction.

Proofs of Propositions 3.2 (subtyping), 3.3 (matching) and 3.4 (substitution) carry over essentially unchanged to the language with dynamic abstractions.

Theorem 3.3 (extension of Theorem 3.1). *Let P be an annotated closed process and suppose $P : \mathsf{ok}$, then P is dynamically safe.*

Proof: By induction on the derivation on $P : \mathsf{ok}$. The unique change is in rule (T-In) when $P = a.D : \mathsf{ok}$. By (T-In), $a \in ch(\mathsf{T})$ and $D : (\mathsf{T})\mathsf{Abs}$.

By rule (Tm-Abs-d): $D = (\!|Q_{\widetilde{x}} : \Gamma_Q|\!)P$; $\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1)$; and $(\Gamma_1)_{|\widetilde{x}} \lesseqgtr \Gamma_Q$. Hence, for each y in $\widetilde{x}$ there exists a consistent type S_y s.t. $\mathsf{S}_y < \Gamma_Q(y)$ and $\mathsf{S}_y < \Gamma_1(y)$.

Consider the document M obtained by replacing in Q every variable $y \in \widetilde{x}$ by some document $M' : \mathsf{S}_y$ (M' exists because S_y is consistent). Obviously, for each y in $\widetilde{x}$, $\mathrm{match}(M, Q, \sigma)$ and $\sigma(y) = M' : \mathsf{S}_y$ with $\mathsf{S}_y < \Gamma_Q(y)$. Therefore Q is dynamically T-consistent and, by Definition 3.14, the process $a.D$ is dynamically safe. □

Theorem 3.4 (extension of Theorem 3.2). *Let P be an annotated closed process. If $P : \mathsf{ok}$ and $P \to P'$ then $P' : \mathsf{ok}$.*

Proof: The proof is straightforward by induction on the derivation of $P \to P'$. The interesting case is (com-d), the other cases are unchanged.

By rule (com-d), $\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i \to P\sigma$ where, for some $j \in I$: $a = a_j$, $A_j = (\!|Q_{\widetilde{x}} : \Gamma_Q|\!)P$, $\mathrm{match}(M, Q, \sigma)$ and for every $y \in \mathrm{dom}(\sigma)$ it holds that $\sigma(y) : \Gamma_Q(y)$.

We have to prove that $P\sigma : \mathsf{ok}$. From $\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i : \mathsf{ok}$, we deduce that $\sum_{i\in I} a_i.A_i : \mathsf{ok}$. Hence, there is a $\mathsf{T} \in \mathcal{T}$ such that $A_j : (\mathsf{T})\mathsf{Abs}$ and $\Gamma_Q \vdash P : \mathsf{ok}$, (Tm-Abs-d). Finally, $\sigma(y) : \Gamma_Q(y)$ for each $y \in \mathrm{dom}(\sigma)$ and Proposition 3.4 (substitution) imply $P\sigma : \mathsf{ok}$. □

Corollary 3.2 (dynamic run-time safety). *Let P be an annotated closed process. If $P : \mathsf{ok}$ and $P \to^* P'$ then P' is dynamically safe.*

Proof: By Theorem 3.3 (extension of type safety) and 3.4 (extension of subject reduction). □

Example 3.5.2 (a directory of services, continued). Consider again the directory of services. Clients can either request a (reference to a) service of a given type, by sending a message to channel *discovery*, or request the directory to update its catalog with a new service, using the channel *publish*. Each request to *discovery* should contain some type information, which should allow the directory service to select a (reference to a) service of that type, taking subtyping into account. Types cannot be passed around explicitly. However one can pass a dynamic abstraction that can do the selection on behalf of the client and return the result back to the client at a private channel. The catalog is maintained on a channel *cat* local to the directory. Thus the directory process can be defined as follows, where $\prod_{i\in I}! \overline{cat}\langle c_i\rangle$ stands for $! \overline{cat}\langle c_1\rangle \mid \cdots \mid ! \overline{cat}\langle c_n\rangle$ (for $I = 1, \ldots, n$) and the following capacities are assumed: $discovery \in ch((ch(\mathbf{1}))\mathsf{Abs})$, $publish, cat \in ch(ch(\mathbf{1}))$.

$$\begin{array}{rcll} Dir & \triangleq & (\nu\, cat)(\prod_{i\in I}! \overline{cat}\langle c_i\rangle \mid & ! \, publish.(?y : ch(\mathbf{1}))! \, \overline{cat}\langle y\rangle \\ & & \mid & ! \, discovery.(?x : (ch(\mathbf{1}))\mathsf{Abs})\; cat.x) \ . \end{array}$$

Note that $(ch(\mathbf{1}))\mathsf{Abs}$ is the type of all abstractions that can consume some channel. A client that wants to publish a new service S that accepts documents of some type T at a new channel $a \in ch(\mathsf{T})$ is:

$$C_1 \triangleq (\nu\, a)(\, \overline{publish}\langle a\rangle \mid S\,) \ .$$

A client that wants to retrieve a reference to a service of type T, or any subtype of it, is:

$$C_2 \triangleq (\nu\, r)(\, \overline{discovery}\langle (\!| ?z : ch(\mathsf{T}) |\!) \overline{r}\langle z\rangle\rangle \mid r.(?y : ch(\mathsf{T}))\, C'\,) \ .$$

Suppose $r \in ch(ch(\mathsf{T}))$. Assuming S and C' are well typed (the latter under $\{y : ch(\mathsf{T})\}$), it is easily checked that the global system

$$P \triangleq Dir \mid C_1 \mid C_2$$

is well typed too.

In reality, the above solution would run into security problems, as the directory executes blindly any abstraction received from clients ($cat.x$). Moreover, services originating from unauthorized clients should not be published. We can avoid these problems using encryption so to authenticate both abstractions and published services. We rely on the encoding of encryption primitives described in Section 3.2. For the purpose of the present example, we extend the encoding to the typed calculus by $[\![\{M\}_k]\!] \triangleq ([k, ?x : ch(\mathbf{T})])\overline{x}\langle [\![M]\!]\rangle$, and $(\!| \, case\ M\ of\ \{x : \mathsf{T}\}_k\ \ in\ \ P \,|\!) \triangleq (\nu\, r)\,([\![M]\!] \bullet [k, r] \mid r.(\!| ?x : \mathsf{T} |\!)\, (\!| P |\!))$, with $r \in ch(\mathbf{T})$.

Assume that every client C_j shares a secret key k_j with the directory. A table associating client identifiers and keys is maintained on a channel *table* local to the directory (hence secure). Assume that identifiers id_j are of a basic type $\mathsf{identifier}$, that keys k_j are names of a sort Key and let $\mathsf{enc}(\mathsf{T})$ be the type of documents $\{M\}_k$ where $M : \mathsf{T}$. Fix the following capacities: $cat \in ch(ch(\mathbf{1}))$, $table \in ch([\mathtt{id}(\mathsf{identifier}), \mathtt{key}(\mathsf{Key})])$, $publish \in ch(\mathtt{service\_p}[\mathtt{id}(\mathsf{identifier}), \mathtt{channel}(\mathsf{enc}(ch(\mathbf{1})))])$, and $discovery \in ch(\mathtt{service\_d}[\mathtt{id}(\mathsf{identifier}), \mathtt{abstr}(\mathsf{enc}((ch(\mathbf{1}))\mathsf{Abs}))])$. The process Dir_s is:

$$\begin{aligned} Dir_s \triangleq (\nu\, cat,\ table)\Big(& \textstyle\prod_{i\in I}! \,\overline{cat}\langle c_i\rangle \mid \prod_{j\in J}! \,\overline{table}\langle[\mathtt{id}(id_j), \mathtt{key}(k_j)]\rangle \\ & \mid ! \,publish.(\mathtt{service\_p}[\mathtt{id}(?x : \mathsf{identifier}), \mathtt{channel}(?z_c : \mathsf{enc}(ch(\mathbf{1})))]) \\ & \ table.([\mathtt{id}(x), \mathtt{key}(?x_k : \mathsf{Key})])\ case\ z_c\ of\ \{y : ch(\mathbf{1})\}_{x_k}\ \ in\ !\overline{cat}\langle y\rangle \\ & \mid ! \,discovery.(\mathtt{service\_d}[\mathtt{id}(?x : \mathsf{identifier}), \mathtt{abstr}(?z_a : \mathsf{enc}((\mathsf{ch}(\mathbf{1}))\mathsf{Abs}))]) \\ & \ table.([\mathtt{id}(x), \mathtt{key}(?x_k : \mathsf{Key})])\, case\ z_a\ of\ \{y : (\mathsf{ch}(\mathbf{1}))\mathsf{Abs}\}_{x_k}\ \ in\ \ cat.y\Big)\ . \end{aligned}$$

The client C_1 may be rewritten as:

$$C_1' \triangleq (\nu\, a)(\ \overline{publish}\langle\mathtt{service\_p}[\mathtt{id}(id_1), \mathtt{channel}(\{a\}_{k_1})]\rangle \mid S\ )$$

and C_2 as:

$$\begin{aligned} C_2' \ \triangleq\ & (\nu\, r)(\ \overline{discovery}\langle\mathtt{service\_d}[\mathtt{id}(id_2), \mathtt{abstr}(\{(\!|?z : ch(\mathsf{T})|\!)\overline{r}\langle z\rangle\}_{k_2})]\rangle \\ & \mid r.(?y : ch(\mathsf{T}))\, C'\ )\ . \end{aligned}$$

Suppose $a \in ch(\mathsf{T}')$, $r \in ch(ch(\mathsf{T}))$ and assume S and C' are well typed under the appropriate contexts. The global system

$$P_s \triangleq (\nu\, k_1,\ k_2)(Dir_s \mid C_1' \mid C_2')$$

is well typed too. An attacker may intercept documents on *publish* or *discovery* and may learn the identifiers of the clients, but not the secret shared keys. As a consequence, it cannot have Dir_s publish unauthorized services or run unauthorized abstractions.

3.6 Barbed equivalence

In [126], Milner and Sangiorgi propose *barbed bisimulation* as a tool for uniformly defining bisimulation-based equivalences. Barbed equivalence is useful for its "portability" when studying a new calculus or a refinement of an existing one, as we are doing here.

Barbed bisimulation is a very coarse relation. According to a common pattern, one closes barbed bisimulation under all contexts, thus getting barbed congruence. Here, we find useful to depart from this pattern so as to capture an *input locality* property for the observed processes (in the same vein as [125]). Approximately, one may think of each observed process P as equipped with an "interface" I: a set of input channels at which services are offered. Input channels in I should remain confined to P, in other words, only external observers that do not own the input capability on channels in I should be considered. Moreover, one wants to consider only well-typed processes and observers. These considerations motivate a form of barbed equivalence presented in the sequel.

The definition relies on the reduction relation of the calculus and on an *observation predicate (barb)* $P \downarrow_a$, which detects the possibility for P of immediately interacting along port a. Being in an asynchronous setting, we restrict our attention to output ports (see e.g. [12]). Thus, in XPi, $P \downarrow_{\overline{a}}$ holds true if P has an output action $\overline{a}\langle M\rangle$, for some M, which is not in the scope of another prefix, or of $(\nu\, a)$ or of an *else* operator; $P \Downarrow_{\overline{a}}$ means that for some P', $P \rightarrow^* P'$ and $P' \downarrow_{\overline{a}}$. We define a version of barbed bisimulation that respects an input interface I. This means output at names in I are not observed, because the observer has not the corresponding input capability.

Definition 3.15 (I-barbed bisimulation). *Let $I \subseteq \mathcal{N}$. A symmetric binary relation on annotated closed processes is a* I-barbed bisimulation *if $(P, R) \in \mathcal{R}$ implies:*

- *whenever $P \rightarrow P'$ then there is R' such that $R \rightarrow^* R'$ and $(P', R') \in \mathcal{R}$;*
- *whenever $P \downarrow_{\overline{a}}$ and $a \notin I$ then $R \Downarrow_{\overline{a}}$.*

Two processes P and R are I-barbed bisimilar*, written $P \dot{\approx}^I R$, if $(P, R) \in \mathcal{R}$ for some I-barbed bisimulation $\mathcal{R}$.*

Note that one gets ordinary barbed bisimulation by setting $I = \emptyset$. The next step is closing I-barbed bisimulation under appropriate contexts, while respecting input locality for names in I. In the sequel, let us denote by $\mathrm{in}(P)$ the set of names that occur free in P in input subject position; similarly for $\mathrm{in}(M)$. Note that, following e.g. [29], we only close under static contexts (in π-calculus, one gets ordinary early bisimulation this way.)

Definition 3.16 (I-barbed equivalence). *Let $I \subseteq \mathcal{N}$. Two well-typed processes P_1 and P_2 are* I-barbed equivalent*, written $P_1 \approx^I P_2$, if for each $\widetilde{h}$ and each well-typed R s.t. $\mathrm{in}(R) \cap I = \emptyset$, it holds that $(\nu\, \widetilde{h})(P_1 \mid R) \dot{\approx}^I (\nu\, \widetilde{h})(P_2 \mid R)$.*

Ordinary barbed equivalence is obtained by setting $I = \emptyset$. Note that I-barbed equivalence is not a congruence (not even ordinary barbed equivalence is), but it is

preserved by restriction, and by parallel composition with those well-typed R s.t. $\mathrm{in}(R) \cap I = \emptyset$.

Example 3.6.1. This example illustrates the effect of considering only well-typed contexts. Suppose $I = \emptyset$ and $a \in ch(\mathtt{f}[\mathsf{int}])$ and consider

$$P = a.(?x : \mathtt{f}[*\mathsf{int}])\ \mathit{case}\ x\ \mathit{of}\quad \begin{array}{ll} \mathtt{f}[\,] & \Rightarrow\ P_1 \\ \_ & \Rightarrow\ P_2\,. \end{array}$$

Clearly, $P \approx^I a.(?x : \mathtt{f}[*\mathsf{int}])P_2$, because no well-typed context ever sends $\mathtt{f}[\,]$ along a, hence the first branch of the *case* is never triggered. Note that this equality does not hold for untyped barbed equivalence.

Example 3.6.2 (a web service, continued). Consider the web service WS and the clients C_1 and C_2 defined in Example 3.2.6, and let $I = \{\mathit{stream}, \mathit{download}\}$. The following equality states that, not surprisingly, requesting WS a streaming service is functionally equivalent to requesting download and then running the player locally, regardless of the capacity of the employed channels (high or low):

$$WS \,|\, C_1 \approx^I WS \,|\, C_2\,.$$

The above equality does not hold for ordinary ($I = \emptyset$) barbed equivalence, because, e.g. C_1 has an output barb on stream, which C_2 does not.

Note that, although defined over all closed processes, $\approx^I$ only makes sense for those processes that do not export input capability of names in I. In the ordinary π-calculus, passing names with only the output capability (plus some mild conditions on replication [11]) is sufficient to guarantee input locality. In XPi this is not the case, in fact input capabilities can be exported by "packaging" input channels in abstractions that are passed around, as for $P = \overline{a}\langle([\,])b.([\,])\mathbf{0}\rangle \,|\, P'$ and $I = \{b\}$ in $P \,|\, a.(?x)(x \bullet [\,]) \rightarrow^* P' \,|\, b.([\,])\mathbf{0}$.

3.7 Conclusions

In this chapter, we have presented XPi, a core calculus for XML messaging, featuring asynchronous communications, pattern matching, name and code mobility, static and dynamic typing. We have proved results on run-time safety and presented a notion of barbed equivalence that is useful to validate interesting equations. Flexibility of the language has been demonstrated by a few examples, mainly concerning description and discovery of web services.

The presence of abstractions makes XPi somewhat related to *higher-order* π-calculus [124], an extension of the π-calculus where processes can be passed around. In fact, XPi might also be viewed as a typed version of higher-order π-calculus with structured documents and pattern matching.

A relevant feature of XPi's type system, is that it is entirely static: static type checking and plain pattern matching suffice, as types of pattern variables are checked statically against channel capacities. We confine dynamic type checking to dynamic abstractions, which can be used whenever no refined typing information on incoming documents is available (e.g. at channels of capacity $\mathbf{T}$).

Chapter

FOUR

Astuce: a typed calculus for querying distributed XML documents

In this chapter we define Astuce*: a typed process calculus studied for querying large and distributed* XML *documents. What we achieve is a functional, strongly-typed programming model where* XML *data are processes that can be queried by means of concurrent pattern-matching expressions. Astuce is based on three main ingredients: an asynchronous process calculus that draws features from π-calculus and concurrent-*ML*; a model where both documents and expressions are represented as processes, and where evaluation is represented as a parallel composition of the two; a static type system based on regular expression types.*

4.1 Introduction

The World Wide Web (WWW) operates as a networked information system that imposes constraints on *resources*. Resources are objects in the system identified via *Uniform Resource Identifiers* (URIs). URIs are used to identify and directly or indirectly address resources, which are described and exchanged in a variety of widely-understood data formats, such as XML, HTML, CSS, JPEG and PNG. An even more constrained architectural style for Web applications has been proposed by Roy Fielding [70] and is known as *Representation State Transfer* (REST). Many people see it as a model for how to build WS. The REST Web is the subset of the WWW in which uniform interfaces are provided. Essentially create, retrieve, update and delete operations are allowed on resources, rather than arbitrary or application-specific functions. The primary purpose of the so called REST-*compliant* WS is to manipulate

XML representations of Web resources using a uniform set of operations. Well-known examples of REST-compliant services are the web-search engines, where a fixed set of operations, on usually large and distributed web documents, is provided.

In this chapter, we consider some facets of the REST approach and we concentrate on the specific problems related to manipulating, principally querying, large and distributed XML documents. Actually, the model we define here cannot be viewed as a model for REST-WS, mostly because we restrict our attention to *create* and *retrieve* operations. As an example of application of Astuce, consider the computation of a *reverse web-link graph* in search engines [63]. That is the computation of a list of web pages, which contain a link to a common target URL. Distribution, concurrency and dynamic acquisition of data must be explicitly taken into account when designing an effective computational model for this kind of applications.

We most particularly pay attention to the processing of messages in a distributed setting. Our proposal takes the form of a process calculus, called *Astuce*, in which XML data are processes that can be queried by means of concurrent pattern-matching expressions. In this model, the evaluation of patterns is distributed among locations, in the sense that the evaluation of a pattern at a node triggers concurrent evaluations of sub-patterns at other nodes, and actions can be carried out upon success or failure of patterns. Syntax and semantics are introduced in Section 4.2. The calculus also provides primitives for storing and aggregating the results of intermediate computations and for orchestrating the evaluation of patterns. In this respect, we radically depart from previous works on XML-centered process calculi, like e.g. XPi and [34, 75], where queries would be programmed as operations invoked on (servers hosting) WS and XML documents would be exchanged in messages. In contrast, we view queries as code being dispatched to the locations "hosting" a document. This shift of view is motivated by our target application domain. In particular, our model is partly inspired by the *MapReduce* paradigm described in [63] that is used to write programs to be executed on Google's large clusters of computers in a simple functional style. Continuing with the reverse web-link graph example above (developed in Example 4.3.2), assume that the documents of interest are cached on different, perhaps replicated, servers. A query that accomplishes the aforementioned task would dispatch sub-queries to every server and create a dedicated reference cell to aggregate the partial results from each server. Sub-queries sift the local documents and transmit to the central reference cell sequences of pages with a link to the target URL, so as to eventually produce the global reverse web-link graph. To achieve reliability, sub-queries may have to report

back periodically with status updates while the "master query" may decide to abort or reinstate queries in case of servers failure.

Another important feature of our model is the definition of a static type system (Section 4.3) based on *regular expression types*, inspired by XDuce and ℂDuce's type systems, which is compatible with Document Type Definitions (DTD) and other XML schema languages. The soundness of the static semantics is proved in Section 4.4 via a subject reduction property. In Section 4.5 we propose possible extensions of the calculus: match construct, concurrency primitives and exceptions. What we achieve is a functional, strongly-typed programming model for computing over distributed XML documents based on three main ingredients: a semantics defined by an asynchronous process calculus in the style of the π-calculus [109] and proposed semantics for concurrent-ML [69]; a model where documents and expressions are both represented as processes, and where evaluation is represented as a parallel composition of the two; and a type system based on regular expression types. Each of these choices is motivated by a feature of the problem: the study of service-oriented applications calls for including concurrency and explicit locations; the need to manipulate large, possibly dynamically generated, documents calls for a streamed model of processing; the documents handled by a service should often obey a predefined schema, hence the need to check that queries are well-typed, preferably before they are executed or "shipped".

4.2 Syntax and semantics of Astuce's terms

4.2.1 Syntax

We consider a simple language of first-order functional expressions, denoted $e, e', \ldots$, enriched with references and recursive *pattern* definitions that are used to extract values from *documents*. Consider an infinite set of *tag names*, $\mathcal{F}$, ranged over by $\mathtt{f}, \mathtt{g}, \ldots$; suppose $\mathcal{F}$ contains a reserved fictitious tag `root` used during pattern-matching evaluation for representing the root tag of XML documents. Consider also a countable set of *names*, $\mathcal{N}$, partitioned into *locations*, $\imath, \jmath, \ell, \ldots$, and *variables*, $x, y, \ldots$. We usually use the notation $\widetilde{\imath}$, $\widetilde{x}$, ... for tuples of names. Before formally defining the calculus, we introduce its main ingredients: documents and patterns.

Documents

An XML document may be seen as a simple textual representation for nested sequences of elements $<\mathtt{f}> \cdots </\mathtt{f}>$. As in the previous chapter, here we follow notations similar to XDuce and choose the simplified version of documents by leaving

aside attributes among other things. A document is an ordered sequence of elements $\mathtt{f}_1[M_1] \cdots \mathtt{f}_n[M_n]$, where $M_1, \ldots, M_n$ are documents. Documents may be empty, denoted $\mathtt{()}$, and can be concatenated, denoted M, M'. The composition operation is associative with identity $\mathtt{()}$. In what follows, let $\mathcal{D}$ be the set containing all documents.

In the following we consider distributed documents, meaning that each element $\mathtt{f}_j[M_j]$ is placed in a given location, say $\imath_j$. Locations are visible only at the level of the operational semantics, in which the contents of a document is represented by the index $\imath_1 \cdots \imath_n$ composed by the list of locations of its elements. As shown in the following example, if a document M_j is stored at index $\imath_1 \cdots \imath_m$, $\mathtt{f}_j[M_j]$ is written as $\mathtt{f}_j(\imath_1 \cdots \imath_m)$. In what follows we sometimes abbreviate indexes, like $\imath_1 \cdots \imath_n$, as $u, v, \ldots$. For the sake of simplicity, locations and indexes are the only values handled in Astuce and we leave aside atomic data values such as strings or integers, which can be easily accommodated and are used only in examples.

Example 4.2.1. On the left we present a fragment of an XML document containing a family tree. Each element describes a man or a woman and is characterized by a name and two lists containing respectively his/her daughters (a list of women) and sons (a list of men). On the right we introduce the representation of each element as an Astuce's *resource*; the global document is represented as the parallel composition of them.

$<$ family $>$	$\langle\, \imath \mapsto node\ \mathtt{family}(\imath_1, \ldots) \,\rangle$
$<$ man $>$	$\langle\, \imath_1 \mapsto node\ \mathtt{man}(\jmath_1, \jmath_2, \jmath_3) \,\rangle$
$<$ name $>$John Doe$<$ /name $>$	$\langle\, \jmath_1 \mapsto node\ \mathtt{name}(\mathtt{John\ Doe}) \,\rangle$
$<$ daughters $>$	$\langle\, \jmath_2 \mapsto node\ \mathtt{daughters}(\jmath_4) \,\rangle$
$<$ woman $>$	$\langle\, \jmath_4 \mapsto node\ \mathtt{woman}(\jmath_5, \jmath_6, \jmath_7) \,\rangle$
$<$ name $>$Jenny Doe$<$ /name $>$	$\langle\, \jmath_5 \mapsto node\ \mathtt{name}(\mathtt{Jenny\ Doe}) \,\rangle$
$<$ daughters $><$ /daughters $>$	$\langle\, \jmath_6 \mapsto node\ \mathtt{daughters}(\mathtt{()}) \,\rangle$
$<$ sons $><$ /sons $>$	$\langle\, \jmath_7 \mapsto node\ \mathtt{sons}(\mathtt{()}) \,\rangle$
$<$ /woman $>$	
$<$ /daughters $>$	
$<$ sons $>$	$\langle\, \jmath_3 \mapsto node\ \mathtt{sons}(\ldots) \,\rangle$
$\vdots$	$\vdots$

As we will see in the formal presentation of the calculus, the keyword *node* simply indicates that a location contains a (fragment of a) document.

Selectors and Patterns

The core of our programming model is a system of distributed pattern matching expressions that concurrently sift through documents to extract information. Like functions, patterns are declared and have a name. Patterns take the form of regular tree expressions enriched with variables used for capturing values. Informally, the declaration

$$Q(\widetilde{x}) \stackrel{\triangle}{=} \mathit{let}\ \left(z_1 = e'_1, \ldots, z_m = e'_m\right)\ \mathit{in}\ \left(\mathit{Reg}(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}\ \mathit{as}\ y\right)\ \mathit{then}\ e_1\ \mathit{else}\ e_2$$

with $\bigcup_{i=1,\ldots,n} \widetilde{x}_i \cup \{y\} \subseteq \widetilde{x} \cup \bigcup_{j=1,\ldots,m} \widetilde{z}_j$ and $\widetilde{x} \cap \bigcup_{j=1,\ldots,m} \widetilde{z}_j = \emptyset$, defines a pattern called Q, with parameters $\widetilde{x}$. Q collects all matched documents in the variable y, also called *capture variable*. When the pattern-matching between Q and a document is evaluated, the expressions $e'_1, \ldots, e'_m$ are evaluated first and their final values are bounded to variables $z_1, \ldots, z_m$ respectively. After, the given document is matched against $\mathit{Reg}(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}$. If the matching succeeds, variable y is used for capturing (storing) the matched document and continuation e_1 is executed. Otherwise compensation e_2 is started. These optional continuations allow to add basic exception and transaction mechanisms to the calculus. In all these steps, variables in $\widetilde{x}$ and $z_1, \ldots, z_m$ are substituted by the received parameters and the results of the evaluations of $e'_1, \ldots, e'_m$, respectively. $\mathit{Reg}(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}$ is called *selector* and stands for a generic regular expression over the tagged patterns $\mathtt{f}_i[Q_i]$ for $i = 1, \ldots, n$.

Definition 4.1 (patterns). *The set $\mathcal{Q}$ of* patterns*, $Q, Q', \ldots$ contains elements defined by using a set of recursive definitions of the form*

$$Q(\widetilde{x}) \stackrel{\triangle}{=} \mathit{let}\ \left(z_1 = e'_1, \ldots, z_m = e'_m\right)\ \mathit{in}\ \left(\mathit{Reg}\ \mathit{as}\ y\right)\ \mathit{then}\ e_1\ \mathit{else}\ e_2$$

where $Q \in \mathcal{Q}$ is a pattern identifier, $\widetilde{x} \cup \bigcup_{i=1,\ldots,m} z_i \subseteq \mathcal{N}$, $y \in (\widetilde{x} \cup \bigcup_{i=1,\ldots,n} z_i)$, the set of the free variables in Reg is a subset of $\widetilde{x} \cup \bigcup_{i=1,\ldots,n} z_i$ and Reg is a selector defined by the grammar in Table 4.1.

The syntax in Table 4.1 is essentially a syntax for defining regular tree grammar and is self-explaining. The choice operator is associative, commutative and has *Empty* as unit.

Notations. In what follows, we write *Reg* for a generic selector and we use the notation $\mathit{Reg}(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}$, when it is necessary to specify tags and patterns involved in the selector. While in examples we sometimes consider "complete" pattern definitions, for easy of presentation, we formally introduce the calculus by considering pattern definitions without local *let* declarations and continuations ($Q(\widetilde{x}) \stackrel{\triangle}{=}$

Selector	*Reg*	::=	$\mathtt{f}[Q(\widetilde{x})]$	*Tagged Pattern*
		|	*All*	*All*
		|	*Empty*	*Empty*
		|	*Reg* | *Reg*	*Choice*
		|	*Reg*, *Reg*	*Sequence*
		|	*Reg*$*$	*Iteration*

Table 4.1: Syntax of selectors.

$Reg(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}\, as\, y)$. For the sake of completeness, in Section 4.5 we extend the calculus to full pattern definitions.

Example 4.2.2. The pattern *names* defined below can be used to extract the names of all persons occurring in the document defined in Example 4.2.1.

$$\begin{aligned} names(x,y) &\triangleq \big(\mathtt{man}[p(x,y,x)] \,|\, \mathtt{woman}[p(x,y,y)]\big)* \\ p(x,y,z) &\triangleq \mathtt{name}[all(z)], \mathtt{daughters}[names(x,y)], \mathtt{sons}[names(x,y)] \\ all(z) &\triangleq All\ \ as\, z\ . \end{aligned}$$

A call to $names(\imath, \ell)$ stores in (the reference located at) $\imath$ the name of all men and in ℓ the name of all women. A call to $names(\ell, \ell)$ will store the names of all persons in ℓ.

An important feature of our model is that patterns may extract multiple sets of values from documents in one pass, which contrasts with the monadic queries expressible with technologies such as XPath.

Witness and Unambiguous Patterns. Next, we define what it means for a pattern to match an index and define the notion of *unambiguous* pattern. Assume *Reg* is the selector $Reg(\mathtt{f}_i[Q_i(\widetilde{v}_i)])_{i=1,\ldots,m}$, where $\widetilde{v}_i$ are the actual parameters used in the pattern invocation. The sequence $\mathtt{f}_{i_1} \cdots \mathtt{f}_{i_n}$ matches *Reg* if and only if it is a "word" in the language of $Reg(\mathtt{f}_i)_{i=1,\ldots,m}$. Consider the relation $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} c_1 \cdots c_n$, with $c_i ::= Q(\widetilde{v}) \mid All \mid Empty$, defined in Table 4.2.

Definition 4.2 (witness). *The sequence $c_1 \cdots c_n$ is a* witness *for Reg of $\mathtt{f}_1 \cdots \mathtt{f}_n$ if $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} c_1 \cdots c_n$ holds. We write $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{Reg}$ if the sequence has no witness for Reg.*

The rules that define $\vdash_{Reg}$ are easy to understand and do not deserve explanations.

$$\text{(W-Empty)}\ \texttt{()} \vdash_{Reg} Empty \quad \text{with } Reg ::= Empty \mid All \mid Reg*$$

$$\text{(W-Atom)}\ \mathtt{f} \vdash_{\mathtt{f}[c]} c \qquad \text{(W-All)}\ \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{All} All \cdots All$$

$$\text{(W-Choice)}\ \frac{\exists i \in \{1,2\} :\ \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg_i} c_1 \cdots c_n}{\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg_1 \mid Reg_2} c_1 \cdots c_n}$$

$$\text{(W-Star)}\ \frac{\exists i \in \{1,\ldots,n\} :\ \begin{array}{l} \mathtt{f}_1 \cdots \mathtt{f}_i \vdash_{Reg} c_1 \cdots c_i \\ \mathtt{f}_{i+1} \cdots \mathtt{f}_n \vdash_{Reg*} c_{i+1} \cdots c_n \end{array}}{\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg*} c_1 \cdots c_n}$$

$$\text{(W-Seq)}\ \frac{\exists i \in \{0,\ldots,n\} :\ \begin{array}{l} \mathtt{f}_1 \cdots \mathtt{f}_i \vdash_{Reg_1} c_1 \cdots c_i \\ \mathtt{f}_{i+1} \cdots \mathtt{f}_n \vdash_{Reg_2} c_{i+1} \cdots c_n \end{array}}{\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg_1, Reg_2} c_1 \cdots c_n}$$

Table 4.2: Relation $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} c_1 \cdots c_n$.

It is standard in XML to restrict to expressions that denote sequences of elements unequivocally. Some schema languages, like DTDs for example [35], restrict to *one-unambiguous* expressions, that is to expressions for which the witnesses can be computed incrementally, reading from a sequence of tags with only one symbol lookahead. While this notion is suitable when working with streamed data of ordered documents, it may impose needless performance penalties when working in a truly concurrent way. For instance, one can be interested in being able to start the evaluation on an element without necessarily matching all its preceding siblings beforehand (a sort of "non-ordered" evaluation). For this, one can require an even stronger notion of unambiguity and say that a selector $Reg(\mathtt{f}_i(Q_i(\widetilde{x}_i)))_{i=1,\ldots,n}$ is *consistently unambiguous* if every tag specifies a unique pattern, i.e. whenever $\mathtt{f}_i = \mathtt{f}_j$ then $Q_i(\widetilde{x}_i)$ and $Q_j(\widetilde{x}_j)$ are the same. A more flexible, but more complex alternative solution, would be to require that, for every sequence of tags and every integer i, the i^{th} component of a witness can be computed only from the value of the i^{th} tag.

For the sake of simplicity, we consider a weaker notion of unambiguity, which allow to avoid backtracking during ("ordered") pattern-matching evaluations on non-streamed documents. In particular, we define unambiguous patterns as follows.

Definition 4.3 (unambiguous pattern). *A selector Reg is said to be* unambiguous *if each sequence of tags has at most one witness for Reg. A pattern is said* unambiguous *if has a unambiguous selector.*

Assume that $c_1 \cdots c_m$ is "the witness" of *Reg* for $\mathtt{f}_1 \cdots \mathtt{f}_m$. When a document

$\mathtt{f}_1[v_1] \cdots \mathtt{f}_m[v_m]$ is matched against a pattern with selector *Reg*, each sub-document v_j is matched against c_j. If $\mathtt{f}_1 \cdots \mathtt{f}_m$ has no witness then the pattern-matching fails. Obviously this notion of unambiguous patterns is not suitable when working with streamed documents or when working in a truly concurrent way because it requires to know in advance the entire word.

Example 4.2.3. We remark the differences among the various definitions of unambiguity by using some simple examples. Suppose $Q_i \neq Q_j$ for $i \neq j$.

Selector $\mathtt{f}[Q_1], \mathtt{g}[Q_2], \mathtt{h}[Q_3] \mid \mathtt{f}[Q_1], \mathtt{g}[Q_2], \mathtt{h}[Q_4]$ is considered ambiguous independently from the chosen definition. In fact tag `h` appears twice as third element of the word `fgh` and specifies two different patterns, Q_3 and Q_4. In other words, there are two witnesses for `fgh`. Moreover, this means that the witness for the word `fgh` cannot be computed incrementally and the 3^{rd} component of the witness for `fgh` cannot be computed from the value of the tag.

Selector $\mathtt{f}[Q_1], \mathtt{g}[Q_2], \mathtt{h}[Q_3] \mid \mathtt{f}[Q_4], \mathtt{g}[Q_2], \mathtt{l}[Q_5]$ is unambiguous according to Definition 4.3, while is ambiguous according to each other definition (because tag `f` is used twice in first position and specifies two different patterns).

Selector $\mathtt{f}[Q_1], \mathtt{g}[Q_2], \mathtt{h}[Q_3] \mid \mathtt{h}[Q_1], \mathtt{g}[Q_4], \mathtt{f}[Q_3]$ is unambiguous according to Definition 4.3 and to one-unambiguity, but not according to the others. This because tag `g` is used twice as second tag and specifies two different patterns.

Finally, selector $\mathtt{f}[Q_1], \mathtt{h}[Q_2], \mathtt{g}[Q_3] \mid \mathtt{f}[Q_1], \mathtt{g}[Q_4], \mathtt{h}[Q_5]$ satisfies all definitions of unambiguous pattern except for the strongest one (consistent unambiguity).

Syntax of the calculus

The presentation of the calculus can be naturally divided into two fragments: a language of functional expressions, or *programs*, that are used in the body of pattern and function declarations; and a language of processes, or *configurations*, that models distributed documents and the concurrent execution of programs.

In the following, we assume that every function identifier f has associated arity $n \geqslant 0$ and a unique definition $f(\widetilde{x}) \stackrel{\triangle}{=} e$ where the variables in $\widetilde{x}$ are distinct and include all free variables of e. We take similar hypotheses for patterns.

Definition 4.4. *The sets $\mathcal{E}$ of* expressions $e, e', \ldots$ *and $\mathcal{P}$ of* processes $P, R, \ldots$ *are defined by the syntax in Table 4.3.*

The first part of Table 4.3 defines the functional part of the calculus. A result is either a variable or an index $\imath_1 \cdots \imath_n$, that is a possibly empty sequence of locations. A result is an expression that immediately returns itself. Expressions include results;

Result	u, v	$::=$	x	*Name*
			$\mid \imath_1 \cdots \imath_n,\ n \geqslant 0$	*Index*
Expression	e	$::=$	u	*Result*
			$\mid \mathtt{f}[u]$	*Element creation*
			$\mid u, v$	*Result composition*
			$\mid f(u_1, \ldots, u_n)$	*Function call*
			$\mid let\ x = e\ in\ e$	*Let*
			$\mid newref\ u$	*New reference*
			$\mid !u$	*Dereferencing*
			$\mid u \mathrel{+}= v$	*Update*
			$\mid try\ v\ Q(u_1, \ldots, u_n)$	*Pattern matching call*
			$\mid wait\ u(x)\ then\ e\ else\ e$	*Wait matching*
Process	P, R	$::=$	e	*Expression*
			$\mid let\ x = P\ in\ R$	*Let*
			$\mid \langle\, \imath \mapsto d \,\rangle$	*Location*
			$\mid P \Rsh R$	*Parallel composition*
			$\mid (\nu\, \imath) P$	*Restriction*
Resource	d	$::=$	$ref\ u$	*Reference with value u*
			$\mid node\ \mathtt{f}(u)$	*Node*
			$\mid try\ \imath\ Q(u_1, \ldots, u_n)$	*Try matching*
			$\mid test\ \imath\ u\ v$	*Test matching*
			$\mid ok\ \imath$	*Successful match*
			$\mid fail\ \imath$	*Failed match*

Table 4.3: Syntax of the calculus.

operators for creating new elements, $\mathtt{f}[u]$; operators for concatenating results u, v; function calls $f(u_1, \ldots, u_n)$; and operators for creating, accessing and updating references. Reference update has a slightly unusual semantics since the effect of $\imath \mathrel{+}= v$ is to append v to the value stored in the reference $\imath$. Actually, we could imagine that each reference is associated with an "aggregating function" that specifies how the sequence of values stored in the reference has to be combined. Here, we only consider index composition.

The expression $try\ v\ Q(u_1, \ldots, u_n)$ is used to apply the pattern Q to the index $v = \imath_1 \cdots \imath_n$. A *try* expression returns at once with the location of a fresh node where

the matching occurs. Moreover, evaluation of patterns is carried out concurrently: the effect of evaluating $let\ z = \big(try\ v\, Q(u_1, \ldots, u_n)\big)\ in\ P$ is to filter v by Q concurrently with the evaluation of P. In this example, z is bound to the location of the "thread" that executes the *try* expression, say ℓ. The location ℓ can be tested in P to check whether the pattern-matching has ended using the expression $wait\ \ell(x)\, then\, e_1\, else\, e_2$. The *wait* statement blocks until the pattern evaluating at ℓ stops. Then the continuation e_1 is evaluated if the matching succeeds, otherwise e_2 is evaluated. In both cases the variable x is bound to v.

The second part of Table 4.3 defines the syntax of configurations. The calculus features operators from the π-calculus: restriction $(\nu\, \imath)P$ specifies the scope of a name $\imath$ local to P; parallel composition $P \Rsh R$ represents the concurrent evaluation of P and R. The result of evaluating $P \Rsh R$ is the result of R's evaluation, this implies that "$\Rsh$" is only *left-commutative* as explained in Section 4.2.2. Overall, a process is a multiset of *let* expressions, describing threads execution, and locations $\langle\, \imath \mapsto d \,\rangle$, that describes a *resource* d located at $\imath$.

The calculus is based on an abstract notion of location that is, at the same time, the minimal *unit of interaction* and the minimal *unit of storage*. Failures are not part of this model (they can be viewed as an orthogonal feature) but could be added, e.g. in the style of [11]. Locations store resources that are generated at run-time. The main resources are $ref\ u$, to store the current state of a reference, and $node\ \mathtt{f}(u)$, to describe an element of the form $\mathtt{f}[M]$ if document M is stored at index u. The calculus explicitly takes into account the distribution of document nodes and, for example, the document $\mathtt{f}[\mathtt{g}[\,]\,\mathtt{h}[\,]]$ can be represented (at run-time) by the parallel composition:

$$(\nu\, \imath_1 \imath_2)\big(\langle\, \imath \mapsto node\ \mathtt{f}(\imath_1\, \imath_2) \,\rangle \Rsh \langle\, \imath_1 \mapsto node\ \mathtt{g}(()) \,\rangle \Rsh \langle\, \imath_2 \mapsto node\ \mathtt{h}(()) \,\rangle\big)\ .$$

The other resources arise in the evaluation of pattern-matching and correspond to different phases in its execution: scheduling a "pattern call" (*try*); waiting for the result of sub-pattern evaluations (*test*); stopping and reporting success (*ok*) or failure ($fail$). In all these cases $\imath$ is the index corresponding to the resource containing the document to analyze. Moreover, in $test\ \imath\ u\ v$, u is the index of the ongoing sub-pattern evaluations and v is the capture reference to be updated with the index of the node located at $\imath$ in case of successful matching.

Binding conventions and Notations. We stipulate that the operators *let*, *wait* and ν are name binders. Notions of alpha-equivalence and of free and bound names ($\mathrm{fn}(\cdot)$ and $\mathrm{bn}(\cdot)$) and free variables ($\mathrm{fv}(\cdot)$) arise as expected. Finally, we identify expressions and terms up to alpha-equivalence.

$$
\begin{array}{ll}
(P \Rsh S) \Rsh R \equiv P \Rsh (S \Rsh R) & (\nu\imath)(P \Rsh R) \equiv ((\nu\imath)P) \Rsh R,\ \imath \notin \mathrm{fn}(R) \\
(P \Rsh S) \Rsh R \equiv (S \Rsh P) \Rsh R & (\nu\imath)(P \Rsh R) \equiv P \Rsh (\nu\imath)R,\ \imath \notin \mathrm{fn}(P) \\
(\nu\imath)(\nu\ell)P \equiv (\nu\ell)(\nu\imath)P & P \Rsh \mathit{let}\ x = S\ \mathit{in}\ R \equiv \mathit{let}\ x = (P \Rsh S)\ \mathit{in}\ R,\ x \notin \mathrm{fn}(P) \\
\multicolumn{2}{c}{(\nu\ell)\mathit{let}\ x = P\ \mathit{in}\ R \equiv \mathit{let}\ x = (\nu\ell)P\ \mathit{in}\ R,\ \ell \notin \mathrm{fn}(R)} \\
\multicolumn{2}{c}{\mathit{let}\ y = (\mathit{let}\ x = P\ \mathit{in}\ S)\ \mathit{in}\ R \equiv \mathit{let}\ x = P\ \mathit{in}\ (\mathit{let}\ y = S\ \mathit{in}\ R),\ x \notin \mathrm{fn}(R)}
\end{array}
$$

Table 4.4: Structural congruence.

In the following, we make use of these abbreviations: if $u = \imath_1 \cdots \imath_n$ then $(\nu\, u)P$ is a shorthand for $(\nu\,\imath_1) \cdots (\nu\,\imath_n)P$; the term $(\nu\,\ell)P \Rsh R$ stands for $((\nu\,\ell)P) \Rsh R$; the term $\mathit{let}\ x = P\ \mathit{in}\ R \Rsh S$ stands for $(\mathit{let}\ x = P\ \mathit{in}\ R) \Rsh S$; and $\mathit{wait}\ \ell(x)\,\mathit{then}\, e_1$ stands for $\mathit{wait}\ \ell(x)\,\mathit{then}\, e_1\, \mathit{else}\, ()$ (and similarly for omitted *then* clause). Moreover, we indicate with $()$ an empty index, that is $() = \imath_1 \cdots \imath_n$ if $n = 0$.

Definition 4.5 (closed processes). *We denote with $\mathcal{P}_{cl}$ the set containing all processes $P \in \mathcal{P}$ such that* $\mathrm{fv}(P) = \emptyset$*, that is all closed processes.*

4.2.2 Reduction semantics

Following [109], Astuce's semantics is based on *structural congruence* and a *reduction relation*.

Definition 4.6 (structural congruence). *The* structural congruence $\equiv$ *is the least congruence satisfying the rules in Table 4.4.*

The only rule which deserves some explanation is the one stating commutativity of the parallel composition operator. Since a process may return a value, we take the convention that the result of a composition $P_1 \Rsh \cdots \Rsh P_n$ is the result of its rightmost term P_n. The values returned by the other processes are discarded. This entails that the order of parallel components is relevant. For this reason, unlike the situation in most process calculi, parallel composition is not a commutative operator. Actually, composition is "left commutative", which means that $(P \Rsh S) \Rsh R$ is equivalent to $(S \Rsh P) \Rsh R$ but that we do not necessarily have $P \Rsh S$ equivalent to $S \Rsh P$. This choice is similar to what is found in calculi introduced for defining the semantics of concurrent-ML [69] and for concurrent extension of object calculi [78]. An advantage is that we directly include sequential composition of processes: the sequential composition $P; R$ can be interpreted by the term $\mathit{let}\ x = P\ \mathit{in}\ R$, where $x \notin \mathrm{fv}(R)$. Moreover it relieves us from the need to encode the operation of returning a result using continuations and sending a message on a result channel, as in the π-calculus.

In works mixing XML and process calculi, as XPi, usually documents are not represented as processes and pattern-matching evaluation is computed independently from process evolution by using an additional ad hoc function. Here, instead, pattern-matching evaluation is disciplined by reductions. The result of a pattern-matching evaluation is a *substitution* that involves the continuations of a possible *wait* expression waiting for the matching result.

Definition 4.7 (substitutions). Substitutions $\sigma, \sigma', \ldots$ *are finite partial maps from the set of variables* $\mathcal{V}$ *to results* $u, v, \ldots$. *For any term* P, $P\sigma$ *denotes the result of applying* σ *onto* P *(with alpha-renaming of bound names and variables if needed.)*

Assume σ is the substitution $[u_1/x_1, \ldots, u_n/x_n]$ and $\widetilde{u} = (u_1, \ldots, u_n)$. We write $f(\widetilde{u}) \stackrel{\triangle}{=} e'$ if $f(\widetilde{x}) \stackrel{\triangle}{=} e$ and $e' = e\sigma$ and we write $Q(\widetilde{u}) \stackrel{\triangle}{=} Reg'$ if the selector of $Q(\widetilde{x})$ is Reg and $Reg' = Reg\sigma$.

A reduction represents an individual computation step and is defined in terms of structural congruence and evaluation contexts.

Definition 4.8 (evaluation context). *The set* $\mathcal{C}$ *of* evaluation contexts, $E, E', \ldots$, *is defined by the following grammar:* $E ::= [.] \mid P \Rsh E \mid E \Rsh P \mid (\nu\, \ell)E \mid \mathit{let}\ x = E\ \mathit{in}\ P$.

For the sake of readability, in the presentation of the reduction semantics, we consider a simplified form of pattern without *let* definitions and continuations. We discuss the semantics of full patterns in Section 4.5.

Definition 4.9 (reduction). *The* reduction relation, $\rightarrow \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$, *is the least binary relation on closed processes satisfying the rules in Table 4.5.*

As usual, we denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. A detailed explanation of the reduction rules follows.

The rules for expressions are similar to traditional semantics for first-order languages, with the difference that resources in a configuration play the role of stores. Likewise, the rules for operators that return new values (the operators $newref$, $\mathtt{f}[\cdot]$ and try) yield reductions of the form $e \rightarrow (\nu\, \ell)(\langle\, \ell \mapsto d\, \rangle \Rsh \ell)$, which means that new values are always allocated in a fresh location. Actually, a quick inspection of the rules shows that resources are created in fresh locations and are always used in a linear way: an expression cannot discard a resource or create two different resources at the same location.

We can divide the rules in Table 4.5 according to the locations involved in the reduction. A location $\langle\, \ell \mapsto ref\ u\, \rangle$ is a reference at ℓ with value u. Reference access, rule (RD), replaces a top-level occurrence of $!\ell$ with the value u. Reference update

$$(\text{FUN})\ \frac{f \text{ declared as } f(\widetilde{x}) \triangleq e}{f(u_1,\ldots,u_n) \to e[\widetilde{u}/\widetilde{x}]} \qquad (\text{REF})\ \frac{u = \imath_1 \cdots \imath_n \quad \ell \text{ fresh name}}{newref\ u \to (\nu\,\ell)(\langle\, \ell \mapsto ref\ u \,\rangle \Rsh \ell)}$$

$$(\text{LET})\ \frac{}{let\ x = u\ in\ P \to P[u/x]} \qquad (\text{WR})\ \frac{w = u, v}{\langle\, \ell \mapsto ref\ u \,\rangle \Rsh \ell \mathrel{+}= v \to \langle\, \ell \mapsto ref\ w \,\rangle \Rsh ()}$$

$$(\text{STR})\ \frac{P \equiv R \quad R \to R' \quad R' \equiv P'}{P \to P'} \qquad (\text{RD})\ \frac{}{\langle\, \ell \mapsto ref\ u \,\rangle \Rsh !\ell \to \langle\, \ell \mapsto ref\ u \,\rangle \Rsh u}$$

$$(\text{CTX})\ \frac{P \to P'}{E[P] \to E[P']} \qquad (\text{ND})\ \frac{u = \imath_1 \cdots \imath_n \quad \imath \text{ fresh name}}{\mathtt{f}[u] \to (\nu\,\imath)(\langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle \Rsh \imath)}$$

$$(\text{COMP})\ \frac{u_1 = \imath_1 \cdots \imath_k \quad u_2 = \imath_{k+1} \cdots \imath_n}{u_1, u_2 \to \imath_1 \cdots \imath_n}$$

$$(\text{TRY})\ \frac{u = \imath_1 \cdots \imath_n \qquad \imath, \ell \text{ distinct fresh names}}{try\ u\, Q(\widetilde{v}) \to (\nu\,\imath)(\nu\,\ell)(\langle\, \imath \mapsto node\ \mathtt{root}(u) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \Rsh \ell\,)}$$

$$(\text{ERR})\ \frac{\begin{array}{c} P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \\ n \geqslant 0 \quad Q(\widetilde{v}) \triangleq Reg\ as\ v_r \quad \mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{Reg} \end{array}}{P \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \to P \Rsh \langle\, \ell \mapsto fail\ \imath \,\rangle}$$

$$(\text{MTC})\ \frac{\begin{array}{c} P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \quad n > 0 \\ Q(\widetilde{v}) \triangleq Reg\ as\ v_r \quad \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n) \quad w = \jmath_1 \cdots \jmath_n \ \text{ fresh names} \end{array}}{\begin{array}{l} P \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \to P \Rsh (\nu\, w)\big(\prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto try\ \imath_k\, Q_k(\widetilde{v}_k) \,\rangle \Rsh \\ \qquad\qquad \langle\, \ell \mapsto test\ \imath\ w\ v_r \,\rangle\big) \end{array}}$$

$$(\text{UNIT})\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(()) \,\rangle \quad Q(\widetilde{v}) \triangleq Reg\ as\ v_r \quad () \vdash_{Reg} Empty}{P \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \to P \Rsh \langle\, \ell \mapsto ok\ \imath \,\rangle}$$

$$(\text{TEST-OK})\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto ok\ \imath_k \,\rangle \quad z \neq \imath, \ell}{P \Rsh \langle\, \ell \mapsto test\ \imath\ \jmath_1 \cdots \jmath_n\ v_r \,\rangle \to P \Rsh let\ z = (v_r \mathrel{+}= \imath_1 \cdots \imath_n)\ in \langle\, \ell \mapsto ok\ \imath \,\rangle}$$

$$(\text{TEST-FAIL})\ \frac{\begin{array}{c} P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto d_k \,\rangle \\ \forall k = 1,\ldots,n : d_k \in \{ok\ \imath_k, fail\ \imath_k\} \qquad \exists j \in 1,\ldots,n : d_j = fail\ \imath_j \end{array}}{P \Rsh \langle\, \ell \mapsto test\ \imath\ \jmath_1 \cdots \jmath_n\ v_r \,\rangle \to P \Rsh \langle\, \ell \mapsto fail\ \imath \,\rangle}$$

$$(\text{WAIT-OK})\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle \Rsh \langle\, \ell \mapsto ok\ \imath \,\rangle}{P \Rsh wait\ \ell(x)\, then\, e_1\, else\, e_2 \to P \Rsh e_1[u/x]}$$

$$(\text{WAIT-FAIL})\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle \Rsh \langle\, \ell \mapsto fail\ \imath \,\rangle}{P \Rsh wait\ \ell(x)\, then\, e_1\, else\, e_2 \to P \Rsh e_2[u/x]}$$

Table 4.5: Reduction semantics.

$\ell \mathrel{+}= v$, rule (WR), has a slightly unusual semantics since its effect is to append v to the value stored in ℓ. In the general case, we have that the value of w in (WR) is given by $op(u, v)$, where op is some "aggregating" function that specifies how the values u and v have to be combined. For example, assume ℓ is an "integer reference" that increments its value by one on every assignment. Then, in Example 4.2.2, a call to $names(\ell, \ell)$ counts the number of people in a family tree document.

A location $\langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle$ is created by the evaluation of an element creation expression $\mathtt{f}[u]$, where u is an index, (ND). A location $\langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle$ is created by the evaluation of a *try* operator. The expression $try\ u\, Q(\widetilde{v})$ applies the pattern Q to the index $u = \imath_1 \cdots \imath_n$, rule (TRY). A *try* expression returns at once with the index ℓ of the fresh location where the matching occurs. It also creates a document node $\langle\, \imath \mapsto node\ \mathtt{root}(u) \,\rangle$ that points to the index u that is processed (we use the reserved tag `root` for the root of this node). Assume that *Reg* is the selector of Q, the *try* resource will trigger evaluation of sub-patterns selected from a witness of *Reg*. If there is no witness, the matching fails, rule (ERR). If a witness exists, the *try* resource spawns new *try* resources and turns into a *test*, rule (MTC), waiting for the results of these evaluations. Upon termination of all sub-patterns, a *test* resource turns into *ok* or $fail$, rules (TEST-OK) and (TEST-FAIL). In case of success, the capture reference v_r is updated with the index of the matched document. In case the witness is *Empty*, hence there are no sub-patterns to evaluate, the *try* resource becomes directly *ok*, (UNIT). Note that updating the capture reference v_r is meaningless – hence omitted – here as we consider only append of indexes as aggregating function for references. The *ok* and $fail$ resources are immutable.

The remaining rules are related to the evaluation of a *wait* expression. The status of a pattern evaluation can be checked with the expression $wait\ \ell(x)\ then\ e_1\ else\ e_2$, see rules (WAIT-OK) and (WAIT-FAIL). If the resource at ℓ is $ok\ \imath$ then the *wait* expression evaluates to $e_1[u/x]$, where u is the index of the node located at $\imath$. If the resource is $fail\ \imath$ then the expression evaluates to $e_2[u/x]$. In all other cases the expression is stalled. Note that by substituting x by u the fictitious tag `root`, added by rule (TRY) when pattern-matching evaluation started, is discarded.

Remark 4.1. In rule (MTC), we compute the witness for all children of an element in one go. This is not always realistic since the size of the children's index can be very large (actually, in real applications, big documents are generally shallow and have a large number of children). It is possible to refine the operational semantics so that each sub-pattern is fired independently, not necessarily following the order of the document. For instance, by considering *consistently-unambiguous* patterns instead

of *unambiguous* ones (as defined in the previous section), we should be able to start the evaluation on an element without necessarily matching all its preceding siblings beforehand. Also, we can imagine that indexes are implemented using streams or linked lists. We have chosen this presentation for the sake of simplicity.

Again in rule (MTC), we have assumed the presence of a capture reference v_r in the pattern. In case a capture is not specified, we can always define a fresh reference and use it as fictitious capture reference in the new generated *test* resource.

4.3 A type system

Applications that exchange and process XML documents rely on type information, such as DTDs, to describe structural constraints on the occurrences of elements. In this section we define a type system for Astuce that disciplines the typing of documents, by using regular types that can be compared to DTDs, and of processes, by using regular types and types for references and resources. The system guarantees that well-typed processes always deal with well-formed documents, that is with documents that can be typed by using regular types. Moreover, it ensures that reference updates and pattern-matching never generate typing errors. This means that a reference always contains elements complying with the corresponding type. Moreover, if a pattern-matching succeeds then the matched document complies with the type expected by the pattern.

4.3.1 Types and subtyping relation

We first introduce document types, which are regular expression types and define a subtyping relation among them. After, we continue by defining types for programs and configurations.

Document Types

In our model, document types, as patterns, take the form of regular tree expressions. Document types are particular kind of patterns: a pattern declaration without parameters, *let* definitions, capture variables and continuations is a type declaration. With a slight abuse of notation, in what follows we use the same keyword *Reg* for indicating both the selector of patterns and of types.

Definition 4.10 (document types). *The set $\mathcal{DT}$ of* document types $A, B, \ldots$ *contains elements defined by using a set of recursive definitions of the form $A \stackrel{\triangle}{=} Reg$*

where Reg is a selector defined by the grammar

$$Reg \ ::= \ \mathtt{f}[B] \ \big| \ \mathit{All} \ \big| \ \mathit{Empty} \ \big| \ Reg \ \big| \ Reg \ \big| \ Reg, Reg \ \big| \ Reg* \ .$$

The syntax describing types' selectors is essentially the syntax in Table 4.1, where patterns are substituted by types and keywords are written in sans serif typeface.

Every pattern $Q \in \mathcal{Q}$ can be associated with the type A obtained by erasing from Q *let* declarations, continuations and variables. A is the type of all documents that are matched by Q. In the following, we assume that functions and patterns are typed explicitly; in Remark 4.2 we extend the type system and define some typing rules that can be applied for verifying well-typedness of them.

There is a natural notion of subtyping $\mathsf{A} < \mathsf{B}$ between regular expression types, meaning that every document of type A is also of type B. The type system is close to what is defined in functional languages for manipulating XML, see e.g. XDuce [87, 88, 89] or the review in [51], hence we stay consistent with actual frameworks used in sequential languages for processing XML data.

Example 4.3.1. A is the type of all documents matched by the pattern *names* defined in Example 4.2.2. Note that the family tree document defined in Example 4.2.1 complies with this type.

$$\begin{array}{llll} \mathsf{A} \stackrel{\triangle}{=} (\mathtt{man}[\mathsf{P}] \,|\, \mathtt{woman}[\mathsf{P}])* & & \mathsf{P} \stackrel{\triangle}{=} \mathtt{name}[\mathsf{All}], \mathtt{daughters}[\mathsf{WL}], \mathtt{sons}[\mathsf{ML}] \\ \mathsf{WL} \stackrel{\triangle}{=} \mathtt{woman}[\mathsf{P}]* & & \mathsf{ML} \stackrel{\triangle}{=} \mathtt{man}[\mathsf{P}]* \end{array}$$

We assume that the pattern *names* is declared with type $(\mathsf{All}, \mathsf{All}) \to \mathsf{A}$. In general, a reference that merges values of type C will have a type B such that $\mathsf{B}, \mathsf{C} < \mathsf{B}$ (see (T-Wt) in Table 4.7).

Witness and subtyping. We can easily adapt the definition of witness to types. Assume A is declared as $\mathsf{A} \stackrel{\triangle}{=} Reg(\mathtt{f}_i[\mathsf{A}_i])_{i=1,\ldots,n}$. We say that there is a witness for A of $\mathtt{g}_1 \cdots \mathtt{g}_m$, denoted $\mathtt{g}_1 \cdots \mathtt{g}_m \vdash_{\mathsf{A}} \mathsf{c}_1 \cdots \mathsf{c}_n$ (with $c_i ::= \mathsf{A}|\mathsf{All}|\mathsf{Empty}$), if and only if the sequence of tags $\mathtt{g}_1 \cdots \mathtt{g}_m$ is in the language of the regular expression $Reg(\mathtt{f}_i)_{i=1,\ldots,n}$. It is worth to notice that each word generated by a regular type A is a document. The language of a type can be formally defined as follows.

Definition 4.11 (language). *Suppose* $A \stackrel{\triangle}{=} Reg(\mathtt{f}_i[A_i])_{i=1,\ldots,n}$. *The* language *of* A, $\mathcal{L}(A)$, *is defined as the set of documents matched by* $Reg(\mathtt{f}_i[A_i])_{i=1,\ldots,n}$.

As a consequence of this definition, $\mathcal{L}(\mathsf{Empty})$ contains only the empty document `()`, while $\mathcal{L}(\mathsf{All})$ corresponds to $\mathcal{D}$.

Types	T, S	$::=$	$\star$ *Resource*
		$\mid$	A *Document*
		$\mid$	$\mathsf{ref}\ \mathsf{A}$ *Reference*
		$\mid$	$\mathsf{node}\ \mathtt{f}(u)$ *Node location*
		$\mid$	$\mathsf{loc}\ \mathtt{f}(\mathsf{A})$ *Matching location*

Table 4.6: Syntax of types.

Based on Definition 4.11, we obtain a natural notion of subtyping $\mathsf{A} < \mathsf{B}$:

Definition 4.12 (subtyping). *Let A and B be two regular expression types (document types). A is a* subtype *of B, written $A < B$ if $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. We write $A \doteq B$ if $\mathcal{L}(A) = \mathcal{L}(B)$.*

In what follows, we write $\overline{\mathsf{A}}$ for some chosen regular expression type whose language is the complement of A. The type $\overline{\mathsf{A}}$ is unnecessary when $\mathsf{A} \doteq \mathsf{All}$, which means that we do not need to introduce a type with an empty language. In the case of type witness, we have $\mathsf{g}_1 \cdots \mathsf{g}_n \not\Vdash_{\mathsf{A}}$ if and only if there is a witness for $\overline{\mathsf{A}}$ of $\mathsf{g}_1 \cdots \mathsf{g}_n$.

Types for Astuce's terms

Astuce's type system relies on: document types, which are used in the typing of patterns, references and nodes, resource types and node types.

Definition 4.13 (types). *The set $\mathcal{T}$ of* types*, ranged over by T, S, ..., is defined by the syntax in Table 4.6.*

Apart from regular expression types A, used as types for document indexes, the type T of a process can also be the resource type $\star$, a constant type for terms that return no values; a reference type $\mathsf{ref}\ \mathsf{A}$; a node type $\mathsf{node}\ \mathtt{f}(u)$ for the type of a location holding an element $\mathtt{f}[u]$; or a matching type $\mathsf{loc}\ \mathtt{f}(\mathsf{A})$, that is the type of a location hosting the evaluation of a pattern, which matches document of type A, on the contents of an element tagged $\mathtt{f}$. In what follows, we consider that references can only hold document values: a reference is of type $\mathsf{ref}\ \mathsf{A}$ and not $\mathsf{ref}\ \mathsf{T}$.

4.3.2 Type checking

We define *contexts* $\Gamma, \Gamma', \ldots$ as finite partial maps from names $\mathcal{N}$ to types $\mathcal{T}$, usually denoted as sets of bindings of the form $\{x_i : \mathsf{T}_i,\ \imath_j : \mathsf{S}_j\}_{i \in I,\, j \in J}$, with x_i and $\imath_j$

distinct. We denote with $\mathrm{dom}(\Gamma)$ the domain of the context Γ and the empty context by $\emptyset$. In what follows, we write $\Gamma, x : \mathsf{T}$ for the context $\Gamma \cup \{x : \mathsf{T}\}$ if $x \notin \mathrm{dom}(\Gamma)$.

The typing rules are reported in Table 4.7. The type system is based on type judgments of the form $\Gamma \vdash P : \mathsf{T}$, meaning that the process P has type T under the hypothesis Γ. We assume that there is a given, fixed set of type declarations of the form $\mathsf{A} \stackrel{\triangle}{=} \mathit{Reg}(\mathtt{f}_i[\mathsf{A}_i])_{i=1,\ldots,n}$. We assume that functions and patterns are well-typed, which is denoted $f : \widetilde{\mathsf{T}} \to \mathsf{S}$ and $Q : \widetilde{\mathsf{T}} \to \mathsf{A}$. The types $\mathsf{T}_1, \ldots, \mathsf{T}_n$ in $\widetilde{\mathsf{T}}$ are the types of the parameters, while S is the type of the body of f and A is the type of the selector of Q. The type of a selector $\mathit{Reg}(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n}$ is obtained by substituting to each pattern Q_i in the selector its corresponding type A_i. Hence the type of Reg is equivalent to some type A such that $\mathsf{A} \stackrel{\triangle}{=} \mathit{Reg}(\mathtt{f}_i[\mathsf{A}_i])_{i=1,\ldots,n}$. In Remark 4.2 we discuss some rules that can be used, together with the rules in Table 4.7, for verifying well-typedness of functions and patterns.

The typing rules for the functional part of the calculus are standard; rule (T-Wt) deserves some explanations. Since a reference collects the sequence of values (documents) that are assigned to it, we check for every assignment of a value of type B into a reference of type $\mathsf{ref}\ \mathsf{A}$ that the relation $\mathsf{A}, \mathsf{B} < \mathsf{A}$ holds. This check allows us to enforce statically the type of references.

The remaining typing rules are for resources and pattern-matching operators. The type of an expression $\mathit{try}\ u\, Q(\widetilde{v})$ is $\mathsf{loc}\ \mathtt{root}(\mathsf{A})$ if the pattern Q matches documents of type A, rule (T-Try). Indeed the effect of this expression is to return a fresh location hosting the evaluation of Q on an element of the form $\mathtt{root}[u]$. Note the generic type B associated to document u. As we do not consider basic types nor basic values, $u : \mathsf{A}$, the type check would ensure that all matching in a well-typed process succeed and it makes no sense to proceed in the pattern evaluations. Pattern-matching, by definition, can fail, thus we simply require well-formedness of u, that is that there exists a regular type B such that $u : \mathsf{B}$. Rule (T-Wait) disciplines the typing of *wait* expressions. A *wait* is well typed only if it is blocking on a location of type $\mathsf{loc}\ \mathtt{f}(\mathsf{A})$, that is the location of a resource that can eventually turn into *ok* or $fail$. The important aspect of this rule is that, while the continuations e_1 and e_2 must have the same type, they are typed under different typing environments: the expression e_1 is typed with the hypothesis $x : \mathsf{A}$ while e_2 is typed with the hypothesis $x : \overline{\mathsf{A}}$. This leads to more precise types for filtering expressions (see Section 4.5.2).

The typing rules for locations are straightforward. Since a resource returns no value it has type $\star$. The rule for node location, (T-Lnd), states that a location containing *node* $\mathtt{f}(u)$ has only one possible type, namely $\mathsf{node}\ \mathtt{f}(u)$ itself. Hence this

$$\text{(T-Nm)}\ \frac{}{\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}} \qquad \text{(T-Fun)}\ \frac{f : (\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{S} \quad \Gamma \vdash u_i : \mathsf{T}_i \ \ i = 1, \ldots, n}{\Gamma \vdash f(u_1, \ldots, u_n) : \mathsf{S}}$$

$$\text{(T-Sub)}\ \frac{\mathsf{A} < \mathsf{B} \quad \Gamma \vdash P : \mathsf{A}}{\Gamma \vdash P : \mathsf{B}} \qquad \text{(T-Let)}\ \frac{\Gamma \vdash P : \mathsf{T} \quad \Gamma, x : \mathsf{T} \vdash R : \mathsf{S}}{\Gamma \vdash \mathit{let}\ x = P\ \mathit{in}\ R : \mathsf{S}}$$

$$\text{(T-Nd)}\ \frac{\Gamma \vdash u : \mathsf{A}}{\Gamma \vdash \mathtt{f}[u] : \mathtt{f}[\mathsf{A}]} \qquad \text{(T-Doc)}\ \frac{\Gamma \vdash \imath_k : \mathsf{node}\ \mathtt{f}_k(u_k) \quad \Gamma \vdash u_k : \mathsf{B}_k \ \ k = 1, \ldots, n}{\Gamma \vdash \imath_1 \cdots \imath_n : \mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n]}$$

$$\text{(T-Ref)}\ \frac{\Gamma \vdash u : \mathsf{A}}{\Gamma \vdash \mathit{newref}\ u : \mathsf{ref}\ \mathsf{A}} \quad \text{(T-Com)}\ \frac{\Gamma \vdash u_i : \mathsf{A}_i \quad i = 1, 2}{\Gamma \vdash u_1, u_2 : \mathsf{A}_1, \mathsf{A}_2} \quad \text{(T-Em)}\ \frac{}{\Gamma \vdash () : \mathsf{Empty}}$$

$$\text{(T-Rd)}\ \frac{\Gamma \vdash u : \mathsf{ref}\ \mathsf{A}}{\Gamma \vdash !u : \mathsf{A}} \qquad \text{(T-Wt)}\ \frac{\Gamma \vdash u : \mathsf{ref}\ \mathsf{A} \quad \Gamma \vdash v : \mathsf{B} \quad \mathsf{A}, \mathsf{B} < \mathsf{A}}{\Gamma \vdash u \mathrel{+}= v : \mathsf{Empty}}$$

$$\text{(T-Par)}\ \frac{\Gamma \vdash P : \mathsf{S} \quad \Gamma \vdash Q : \mathsf{T}}{\Gamma \vdash P \Rsh Q : \mathsf{T}} \qquad \text{(T-Res)}\ \frac{\begin{array}{c}\Gamma, \ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n \vdash P : \mathsf{T} \\ u = (\ell_1 \cdots \ell_n) \quad u \cap \mathrm{fn}(\Gamma) = \emptyset\end{array}}{\Gamma \vdash (\nu\, u)P : \mathsf{T}}$$

$$\text{(T-Try)}\ \frac{Q : (\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{A} \quad \Gamma \vdash v_i : \mathsf{T}_i \quad i = 1, \ldots, n \quad \Gamma \vdash u : \mathsf{B}}{\Gamma \vdash \mathit{try}\ u\, Q(v_1, \ldots, v_n) : \mathsf{loc}\ \mathtt{root}(\mathsf{A})}$$

$$\text{(T-Wait)}\ \frac{\Gamma \vdash u : \mathsf{loc}\ \mathtt{f}(\mathsf{A}) \quad \Gamma, x : \mathsf{A} \vdash e_1 : \mathsf{T} \quad \Gamma, x : \overline{\mathsf{A}} \vdash e_2 : \mathsf{T}}{\Gamma \vdash \mathit{wait}\ u(x)\, \mathit{then}\, e_1\, \mathit{else}\, e_2 : \mathsf{T}}$$

$$\text{(T-Lre)}\ \frac{\Gamma \vdash \ell : \mathsf{ref}\ \mathsf{A} \quad \Gamma \vdash u : A}{\Gamma \vdash \langle\, \ell \mapsto \mathit{ref}\ u \,\rangle : \star} \quad \text{(T-Lnd)}\ \frac{\Gamma \vdash \ell : \mathsf{node}\ \mathtt{f}(\imath_1 \cdots \imath_n)}{\Gamma \vdash \langle\, \ell \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle : \star}$$

$$\text{(T-Lok)}\ \frac{\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A}) \quad \Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \quad \Gamma \vdash u : \mathsf{A}}{\Gamma \vdash \langle\, \ell \mapsto \mathit{ok}\ \imath \,\rangle : \star}$$

$$\text{(T-Lfail)}\ \frac{\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A}) \quad \Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \quad \Gamma \vdash u : \overline{\mathsf{A}}}{\Gamma \vdash \langle\, \ell \mapsto \mathit{fail}\ \imath \,\rangle : \star}$$

$$\text{(T-Ltry)}\ \frac{\begin{array}{c}\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A}) \quad \Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \\ Q : (\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{A} \quad \Gamma \vdash v_i : \mathsf{T}_i \quad i = 1, \ldots, n\end{array}}{\Gamma \vdash \langle\, \ell \mapsto \mathit{try}\ \imath\, Q(\widetilde{v}) \,\rangle : \star}$$

$$\text{(T-Ltest)}\ \frac{\begin{array}{c}\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A}) \quad \Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \quad \Gamma \vdash \jmath_k : \mathsf{loc}\ \mathtt{f}_k(\mathsf{A}_k) \quad k = 1, \ldots, n \\ w = \jmath_1 \cdots \jmath_n \quad \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n \quad \Gamma \vdash v : \mathsf{ref}\ \mathsf{B} \quad \mathsf{B}, \mathsf{A} < \mathsf{B}\end{array}}{\Gamma \vdash \langle\, \ell \mapsto \mathit{test}\ \imath\ w\ v \,\rangle : \star}$$

Table 4.7: Typing rules.

rule avoids the presence of two *node* resources with the same location but containing different elements. Actually, we could extend our type system in a simpler way to ensure that well-typed configurations are also *well-formed*, that is cannot have two resources at the same location.

Definition 4.14 (well-formed configuration). *A configuration P is* well-formed *if for every location ℓ it contains at most one definition $\langle\, \ell \mapsto d \,\rangle$.*

It is easy to prove that well-formedness is preserved by structural congruence, substitutions and reductions (see [6] for details). By rule (T-LTRY), a location ℓ containing a *try* resource, evaluating a pattern Q of type A, is well typed if ℓ is of type $\mathsf{loc}\ \mathtt{f}(\mathsf{A})$ and the root tag of the evaluated document is $\mathtt{f}$. Note that no assumption is made on index $\imath_1 \cdots \imath_n$, which, as already said for rule (T-TRY), might well not be of type A. Finally, (T-LTEST) verifies that if a location ℓ is waiting for the sub-pattern evaluation, then there is a witness for A (the type of documents accepted by the pattern evaluated at ℓ) for $\mathtt{f}_1 \cdots \mathtt{f}_n$ (the word composed by the root tags of the children of the document at $\imath$ matched against the pattern). This is a necessary condition for having that document at $\imath$ complies with type A. Moreover, (T-LTEST) ensures that the type of the capture variable is a reference type and is compatible with the type A of the matched documents.

An important feature of our calculus is that every pattern is strongly typed: its type is the regular expression obtained by erasing capture variables. Likewise we can type locations, expressions and processes using a combination of regular expression types and ref types.

Remark 4.2 (well-typed patterns). From now onward, we assume all patterns and functions are well-typed. We extend now the type system in Table 4.7 with judgments for establish well-typedness of pattern and function definitions. The new typing rules are reported in Table 4.8.

The type of a selector *Reg* is obtained from *Reg* by substituting every pattern identifier Q_i with the corresponding type A_i, rule (T-SEL). Rule (T-PAT) checks if the definition $Q(x_1, \ldots, x_n) \stackrel{\triangle}{=} \mathit{let}\ z_1 = e'_1, \ldots, z_m = e'_m\ \mathit{in}\ \mathit{Reg}\, \mathit{as}\, x_k\, \mathit{then}\, e_1\, \mathit{else}\, e_2$ respects the declared type $(\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{A}$. Therefore, that upon receiving its actual parameters of type $\mathsf{T}_1, \ldots, \mathsf{T}_n$ and evaluating the expressions in the *let* part, pattern Q actually matches documents of type A. In particular it is checked that the type of the selector *Reg* is A, that continuations e_1 and e_2 are well typed, and that the type T_k associated to the capture variable x_k is compatible with A. Rule (T-FDEC) verifies if the definition $f \stackrel{\triangle}{=} e$ complies with the provided type $(\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{S}$ by

$$\text{(T-Sel)}\quad \frac{Reg = Reg(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\ldots,n} \quad \Gamma \vdash Q_i(\widetilde{x}_i) : (\widetilde{\mathsf{T}}_i) \to \mathsf{A}_i \quad i = 1,\ldots,n \qquad Reg(\mathtt{f}_i[\mathsf{A}_i])_{i=1,\ldots,n} \doteq \mathsf{A}}{\Gamma \vdash Reg : \mathsf{A}}$$

$$\text{(T-Pat)}\quad \frac{\begin{array}{c} Q(x_1,\ldots,x_n) \stackrel{\triangle}{=} let\ z_1 = e'_1,\ldots,z_m = e'_m\ in\ Reg\ as\ x_k\ then\ e_1\ else\ e_2 \\ \mathrm{fn}(Q(\widetilde{x})) = \emptyset \quad x_1 : \mathsf{T}_1,\ldots,x_n : \mathsf{T}_n \vdash e'_i : \mathsf{T}'_i \quad i = 1,\ldots,m \\ x_1 : \mathsf{T}_1,\ldots,x_n : \mathsf{T}_n, z_1 : \mathsf{T}'_1,\ldots,z_m : \mathsf{T}'_m \vdash Reg : \mathsf{A} \\ \mathsf{T}_k = \mathsf{ref\ B} \quad \mathsf{B}, \mathsf{A} < \mathsf{B} \\ x_1 : \mathsf{T}_1,\ldots,x_n : \mathsf{T}_n, z_1 : \mathsf{T}'_1,\ldots,z_m : \mathsf{T}'_m \vdash e_1 : \mathsf{S}_1 \\ x_1 : \mathsf{T}_1,\ldots,x_n : \mathsf{T}_n, z_1 : \mathsf{T}'_1,\ldots,z_m : \mathsf{T}'_m \vdash e_2 : \mathsf{S}_2 \end{array}}{\Gamma \vdash Q(x_1,\ldots,x_n) : (\mathsf{T}_1,\ldots,\mathsf{T}_n) \to \mathsf{A}}$$

$$\text{(T-Fdec)}\quad \frac{f \stackrel{\triangle}{=} e \quad \Gamma, x_1 : \mathsf{T}_1,\ldots,x_n : \mathsf{T}_n \vdash e : \mathsf{S}}{\Gamma \vdash f(x_1,\ldots,x_n) : (\mathsf{T}_1,\ldots,\mathsf{T}_n) \to \mathsf{S}}$$

Table 4.8: Typing rules for functions and patterns.

checking if the type of the body e is S when evaluated in a context where the formal parameters of f have associated types $\mathsf{T}_1,\ldots,\mathsf{T}_n$.

Example 4.3.2 (the reverse Web-Link Graph). We study the *reverse web-link graph* application [63], used e.g. in the Google's search-engine to compute page ranks. The goal is to build a list of all pages containing a link to a given URL. We consider a calculus enriched with an atomic type for strings, String, and a conditional construct $if - then - else$ to test equalities between strings, these extensions are straightforward to accommodate. Moreover, in pattern definitions, we sometimes use "$\mathsf{T}\ as\ x$" as a shortcut for the pattern invocation $Q(x)$, where $Q(x) \stackrel{\triangle}{=} All\ as\ x$ and T is the expected document type. We assume that web pages in the index are stored as documents of type $\mathsf{WP} = \mathtt{pg}[\mathsf{B}]$, where B is the type ($\mathtt{url}[\mathsf{String}]$, $\mathtt{link}[\mathsf{URL}*]$, $\mathtt{text}[\mathsf{String}]$) and URL is a shorthand for $\mathtt{url}[\mathsf{String}]$, meaning that for each page we have its location (`url`), a list of its hyperlinks (`link`) and its textual content (`text`). For simplicity, assume that each list contains no duplicate hyperlinks. The following patterns are used for building a reverse web-link graph:

$$
\begin{array}{rcl}
revWL(t,r) & \triangleq & \big(\mathtt{pg}[revWL'(t,r)]\big)* \\
revWL'(t,r) & \triangleq & let\ x = newref\ ()\ ,\ y = newref\ ()\ in \\
 & & \big(\mathtt{url}[\mathsf{String}\, as\, x], \mathtt{link}[\mathsf{URL}* as\, y], \mathtt{text}[\mathsf{String}]\big) \\
 & & then\big(try\ !y\ sift(t, !x, r)\big) \\
sift(t,t',r) & \triangleq & \big(\mathtt{url}[sift'(t,t',r)]\big)* \\
sift'(t,t',r) & \triangleq & let\ z = newref\ ()\ in\ \big(\mathsf{String}\, as\, z\big) \\
 & & then\big(if\ z = t\ then\ r \mathrel{+}= \mathtt{url}[t']\big)\ .
\end{array}
$$

The main pattern is $revWL(t,r)$, where t is the string representing the target URL, and r is a (global) reference cell for t's reverse-index. *revWL* visits each indexed page and invokes $revWL'$, which extracts the page's location ($\mathsf{String}\, as\, x$) and list of links ($\mathsf{URL}* as\, y$), and stores them in two fresh references x and y. Then the pattern *sift* is used to test whether the list of URL in y contains the target location t. If true, the result r is updated by adding to it the value of x (that is passed as the second parameter of *sift*). In each pattern, the "location" parameters t and t' have type String while the final result, held in the parameter r, is a reference holding values of type $\mathsf{URL}*$. Hence the pattern *revWL* has type $(\mathsf{String}, \mathsf{ref}\ (\mathsf{URL}*)) \to \mathsf{WP}*$ and *sift* has type $(\mathsf{String}, \mathsf{String}, \mathsf{ref}\ (\mathsf{URL}*)) \to \mathsf{URL}*$. Assume $\imath_1 \cdots \imath_n$ is the index of the web pages of interest, possibly stored in different physical locations, we can create a reverse index for the target location *ta* with the expression: $let\ z = newref\ ()\ in\ try\ \imath_1 \cdots \imath_n\ revWL(ta, z)$. Note that patterns and functions are evaluated locally at each site, while the result reference z is "global" (it is local to the caller, but is accessed by every site for storing the results.)

4.4 Properties of typing

As usual, it is useful to guarantee that properties ensured by the type system are preserved at run-time. A subject reduction property is always used for guaranteeing that well-typedness is preserved by reductions. Before introducing this result, we need to prove a few preliminary lemmata. Firstly, we prove that structural congruence and substitution preserve well-typedness; after, we introduce some interesting properties of (unambiguous) patterns and languages.

Lemma 4.1 (weakening). *If* $\Gamma, x : \mathsf{T} \vdash P : \mathsf{T}'$ *and* $x \notin \mathrm{fn}(P)$ *then* $\Gamma \vdash P : \mathsf{T}'$ *and vice versa.*

Proof: The proof is straightforward by induction on the derivation of $\Gamma, x : \mathsf{T} \vdash P : \mathsf{T}'$. □

Proposition 4.1 (subject congruence). *If $P \equiv Q$ and $\Gamma \vdash P : T$ then $\Gamma \vdash Q : T$.*

PROOF: The proof is straightforward by induction on the derivation of $P \equiv Q$; the proof proceeds by distinguishing the last structural rule applied and relies on Lemma 4.1 (weakening). □

Proposition 4.2 (substitution). *If $\Gamma, x : T \vdash P : T'$ and $\Gamma \vdash u : T$ then $\Gamma \vdash P[u/x] : T'$.*

PROOF: The proof is straightforward by induction on the derivation of $\Gamma, x : \mathsf{T} \vdash P : \mathsf{T}'$ and proceeds by distinguishing the last typing rule applied in the derivation. The base case is rule (T-NM), which can be easily proved by using the hypothesis $\Gamma \vdash u : \mathsf{T} = \mathsf{T}'$. In the other cases the proof proceeds by inductive hypothesis. □

Given a regular expression type A, its language contains the elements (documents) $\mathtt{a}, \mathtt{b}, \ldots$ generated by the grammar: $\mathtt{a} ::= \mathtt{()} \mid \mathtt{f[a]} \mid \mathtt{aa}$. It is important to note that by substituting the sequence "$\mathtt{aa}$" with "$\mathtt{a,a}$" and $\mathtt{()}$ with Empty, we obtain a grammar for regular types that does not contain choices, "$\mid$", nor iterations, "$*$". With a slight abuse of notation, in what follows we use $\mathtt{a}, \mathtt{b}, \ldots$ for indicating both elements generated by the previous grammar and the corresponding types.

Lemma 4.2. *If $\mathtt{f}_1[a_1] \cdots \mathtt{f}_n[a_n] \in \mathcal{L}(A)$, with $n > 0$, then $A \doteq \mathtt{f}_1[a_1], \ldots, \mathtt{f}_n[a_n] \mid A$. If $\mathtt{()} \in \mathcal{L}(A)$ then $A \doteq \mathsf{Empty} \mid A$.*

PROOF: The result follows by Definition 4.11 (language) and by observing that $\mathcal{L}(\mathtt{f}_1[\mathtt{a}_1], \ldots, \mathtt{f}_n[\mathtt{a}_n]) = \{\mathtt{f}_1[\mathtt{a}_1] \cdots \mathtt{f}_n[\mathtt{a}_n]\} \subseteq \mathcal{L}(\mathsf{A})$ and $\mathcal{L}(\mathsf{Empty}) = \{\mathtt{()}\} \subseteq \mathcal{L}(\mathsf{A})$. □

The following lemma has been introduced for giving evidence to some connections between the relation of witness and subtyping.

Lemma 4.3. *Assume A is a regular type.*

(1) *If $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_A A_1 \cdots A_n$, with $n > 0$, then $\mathtt{f}_1[A_1], \ldots, \mathtt{f}_n[A_n] < A$. If $\mathtt{()} \vdash_A \mathsf{Empty}$ then $\mathsf{Empty} < A$.*

(2) *If $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_A$ then for each $B_1, \ldots, B_n$ regular types it holds that $\mathtt{f}_1[B_1], \ldots, \mathtt{f}_n[B_n] < \overline{A}$.*

PROOF:

(1) By definition of (type) witness.

(2) Suppose $\mathsf{A} = \mathit{Reg}(\mathtt{g}_j[\mathsf{A}_j])_{j=1,\ldots,k}$. By definition, $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{\mathsf{A}}$ implies $\mathtt{f}_1 \cdots \mathtt{f}_n \notin \mathit{Reg}(\mathtt{g}_j)_{j=1,\ldots,k}$. Hence, by Definition 4.11 (language), for each $\mathtt{b}_i \in \mathcal{L}(\mathsf{B}_i)$, with $i = 1, \ldots, n$, we have $\mathtt{f}_1[\mathtt{b}_1] \cdots \mathtt{f}_n[\mathtt{b}_n] \notin \mathcal{L}(\mathsf{A})$, that is $\mathtt{f}_1[\mathtt{b}_1] \cdots \mathtt{f}_n[\mathtt{b}_n] \in \mathcal{L}(\overline{\mathsf{A}})$ by definition of $\bar{\cdot}$. By Definition 4.11 (language) and 4.12 (subtyping), $\mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n] < \overline{\mathsf{A}}$.

□

In the following we define the function $[\![u]\!]_\Gamma$ with parameters an index u and a context Γ. $[\![u]\!]_\Gamma$ can be used for deducing, according to the assumptions in Γ, the element a that is the only word in the language associated to the "minimal type" of u. Note that $[\![u]\!]_\Gamma$ belongs to the language associated to each type complying with u.

Definition 4.15 ($[\![u]\!]_\Gamma$).
$[\![()]\!]_\Gamma = ()$
$[\![\imath_1 \cdots \imath_n]\!]_\Gamma = \mathtt{f}_1[\mathsf{a}_1] \cdots \mathtt{f}_n[\mathsf{a}_n]$ *if* $\Gamma \vdash \imath_i : \mathsf{node}\,\mathtt{f}_i(u_i)$ *and* $[\![u_i]\!]_\Gamma = \mathsf{a}_i$.

Lemma 4.4. *Assume A is a regular type and let be $u = \imath_1 \cdots \imath_n$, with $n \geqslant 0$. $\Gamma \vdash u : A \Leftrightarrow [\![u]\!]_\Gamma \in \mathcal{L}(A)$.*

Proof:

($\Rightarrow$): By induction on the depth, d, of the document u:

$d = 0$**:** $u = ()$ and $[\![()]\!]_\Gamma = ()$. $\Gamma \vdash () : \mathsf{Empty}$, (T-Em), and $() \in \mathcal{L}(\mathsf{Empty})$.

$d = m + 1$**:** $u = \imath_1 \cdots \imath_n$. We distinguish two cases depending on the last rule applied for deducing $\Gamma \vdash u : \mathsf{A}$

(T-Doc): $\mathsf{A} = \mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n]$ and for each $k = 1, \ldots, n$ it holds that $\Gamma \vdash \imath_k : \mathsf{node}\,\mathtt{f}_k(u_k)$ and $\Gamma \vdash u_k : \mathsf{B}_k$. Moreover, each u_k has depth smaller than d, hence by inductive hypothesis $[\![u_k]\!]_\Gamma \in \mathcal{L}(\mathsf{B}_k)$.

(T-Sub):

- $\Gamma \vdash u : \mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n]$, that is $\Gamma \vdash \imath_k : \mathsf{node}\,\mathtt{f}_k(u_k)$, and $\Gamma \vdash u_k : \mathsf{B}_k$. As in the previous point, each u_k has depth smaller than d, hence by inductive hypothesis, $[\![u_k]\!]_\Gamma \in \mathcal{L}(\mathsf{B}_k)$.
- $\mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n] < \mathsf{A}$ implies $\mathcal{L}(\mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n]) \subseteq \mathcal{L}(\mathsf{A})$.

In both cases, $[\![u]\!]_\Gamma = \mathtt{f}_1[[\![u_1]\!]_\Gamma] \cdots \mathtt{f}_n[[\![u_n]\!]_\Gamma] \in \mathcal{L}(\mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n])$, thus $[\![u]\!]_\Gamma \in \mathcal{L}(\mathsf{A})$.

($\Leftarrow$): By induction on the depth, d, of the document u:

$d = 0$**:** $u = ()$ and $[\![()]\!]_\Gamma = ()$. $() \in \mathcal{L}(\mathsf{A})$ and $\Gamma \vdash () : \mathsf{Empty}$, (T-Em). $() \in \mathcal{L}(\mathsf{A})$ implies, by Lemma 4.2, $\mathsf{A} \doteq \mathsf{Empty} \mid \mathsf{A}$, hence $\mathsf{Empty} < \mathsf{A}$ and, by (T-Sub), $\Gamma \vdash () : \mathsf{A}$.

$d = m + 1$**:** $u = \imath_1 \cdots \imath_n$. $[\![u]\!]_\Gamma = \mathtt{f}_1[\mathsf{a}_1] \cdots \mathtt{f}_n[\mathsf{a}_n] \in \mathcal{L}(\mathsf{A})$ and, by Definition 4.15, $\mathsf{a}_i = [\![u_i]\!]_\Gamma$, and $\Gamma \vdash \imath_i : \mathsf{node}\,\mathtt{f}_i(u_i)$ for $i = 1, \ldots, n$. Obviously, for each $i = 1, \ldots, n$ it holds that $\mathsf{a}_i \in \mathcal{L}(\mathsf{a}_i)$ and by induction $\Gamma \vdash u_i : \mathsf{a}_i$. $\mathtt{f}_1[\mathsf{a}_1] \cdots \mathtt{f}_n[\mathsf{a}_n] \in \mathcal{L}(\mathsf{A})$ implies, by Lemma 4.2, $\mathsf{A} \doteq \mathtt{f}_1[\mathsf{a}_1], \ldots, \mathtt{f}_n[\mathsf{a}_n] \mid \mathsf{A}$. Therefore, $\mathtt{f}_1[\mathsf{a}_1], \ldots, \mathtt{f}_n[\mathsf{a}_n] < \mathsf{A}$ and, by rules (T-Doc) and (T-Sub), $\Gamma \vdash u : \mathsf{A}$.

□

The following results are useful for ensuring that pattern-matching never produces typing errors. In other words, a document matching a pattern always complies with its type, and vice versa, when the pattern-matching fail the document complies with the complement of the type associated to the pattern.

Lemma 4.5. *Let be $Q(\widetilde{x})$ a pattern with selector $Reg = Reg(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\dots,k}$ and suppose $\Gamma \vdash Q(\widetilde{x}) : (\widetilde{T}) \to A$. If $\Gamma \vdash \widetilde{v} : \widetilde{T}$, $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg[\widetilde{v}/\widetilde{x}]} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n)$, with $n > 0$ and $\Gamma \vdash Q_i(\widetilde{v}_i) : A_i$, for $i = 1, \dots, n$, then $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_A A_1 \cdots A_n$. If $() \vdash_{Reg[\widetilde{v}/\widetilde{x}]}$ Empty then $() \vdash_A$ Empty.*

Proof: Suppose $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg[\widetilde{v}/\widetilde{x}]} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n)$, with $n > 0$. By the well-typedness of $Q(\widetilde{x})$, (T-Pat) and (T-Sel): $\widetilde{x} : \widetilde{\mathsf{T}} \vdash Reg(\mathtt{f}_i[Q_i(\widetilde{x}_i)])_{i=1,\dots,k} : \mathsf{A}$ and there are $\mathsf{A}_1, \dots, \mathsf{A}_k$ such that for each $i = 1, \dots, k$ it holds that $\widetilde{x} : \widetilde{\mathsf{T}} \vdash Q_i(\widetilde{x}_i) : (\widetilde{\mathsf{T}}_i) \to \mathsf{A}_i$ and $Reg(\mathtt{f}_i[\mathsf{A}_i])_{i=1,\dots,k} \doteq \mathsf{A}$. By $\Gamma \vdash \widetilde{v} : \widetilde{\mathsf{T}}$ and Proposition 4.2 (substitution), $Reg[\widetilde{v}/\widetilde{x}] : \mathsf{A}$ and $Q_i(\widetilde{v}_i) : (\widetilde{\mathsf{T}}_i) \to \mathsf{A}_i$. By definition of witness, $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n)$ implies that $\mathtt{f}_1 \cdots \mathtt{f}_n \in Reg(\mathtt{f}_i)_{i=1,\dots,k}$. From this and $Q_i(\widetilde{v}_i) : (\widetilde{\mathsf{T}}_i) \to \mathsf{A}_i$ it follows that $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n$.

Suppose $() \vdash_{Reg[\widetilde{v}/\widetilde{x}]}$ *Empty*. This means that the empty document is matched by $Q(\widetilde{x})$, hence is in the language of A and $() \vdash_{\mathsf{A}} \mathsf{Empty}$ by definition. □

Lemma 4.6. *Suppose A unambiguous and $A \neq$ All. $\mathtt{f}_1 \cdots \mathtt{f}_j \cdots \mathtt{f}_n \vdash_A A_1 \cdots A_j \cdots A_n$ implies $\mathtt{f}_1[A_1], \dots, \mathtt{f}_j[\overline{A}_j], \dots, \mathtt{f}_n[A_n] < \overline{A}$.*

Proof: By Lemma 4.3 (1), $\mathtt{f}_1 \cdots \mathtt{f}_j \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_j \cdots \mathsf{A}_n$ implies $\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\mathsf{A}_j], \dots, \mathtt{f}_n[\mathsf{A}_n] < \mathsf{A}$, that is $\mathcal{L}(\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\mathsf{A}_j], \dots, \mathtt{f}_n[\mathsf{A}_n]) \subseteq \mathcal{L}(\mathsf{A})$ (by Definition 4.12 (subtyping)). By definition of $\bar{\cdot}$, $\mathcal{L}(\overline{\mathsf{A}}_j) = \overline{\mathcal{L}(\mathsf{A}_j)}$, hence $\mathcal{L}(\overline{\mathsf{A}}_j) \cap \mathcal{L}(\mathsf{A}_j) = \emptyset$. This implies that $\mathcal{L}(\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\overline{\mathsf{A}}_j], \dots, \mathtt{f}_n[\mathsf{A}_n]) \cap \mathcal{L}(\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\mathsf{A}_j], \dots, \mathtt{f}_n[\mathsf{A}_n]) = \emptyset$. From the unambiguity of A, it follows that $\mathcal{L}(\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\overline{\mathsf{A}}_j], \dots, \mathtt{f}_n[\mathsf{A}_n]) \cap \mathcal{L}(\mathsf{A}) = \emptyset$. Therefore, $\mathcal{L}(\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\overline{\mathsf{A}}_j], \dots, \mathtt{f}_n[\mathsf{A}_n]) \subseteq \mathcal{L}(\overline{\mathsf{A}})$ and, again by Definition 4.12 (subtyping), $\mathtt{f}_1[\mathsf{A}_1], \dots, \mathtt{f}_j[\overline{\mathsf{A}}_j], \dots, \mathtt{f}_n[\mathsf{A}_n] < \overline{\mathsf{A}}$. □

Lemma 4.7. *Let $Q(\widetilde{x})$ be a unambiguous pattern with selector Reg and suppose $\Gamma \vdash Q(\widetilde{x}) : (\widetilde{T}) \to A$, $\Gamma \vdash \imath_i : \mathsf{node}\, \mathtt{f}_i(u_i)$, for $n \geqslant 0$, $i = 1, \dots, n$, and $\Gamma \vdash \widetilde{v} : \widetilde{T}$. If $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{Reg[\widetilde{v}/\widetilde{x}]}$ then $\Gamma \vdash \imath_1 \cdots \imath_n : \overline{A}$.*

Proof: Suppose $Reg = Reg(\mathtt{f'}_i[Q_i(\widetilde{x}_i)])_{i=1,\dots,k}$ and $n > 0$. From the well-typedness of $Q(\widetilde{x})$, (T-Pat) and (T-Sel): $\widetilde{x} : \widetilde{\mathsf{T}} \vdash Reg : \mathsf{A}$, $\widetilde{x} : \widetilde{\mathsf{T}} \vdash Q_i(\widetilde{x}_i) : (\widetilde{\mathsf{T}}_i) \to \mathsf{A}_i$ and $\mathsf{A} \doteq Reg(\mathtt{f'}_i[\mathsf{A}_i])_{i=1,\dots,k}$. By definition of witness, $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{Reg[\widetilde{v}/\widetilde{x}]}$ means that $\mathtt{f}_1 \cdots \mathtt{f}_n \notin Reg(\mathtt{f'}_i)_{i=1,\dots,k}$, hence, by definition, $\mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{\mathsf{A}}$. By Lemma 4.3 (2), for

each B_i it holds that $\mathtt{f}_1[\mathsf{B}_1], \ldots, \mathtt{f}_n[\mathsf{B}_n] < \overline{\mathsf{A}}$. Thus, $\mathtt{f}_1[[\![u_1]\!]_\Gamma], \ldots, \mathtt{f}_n[[\![u_n]\!]_\Gamma] < \overline{\mathsf{A}}$ and $[\![\imath_1 \cdots \imath_n]\!]_\Gamma \in \mathcal{L}(\mathtt{f}_1[[\![u_1]\!]_\Gamma], \ldots, \mathtt{f}_n[[\![u_n]\!]_\Gamma]) \subseteq \mathcal{L}(\overline{\mathsf{A}})$, hence, by Lemma 4.4, $\Gamma \vdash \imath_1 \cdots \imath_n : \overline{\mathsf{A}}$.

Suppose $n = 0$. $() \not\models_{Reg[\widetilde{v}/\widetilde{x}]}$ means that $()$ is not matched by $Q(\widetilde{x})$, hence is not in the language of A and, by definition of $\bar{\cdot}$, $() \in \mathcal{L}(\overline{\mathsf{A}})$ and $\Gamma \vdash () : \overline{\mathsf{A}}$. □

We are now ready to prove that well-typedness is preserved by reductions. The proof is quite involved since it is not possible to reason on a whole document at once: its content is scattered across distinct resource locations. This complexity reflects actual restrictions imposed when working with distributed documents, e.g. that they can never be checked locally.

Theorem 4.1 (subject reduction). *Suppose that P is well formed and contains only unambiguous patterns and T contains only unambiguous types. If $\Gamma \vdash P : \mathsf{T}$ and $P \to R$ then $\Gamma \vdash R : \mathsf{T}$.*

PROOF: The proof proceeds by induction on the derivation of $P \to R$; we distinguish the last reduction rule applied. We first consider the most interesting cases:

(MTC): by rules (T-PAR), (T-TRY), and (T-LND)

$$\Gamma \vdash \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \Rsh \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle : \star$$

implies $\Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \ldots \imath_n)$, $\Gamma \vdash \imath_k : \mathsf{node}\ \mathtt{f}_k(w_k)$, $\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A})$, $\Gamma \vdash Q(\widetilde{x}) : (\widetilde{\mathsf{T}}) \to \mathsf{A}$ and $\Gamma \vdash \widetilde{v} : \widetilde{\mathsf{T}}$.

By (MTC), $P \to R$ with $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n)$ and

$$R = \textstyle\prod_{k=1\ldots n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \Rsh \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh (\nu\, \jmath_1 \cdots \jmath_n)(\textstyle\prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto try\ \imath_k\, Q_k(\widetilde{v}_k) \,\rangle \Rsh \langle\, \ell \mapsto test\ \imath\, \jmath_1 \cdots \jmath_n\ v_r \,\rangle)$$

if v_r is the capture reference of $Q(\widetilde{v})$.

Suppose $\Gamma \vdash Q_k(\widetilde{x}_k) : (\widetilde{\mathsf{T}}_k) \to \mathsf{A}_k$, then $\Gamma, \jmath_k : \mathsf{loc}\ \mathtt{f}_k(\mathsf{A}_k)_{k=1,\ldots,n} \vdash \langle\, \jmath_k \mapsto try\ \imath_k\, Q_k(\widetilde{v}_k) \,\rangle : \star$ for each $k = 1, \ldots, n$, (T-LTRY).

We have already said that: $\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A})$ and $\Gamma \vdash \imath : \mathsf{node}\ \mathtt{f}(\imath_1 \ldots \imath_n)$. By (T-PAT), $\mathsf{T}_r = \mathsf{ref}\ \mathsf{B}$ and $\mathsf{B}, \mathsf{A} < \mathsf{B}$. Moreover, all premises of Lemma 4.5 are satisfied, hence $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n$. By (T-LTEST), $\Gamma, \jmath_k : \mathsf{loc}\ \mathtt{f}_k(\mathsf{A}_k)_{k=1,\ldots,n} \vdash \langle\, \ell \mapsto test\ \imath\, \jmath_1 \cdots \jmath_n\ v_r \,\rangle : \star$ and $\Gamma \vdash Q : \star$, by (T-RES) and (T-PAR).

(ERR): $\langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \to \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k) \,\rangle \Rsh \langle\, \ell \mapsto fail\ \imath \,\rangle$ with $Q(\widetilde{v}) \stackrel{\triangle}{=} Reg\ as\ v_r$ and $\mathtt{f}_1 \cdots \mathtt{f}_n \not\models_{Reg}$.

By (T-LTRY), $\Gamma \vdash \ell : \mathsf{loc}\,\mathtt{f}(\mathsf{A})$, $\Gamma \vdash \widetilde{v} : \widetilde{\mathsf{T}}$ and $\Gamma \vdash Q(\widetilde{x}) : (\widetilde{\mathsf{T}}) \to \mathsf{A}$. Moreover, $\Gamma \vdash \imath_k : \mathsf{node}\,\mathtt{f}_k(w_k)$, by (T-LND). By Lemma 4.7, $\Gamma \vdash \imath_1 \cdots \imath_n : \overline{\mathsf{A}}$ and by (T-LFAIL), $\Gamma \vdash \langle\, \ell \mapsto \mathit{fail}\ \imath \,\rangle : \star$. The result follows by applying rule (T-PAR);

(TEST-OK): $\langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto \mathit{ok}\ \imath_k \,\rangle \Rsh \langle\, \ell \mapsto \mathit{test}\ \imath\ \jmath_1 \cdots \jmath_n\ v_r \,\rangle \to \langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto \mathit{ok}\ \imath_k \,\rangle \Rsh \mathit{let}\ \ z\ =\ (v_r\ +\!\!=\ \imath_1 \cdots \imath_n)\ \mathit{in}\ \langle\, \ell \mapsto \mathit{ok}\ \imath \,\rangle$ (with $z \neq \imath, \ell$).
$\Gamma \vdash \langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto \mathit{ok}\ \imath_k \,\rangle \Rsh \langle\, \ell \mapsto \mathit{test}\ \imath\ \jmath_1 \cdots \jmath_n\ v_r \,\rangle : \star$ implies by (T-LND), $\Gamma \vdash \imath : \mathsf{node}\,\mathtt{f}(\imath_1 \cdots \imath_n)$ and

- by (T-LTEST), $\Gamma \vdash \ell : \mathsf{loc}\ \mathtt{f}(\mathsf{A})$, $\Gamma \vdash \jmath_k : \mathsf{loc}\ \mathtt{f}_k(\mathsf{A}_k)$, $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n$, $\Gamma \vdash v_r : \mathsf{T}_r = \mathsf{ref}\ \mathsf{B}$ and $\mathsf{B}, \mathsf{A} < \mathsf{B}$;
- by (T-LOK), $\forall k = 1, \ldots, n$: it holds that $\Gamma \vdash \imath_k : \mathsf{node}\,\mathtt{f}_k(u_k)$ and $\Gamma \vdash u_k : \mathsf{A}_k$;

By Lemma 4.3 (1), $\mathtt{f}_1[\mathsf{A}_1], \ldots, \mathtt{f}_n[\mathsf{A}_n] < \mathsf{A}$. By (T-DOC), $\Gamma \vdash \imath_1 \cdots \imath_n : \mathtt{f}_1[\mathsf{A}_1], \ldots, \mathtt{f}_n[\mathsf{A}_n]$ and, by (T-SUB), $\Gamma \vdash \imath_1 \cdots \imath_n : \mathsf{A}$. Thus, by $z \neq \imath, \ell$, (T-WT), (T-LET) and (T-PAR), $\Gamma \vdash \langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto \mathit{ok}\ \imath_k \,\rangle \Rsh \mathit{let}\ \ z\ =\ (v_r\ +\!\!=\ \imath_1 \cdots \imath_n)\ \mathit{in}\ \langle\, \ell \mapsto \mathit{ok}\ \imath \,\rangle : \star$.

(TEST-FAIL): in $P = \langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto d_k \,\rangle \Rsh \langle\, \ell \mapsto \mathit{test}\ \imath\ \jmath_1 \cdots \jmath_n\ v_r \,\rangle$ there is a k such that $\langle\, \jmath_k \mapsto \mathit{fail}\ \imath_k \,\rangle$. Suppose that for each $j = 1, \ldots, n$, with $j \neq k$, $\langle\, \jmath_j \mapsto \mathit{ok}\ \imath_j \,\rangle$ (similar proof in the other cases). Hence, $\Gamma \vdash P : \star$ implies:

- by (T-LFAIL): $\Gamma \vdash \jmath_k : \mathsf{loc}\,\mathtt{f}_k(\mathsf{A}_k)$, $\Gamma \vdash \imath_k : \mathsf{node}\,\mathtt{f}_k(\widetilde{u}_k)$, and $\Gamma \vdash u_k : \overline{\mathsf{A}}_k$;
- by (T-LTEST): $\Gamma \vdash \ell : \mathsf{loc}\,\mathtt{f}(\mathsf{A})$ and $\mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n$.

$A \neq \mathsf{All}$, because otherwise the matching cannot fail (by definition of witness it would be $\mathsf{A}_k = \mathsf{All}$ and $\mathcal{L}(\mathsf{All}) = \mathcal{D}$). Hence, by Lemma 4.6, $\mathtt{f}_1[\mathsf{A}_1], \ldots, \mathtt{f}_k[\overline{\mathsf{A}}_k], \ldots, \mathtt{f}_n[\mathsf{A}_n] < \overline{\mathsf{A}}$. By rule (T-DOC), $\Gamma \vdash \imath_1 \cdots \imath_n : \mathtt{f}_1[\mathsf{A}_1], \ldots, \mathtt{f}_k[\overline{\mathsf{A}}_k], \ldots, \mathtt{f}_n[\mathsf{A}_n]$ and by (T-SUB), $\Gamma \vdash \imath_1 \cdots \imath_n : \overline{\mathsf{A}}$. By (T-LND) and $\Gamma \vdash P : \star$, $\Gamma \vdash \imath : \mathsf{node}\,\mathtt{f}(\imath_1 \cdots \imath_n)$, thus, by (T-LFAIL) and (T-PAR), $\Gamma \vdash \langle\, \imath \mapsto \mathit{node}\ \mathtt{f}(\imath_1 \cdots \imath_n) \,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto d_k \,\rangle \Rsh \langle\, \ell \mapsto \mathit{fail}\ \imath \,\rangle : \star$.

(WAIT-OK), (WAIT-FAIL): in these cases the proof follows by observing that $\Gamma \vdash \ell : \mathsf{loc}\,\mathtt{f}(A)$, $\Gamma \vdash \imath : \mathsf{node}\,\mathtt{f}(u)$ and $\Gamma \vdash \langle\, \ell \mapsto \mathit{ok}\ \imath \,\rangle : \star$ imply $\Gamma \vdash u : A$, (T-LOK) (and similarly, $\Gamma \vdash \langle\, \ell \mapsto \mathit{fail}\ \imath \,\rangle : \star$ imply $\Gamma \vdash u : \overline{A}$, (T-LFAIL)) and by applying Proposition 4.2 (substitution).

(FUN): $f(\widetilde{u}) \to e[\widetilde{u}/\widetilde{x}]$. By (T-FUN) $\Gamma \vdash f(\widetilde{u}) : \mathsf{T}_0$ implies $\Gamma \vdash f : (\widetilde{\mathsf{T}}) \to \mathsf{T}_0$ and $\Gamma \vdash u_i : \mathsf{T}_i$. By (T-FDEC), $\Gamma \vdash f : (\widetilde{\mathsf{T}}) \to \mathsf{T}_0$ means that $f(\widetilde{x}) \stackrel{\triangle}{=} e$ and $\Gamma, x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n \vdash e : \mathsf{T}_0$. The result follows by applying Proposition 4.2 (substitution): $\Gamma \vdash e[\widetilde{u}/\widetilde{x}] : \mathsf{T}_0$.

(REF): $newref\ \ u \to (\nu\,\ell)(\langle\, \ell \mapsto ref\ u \,\rangle \Rsh \ell)$. By (T-REF), $\Gamma \vdash newref\ u : \mathsf{ref}\ \mathsf{A}$ implies $\Gamma \vdash u : \mathsf{A}$. The result follows by considering $\ell : \mathsf{ref}\ \mathsf{A}$ and by applying (T-RES), (T-PAR), (T-LRE) and (T-NM).

(LET): $let\ x = u\ in\ \ P \to P[u/x]$. By (T-LET), $\Gamma \vdash let\ x = u\ in\ \ P : \mathsf{T}'$ implies $\Gamma \vdash u : \mathsf{T}$ and $\Gamma, x : \mathsf{T} \vdash P : \mathsf{T}'$ and, by Proposition 4.2 (substitution), $E \vdash P[u/x] : \mathsf{T}'$.

(WR): $\langle\, \ell \mapsto ref\ u \,\rangle \Rsh l \mathrel{+}= v \ \to\ \langle\, \ell \mapsto ref\ u, v \,\rangle \Rsh ()$. By (T-PAR), (T-LRE) and (T-WT), $\Gamma \vdash \langle\, \ell \mapsto ref\ u \,\rangle \Rsh l \mathrel{+}= v : \mathsf{Empty}$ implies $\Gamma \vdash \ell : \mathsf{ref}\ \mathsf{A}$, $\Gamma \vdash u : \mathsf{A}$, $\Gamma \vdash v : \mathsf{B}$ and $\mathsf{A}, \mathsf{B} < \mathsf{A}$. By (T-COM), $\Gamma \vdash u, v : \mathsf{A}, \mathsf{B}$ and, by (T-SUB), $\Gamma \vdash u, v : \mathsf{A}$. Finally, by (T-LRE), (T-EM) and (T-PAR), $\Gamma \vdash \langle\, \ell \mapsto ref\ u, v \,\rangle \Rsh () : \mathsf{Empty}$.

(STR): the proof relies on Proposition 4.1 (subject congruence) and proceeds by applying the inductive hypothesis.

(RD) $\langle\, \ell \mapsto ref\ u \,\rangle \Rsh !\ell \ \to\ \langle\, \ell \mapsto ref\ u \,\rangle \Rsh u$. By (T-LRE) and (T-RD) $\Gamma \vdash \langle\, \ell \mapsto ref\ u \,\rangle \Rsh !\ell : \mathsf{A}$ implies $\Gamma \vdash \ell : \mathsf{ref}\ \mathsf{A}$ and $\Gamma \vdash u : \mathsf{A}$. Hence, $\Gamma \vdash \langle\, \ell \mapsto ref\ u \,\rangle \Rsh u : \mathsf{A}$ by (T-PAR), (T-LRE) and (T-NM).

(CTX): the proof is straightforward on the structure of the context E and by applying in each case the inductive hypothesis.

(ND): $\mathtt{f}[u] \to (\nu\imath)(\langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle \Rsh \imath)$. By (T-ND), $\Gamma \vdash \mathtt{f}[u] : \mathtt{f}[\mathsf{A}]$ implies $\Gamma \vdash u : \mathsf{A}$. By (T-NM), $\Gamma, \imath : \mathsf{node}\ \mathtt{f}(u) \vdash \imath : \mathsf{node}\,\mathtt{f}(u)$ and, by (T-DOC), $\Gamma, \imath : \mathsf{node}\ \mathtt{f}(u) \vdash \imath : \mathtt{f}[\mathsf{A}]$. Thus, by (T-PAR) and (T-RES), $\Gamma \vdash (\nu\,\imath)(\langle\, \imath \mapsto node\ \mathtt{f}(u) \,\rangle \Rsh \imath) : \mathtt{f}[\mathsf{A}]$.

(COMP): $u_1, u_2 \to \imath_1 \cdots \imath_n$ if $u_1 = \imath_1 \cdots \imath_k$ and $u_2 = \imath_{k+1} \cdots \imath_n$. By (T-COM), $\Gamma \vdash u_1, u_2 : \mathsf{A}_1, \mathsf{A}_2$ implies $\Gamma \vdash u_i : \mathsf{A}_i$ for $i = 1, 2$. $\Gamma \vdash \imath_1 \cdots \imath_n : \mathsf{A}_1, \mathsf{A}_2$, by (T-DOC) and, if needed, (T-SUB).

(TRY): $try\ \ u\ \ Q(\widetilde{v}) \ \to\ (\nu\,\imath, \ell)(\langle\, \imath \mapsto node\ \mathtt{root}(u) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \Rsh \ell\,)$. By (T-TRY), $\Gamma \vdash try\ u\, Q(\widetilde{v}) : \mathsf{loc}\,\mathtt{root}(\mathsf{A})$ implies $\Gamma \vdash Q : (\widetilde{\mathsf{T}}) \to \mathsf{A}$, $\Gamma \vdash \widetilde{v} : \widetilde{\mathsf{T}}$, and $\Gamma \vdash u : \mathsf{B}$.

By (T-LND), $\Gamma, \imath : \mathsf{node}\,\mathtt{root}(u) \vdash \langle\, \imath \mapsto node\ \mathtt{root}(u) \,\rangle : \star$ and, by (T-LTRY), $\Gamma, \imath : \mathsf{node}\,\mathtt{root}(u), \ell : \mathsf{loc}\,\mathtt{root}(\mathsf{A}) \vdash \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle : \star$. Finally, by (T-RES), (T-PAR), and (T-NM), $\Gamma \vdash (\nu\,\imath, \ell)(\langle\, \imath \mapsto node\ \mathtt{root}(u) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \Rsh \ell\,) : \mathsf{loc}\,\mathtt{root}(\mathsf{A})$;

(UNIT): $\langle\, \imath \mapsto node\ \mathtt{f}(()) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle \ \to\ \langle\, \imath \mapsto node\ \mathtt{f}(()) \,\rangle \Rsh \langle\, \ell \mapsto ok\ \imath \,\rangle$ implies $Q(\widetilde{v}) \triangleq Reg\, as\, v_k$ and $() \vdash_{Reg} Empty$. By (T-PAR), (T-LND) and (T-LTRY), $\Gamma \vdash \langle\, \imath \mapsto node\ \mathtt{f}(()) \,\rangle \Rsh \langle\, \ell \mapsto try\ \imath\, Q(\widetilde{v}) \,\rangle : \star$ implies $\Gamma \vdash \ell : \mathsf{loc}\,\mathtt{f}(\mathsf{A})$, $\Gamma \vdash \imath : \mathsf{node}\,\mathtt{f}(())$, $\Gamma \vdash Q(\widetilde{x}) : \widetilde{\mathsf{T}} \to \mathsf{A}$, and $\Gamma \vdash \widetilde{v} : \widetilde{\mathsf{T}}$. By Lemma 4.5, $() \vdash_{Reg} Empty$ and $Empty : \mathsf{Empty}$ we have $() \vdash_{\mathsf{A}} \mathsf{Empty}$. By Lemma 4.3 (1), $\mathsf{Empty} < \mathsf{A}$ and, by (T-EM) and (T-SUB), $\Gamma \vdash () : \mathsf{A}$. By (T-LOK) and (T-PAR), $\Gamma \vdash \langle\, \imath \mapsto node\ \mathtt{f}(()) \,\rangle \Rsh \langle\, \ell \mapsto ok\ \imath \,\rangle : \star$.

□

We do not state a *progress theorem* in connection with Theorem 4.1. Indeed, there exists no notion of errors in our calculus (like e.g. the notion of "message not understood" in object-oriented languages) as it is perfectly acceptable for a pattern-matching to fail or to get blocked on a *wait* statement. Nonetheless the subject reduction theorem is still useful. For instance, we can use it for optimizations purposes, like detecting that a specific matching will always fail.

4.5 Extensions

In this section we extend syntax, operational semantics and type system for showing how to deal with pattern definitions enriched with local declarations and continuations. We also introduce some examples that show how to interpret interesting programming idioms in our model, like spawning an expression in a new thread or handling user-defined exceptions.

4.5.1 Full pattern definitions

For the sake of simplicity, we have presented the calculus by considering patterns without local declarations and continuations; we discuss here changes introduced by the presence of full pattern definitions.

First of all, the syntax of the *test* resource is modified for taking into account pattern's continuations, e_1 and e_2. The new test resource is $\mathit{test}\ \imath\ w\ u\ e_1\ e_2$.

A revised semantics is needed. Reduction rules in Table 4.5 that regulate pattern-matching evaluation are substituted by the corresponding ones in Table 4.9.

Rules (TEST'-OK), (UNIT'), (TEST'-FAIL) and (ERR') extend the previous versions by simply starting the execution of continuations e_1 and e_2. The *test* resource created by (MTC') takes note of the continuations of the pattern Q and of the capture reference v_r. Local declaration D in Q are taken into account when evaluating sub-pattern-matchings and continuations.

Concerning the type system, a revised version of rule (T-LTEST) in Table 4.7 is needed for verifying well-typedness of continuations.

$$\text{(T-LTEST')}\ \frac{\begin{array}{c}\Gamma \vdash \ell : \mathsf{loc}\,\mathtt{f}(\mathsf{A}) \quad \Gamma \vdash \imath : \mathsf{node}\,\mathtt{f}(u) \quad \Gamma \vdash \jmath_k : \mathsf{loc}\,\mathtt{f}_k(\mathsf{A}_k) \\ w = (\jmath_1 \cdots \jmath_n) \quad \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{\mathsf{A}} \mathsf{A}_1 \cdots \mathsf{A}_n \\ \Gamma \vdash e_1 : \mathsf{T}_1 \quad \Gamma \vdash e_2 : \mathsf{T}_2 \quad \Gamma \vdash u : \mathsf{ref}\ \mathsf{B} \quad \mathsf{B}, \mathsf{A} < \mathsf{B}\end{array}}{\Gamma \vdash \langle\, \ell \mapsto \mathit{test}\ \imath\ w\ u\ e_1\ e_2 \,\rangle : \star}$$

$$\text{(ERR')}\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n)\,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k)\,\rangle \quad n \geqslant 0 \qquad Q(\widetilde{v}) \triangleq let\ D\ in\ Reg\ as\ v_r\ then\ e_1\ else\ e_2 \quad \mathtt{f}_1 \cdots \mathtt{f}_n \not\vdash_{Reg} \quad z \neq \ell, \imath}{P \Rsh \langle\, \ell \mapsto try\ \imath\ Q(\widetilde{v})\,\rangle \to P \Rsh let\ z = (let\ D\ in\ e_2)\ in \langle\, \ell \mapsto fail\ \imath\,\rangle}$$

$$\text{(MTC')}\ \frac{\begin{array}{c} P \equiv \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \cdots \imath_n)\,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \imath_k \mapsto node\ \mathtt{f}_k(w_k)\,\rangle \quad n > 0 \\ Q(\widetilde{v}) \triangleq let\ D\ in\ Reg\ as\ v_r\ then\ e_1\ else\ e_2 \\ \mathtt{f}_1 \cdots \mathtt{f}_n \vdash_{Reg} Q_1(\widetilde{v}_1) \cdots Q_n(\widetilde{v}_n) \qquad w = \jmath_1 \cdots \jmath_n \ \text{ fresh names} \end{array}}{\begin{array}{l} P \Rsh \langle\, \ell \mapsto try\ \imath\ Q(\widetilde{v})\,\rangle \to P \Rsh \big(let\ D\ in\ (\nu\, w)\big(\prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto try\ \imath_k\ Q_k(\widetilde{v}_k)\,\rangle \\ \qquad\qquad \Rsh \langle\, \ell \mapsto test\ \imath\ w\ v_r\ e_1\ e_2\,\rangle\big)\big) \end{array}}$$

$$\text{(UNIT')}\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(())\,\rangle \quad z \neq \ell, \imath \qquad Q(\widetilde{v}) \triangleq let\ D\ in\ Reg\ as\ v_r\ then\ e_1\ else\ e_2 \quad () \vdash_{Reg} Empty}{P \Rsh \langle\, \ell \mapsto try\ \imath\ Q(\widetilde{v})\,\rangle \to P \Rsh let\ z = (let\ D\ in\ e_1)\ in \langle\, \ell \mapsto ok\ \imath\,\rangle}$$

$$\text{(TEST'-OK)}\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \ldots \imath_n)\,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto ok\ \imath_k\,\rangle \qquad w = \jmath_1 \cdots \jmath_n \quad z \notin \mathrm{fn}(e_1) \quad z, z' \neq \ell, \imath, v_r}{\begin{array}{l} P \Rsh \langle\, \ell \mapsto test\ \imath\ w\ v_r\ e_1\ e_2\,\rangle \to P \Rsh let\ z' = \big(let\ z = (v_r \mathrel{+}= \imath_1 \cdots \imath_n)\ in\ e_1\big) \\ \qquad\qquad in \langle\, \ell \mapsto ok\ \imath\,\rangle \end{array}}$$

$$\text{(TEST'-FAIL)}\ \frac{P = \langle\, \imath \mapsto node\ \mathtt{f}(\imath_1 \ldots \imath_n)\,\rangle \Rsh \prod_{k=1,\ldots,n} \langle\, \jmath_k \mapsto d_k\,\rangle \quad w = \jmath_1 \cdots \jmath_n \quad z \neq \ell, \imath \qquad \forall k \in 1, \ldots, n : d_k \in \{ok\ \imath_k, fail\ \imath_k\} \qquad \exists j \in 1, \ldots, n : d_j = fail\ \imath_j}{P \Rsh \langle\, \ell \mapsto test\ \imath\ w\ v_r\ e_1\ e_2\,\rangle \to P \Rsh let\ z = e_2\ in \langle\, \ell \mapsto fail\ \imath\,\rangle}$$

Table 4.9: Reduction semantics with full pattern definitions.

4.5.2 Types and pattern-matching

We can encode a "traditional" *match* operator, as found in XDuce for example, that matches the pattern Q against u and conditionally proceeds as e_1 or e_2. Assume y is a fresh variable ($y \notin \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$), we define:

$$match\ u\ with\ Q(\widetilde{v})\ then\ e_1\ else\ e_2 \triangleq \begin{cases} let\ x = \big(try\ u\, Q(\widetilde{v})\big) \\ \ in\ \big(wait\ x(y)\, then\, e_1\, else\, e_2\big) \end{cases}.$$

This example allows us to emphasize the role of the variable y when typing a *wait* statement. Let $e \triangleq \big(match\ \ z\ with\ Empty\, then\, \mathtt{f}[z]\, else\, z\big)$ be the expression that returns z if it is not empty else returns $\mathtt{f}[z]$. Assume z is a variable of type All, then the most precise type for e is also All. In contrast, if we consider the expression $let\, x = \big(try\, z\, \mathsf{Empty}\big)\, in \big(wait\, x(y)\, then\, \mathtt{f}[y]\, else\, y\big)$, which is equivalent to e, we obtain

the more precise type $\overline{\mathsf{Empty}}$, that is, we prove that the returned value cannot be empty. Indeed y plays the role of an alias for the value of z that is used with type Empty in the continuation $\mathtt{f}[y]$ and with type $\overline{\mathsf{Empty}}$ in y (and we have $\mathtt{f}[\mathsf{Empty}] < \overline{\mathsf{Empty}}$).

4.5.3 Concurrency

We show how to model simple threads, that is, we want to encode an operator *spawn* such that the effect of $spawn\ e_1;\ e_2$ is to evaluate e_1 in parallel with e_2, yielding the value of e_2 as a result. The simplest solution is to interpret $spawn\ e_1;\ e_2$ by the configuration $e_1 \Rsh e_2$. A disadvantage of this solution is that it is not possible to test in e_2 whether the evaluation of e_1 has ended.

Another simple approach to encode *spawn* is to rely on the pattern-matching mechanism. Let Q be the pattern $Q(\,) \stackrel{\triangle}{=} (Empty\, then\, e_1)$. We can interpret the statement $spawn\ e_1; e_2$ with the expression $let\ x = (try\ \mathtt{()}\ Q(\,))\ in\ e_2$. Indeed we have:

$$let\ x = (try\ \mathtt{()}\ Q(\,))\ in\ e_2 \rightarrow^* (\nu\, \imath\ell)\big(\langle\, \imath \mapsto node\ \mathtt{root}(\mathtt{()})\,\rangle \Rsh (let\ z = e_1\ in \langle\, \ell \mapsto ok\ \imath\,\rangle) \Rsh e_2[\ell/x]\big)\ .$$

In the resulting process, e_1 and e_2 are evaluated concurrently and the resource $\langle\, \ell \mapsto ok\ \imath\,\rangle$ cannot interact with e_2 until the evaluation of e_1 ends. Hence an occurrence of the expression $(wait\ x(y)\, then\, e)$ in e_2 acts as an operator blocking the execution of e until e_1 returns a value. We can in fact improve our encoding so that the result of e_1 is bound to z in e as follows:

$$spawn\ e_1;\ e_2 \stackrel{\triangle}{=} (\nu\, \imath\ell)\big(let\ z = e_1\ in\ (\langle\, \imath \mapsto node\ \mathtt{root}(z)\,\rangle \Rsh \langle\, \ell \mapsto ok\ \imath\,\rangle) \Rsh e_2[\ell/x]\big)\ .$$

It emerges from this example that a *try* location can be viewed as a *future*, that is a reference to the "future result" of an asynchronous computation. More generally, we can liken a process $(\langle\, \imath \mapsto node\ \mathtt{f}(u)\,\rangle \Rsh \langle\, \ell \mapsto ok\ \imath\,\rangle)$ to an (asynchronous) output action $\ell!\langle ok, u\rangle$ as found in process calculi such as the π-calculus. Similarly, we can compare an expression $wait\ \ell(x)\, then\, e_1\, else\, e_2$ with a combination of input action and matching, $\ell?(x).\{ok \Rightarrow e_1 \mid fail \Rightarrow e_2\}$, with the following synchronization rules:

$$\begin{array}{llll} \ell!\langle ok, u\rangle & \parallel \ell?(x).\{ok \Rightarrow e_1 \mid fail \Rightarrow e_2\} & \rightarrow\ \ell!\langle ok, u\rangle & \parallel e_1[u/x] \\ \ell!\langle fail, u\rangle & \parallel \ell?(x).\{ok \Rightarrow e_2 \mid fail \Rightarrow e_2\} & \rightarrow\ \ell!\langle ok, u\rangle & \parallel e_2[u/x] \end{array}$$

The main distinction with "traditional process calculi" is that we are in a situation where outputs are replicated. For this reason, we can have multiple *wait* operators

synchronizing on the same location ℓ without the need for global consensus (or a lock) on the resource at ℓ. Nonetheless, since the calculus can express atomic reads and writes on a shared memory, it could be useful to rely on a standard mutual exclusion algorithm for accessing references. We could also interpret high-level primitives for mutexes directly in our calculus (see e.g. [78] for an example). Note also that there is no need for replication in our calculus since resources are persistent and recursive behaviors can be encoded using recursive function declarations.

4.5.4 Exceptions

We show how to model a simple exception mechanism in our calculus. Suppose we need to check that a document u of type A in Example 4.3.1 (the type of family trees) contains only women. This can be achieved using the pattern declarations $Q(\,) \stackrel{\triangle}{=} \texttt{woman}[Q'(\,)]*$ and $Q'(\,) \stackrel{\triangle}{=} \texttt{name}[\mathit{All}], \texttt{daughters}[Q(\,)], \texttt{sons}[\mathit{Empty}]$ and a matching expression $\mathit{try}\ u\, Q(\,)$. A drawback of this approach is that we need to wait for the completion of all sub-patterns to terminate before completing the computation, even if the matching trivially fails because we find an element tagged `man` early in the matching. A natural optimization is to use an explicit handling of failures, e.g. to add primitives to kill and "ping" (the location of) a *try* resource in the style of [11]. Another solution is to encode a basic mechanism for handling exceptions using the following derived operators, where $\imath_e$ is a default name associated to the location $\langle\, \imath_e \mapsto \mathit{node}\ \texttt{root}(\texttt{()})\,\rangle$:

$$
\begin{array}{rcll}
\mathit{exception} & = & (\nu\,\ell)\ell & \textit{creates a fresh (location) exception} \\
\mathit{throw}\ \ell & = & \langle\, \ell \mapsto \mathit{ok}\ \imath_e \,\rangle \Rsh \texttt{()} & \textit{raises an exception at } \ell \\
\mathit{catch}\ \ell\ e & = & \mathit{wait}\ \ell(x)\,\mathit{then}\, e & \textit{catches exception } \ell \textit{ and runs } e \quad (x \notin \mathrm{fv}(e))
\end{array}
$$

A simple example is to raise the exception at the end of a computation, like in the expression $\mathit{let}\ x\ =\ \mathit{exception}\ \mathit{in}\ \big((\cdots ; \mathit{throw}\ \ x) \Rsh \mathit{catch}\ \ x\ \ e\big)$. If and when the *throw* expression is evaluated, we obtain a configuration of the form $(\nu\,\ell)\big(\cdots \Rsh \langle\, \ell \mapsto \mathit{ok}\ \imath_e \,\rangle \Rsh \mathit{wait}\ \ell(x)\,\mathit{then}\, e\big)$, which starts the execution of e. For instance, it is possible to raise the exception in the compensation part of a pattern declaration and to redefine the pattern Q above in: $Q(x) \stackrel{\triangle}{=} \texttt{woman}[Q'(\,)] * \mathit{else}\,\mathit{throw}\ x$.

With our encoding, it is not possible to abort the execution of a whole "program block" using exceptions. Using a more involved encoding, e.g. based on CPS transforms, we could interpret this more general exception model.

4.6 Conclusions

In this chapter, we have proposed a formal model for computing over large, perhaps dynamically generated, distributed XML documents. We define a typed process calculus and show that it supports a first-order type system with subtyping based on regular expression types, a system compatible with DTD and other schema languages for XML. Our goal is to provide formal tools for studying concurrent computation models based on service composition and streamed XML data. Hence, the key aspect of the calculus is that documents are not represented as first class values exchanged in messages (like in XPi and in almost all works mixing process calculi and XML). This because this approach is inappropriate in the case of very large or dynamically generated data. At the opposite, in Astuce documents are considered as special kind of processes that can be randomly accessed through the use of distributed indexes.

Concerning query evaluation, in Astuce we take a strongly typed approach by extending the functional approach taken in e.g. XDuce and defining distributed *regular expression patterns*. As a byproduct, Astuce could be a basis for developing concurrent extensions of strongly typed languages for XML, such as XDuce. It could also be used to provide the semantics of systems in which XML documents contain active code that can be executed on distributed sites (i.e. processes and document texts are mixed), like in the Active XML system for example [4]. Nonetheless, since the operational semantics does not dictate how regular patterns should be implemented, we can take inspiration from these systems to implement efficient and scalable filtering primitives. Conversely, Astuce can be used to give a formal semantics to these systems.

What we have proposed here is a complementary model to the most popular one, at the basis e.g. of XPi and πDuce, which follows the *fiefdoms and emissaries* approach [85]. In this approach, WS are seen as fiefdoms, which own and manage data, while messages used for accessing services are the emissaries, which allow fiefdoms to collaborate with each other. This approach fits well with the message-oriented approach to WS and has several advantages, notably it makes easy to deal with security (confidentiality, access control and so on). In Astuce we have chosen to focus on distribution and to investigate a different (more complex) approach, but there is still a lot of work to be done to make it realistic. For example, it would be necessary to undersand what happens in presence of updates, to study security related problems from another point of view and to introduce different solutions to these problems.

Part II

Policies and obligations

Chapter

FIVE

Responsiveness of services

In this chapter we study one of the properties we consider critical for SOA*:* responsiveness*. A responsive service is a service that guarantees that each request is eventually followed by a reply. Here, we model services as π-calculus processes and propose two distinct type systems, each of which statically guarantees responsive usage of names in well-typed π-calculus processes. In a process calculus, an agent guarantees responsive usage of a channel name r if a communication along r is guaranteed to eventually take place. In the first system, we achieve responsiveness by combining techniques for deadlock and livelock avoidance with* linearity *and* receptiveness*. The latter is a guarantee that a name is ready to receive as soon as it is created. These conditions imply relevant limitations on the nesting of actions and on multiple use of names in processes. In the second system, we relax these requirements so as to permit certain forms of nested inputs and multiple outputs. We demonstrate the expressiveness of the second system by showing that Cook and Misra's service orchestration language* ORC *can be encoded into well-typed processes.*

5.1 Introduction

Contracts are at the basis of SOA and describe services not only in functional terms, but also specifying a non-functional part containing the rules of engagement between consumers and providers, also known as *policies*. Policies define some constraints on the behavior of services such as *obligations*, which require a service to perform an action or to be in one or more allowable states, and *permissions*, which enable one or more actions. The *policy-oriented* model [27] describes WS by focusing on those aspects related to policies. Obligations and permissions are used to constrain

the behavior of services, to ensure security (e.g. by performing access control) and to guarantee a certain level of quality of service. To this purpose, obligations usually require agents providing services to be in a certain state, such as a readiness state, or to perform some actions, such as sending a reply after each invocation. A service in a readiness state is said to be *receptive*, while a service that *eventually* reply after each invocation is said *responsive*. The concept of receptiveness has been already formalized and studied from the process calculi viewpoint by Sangiorgi [125]. We first concentrate on responsiveness of services. In this chapter, first we formally define what it means for a process to be responsive and after we propose two distinct type systems, each of which statically guarantees responsive usage of names in well-typed π-calculus processes.

In a process calculus, an agent guarantees responsive usage of a channel name r if a communication along r is guaranteed to eventually take place. That is, under a suitable assumption of fairness, all computations contain at least one reduction with r as subject. Here we are interested in the case where r is a reply channel passed to a service or function, and we call this property *responsiveness*. As an example, a network of processes S may contain a service $!a(x, r).P$ invocable in RPC style: the caller sends at a an argument x and a return channel r. S's responsive usage of r implies that every request at a will be eventually replied.

In this chapter, we aim at individuating substantial classes of π-calculus processes that guarantee responsiveness and that can be checked statically, that is, without having to explicitly or implicitly unfold the behavior of the process under consideration. In the past decade, several type systems for the π-calculus have been proposed to analyze properties that share some similarities with responsiveness, such as linearity [99], uniform receptiveness [125], lock freedom [95, 96] and termination [64]; they will be examined throughout the chapter. However none of the above mentioned properties alone is sufficient, or even necessary, to ensure the property we are after.

The first system we propose (Section 5.3) builds around Sangiorgi's system for uniform receptiveness [125]. However, uniformity is discarded and some other constraints are introduced, as explained below. As expected, most difficulties in achieving responsiveness originate from responsive names being passed around. If an intended receiver of a responsive name r, say $a(x).P$, never becomes available, r might never be delivered, hence used. In this respect, receptiveness is useful, because it ensures that inputs on a and on r are available as soon as they are created.

Even when delivery of r is ensured, however, one should take care that r will be

processed properly. Indeed, the recipient of r might just "forget" about r, like in $(\nu\, a,\, r)(a(x).\mathbf{0} \mid \overline{a}\langle r\rangle)$; or r might be passed from one recipient to another, its use as a subject being delayed forever, like in

$$(\nu\, a,\, b)\big(\, !a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle \mid \overline{a}\langle r\rangle \,\big)\,. \tag{5.1}$$

The first situation can be avoided by imposing that in the receiver $a(x).P$, name x occurs at least once in the body P. In fact, it is necessary that any responsive name be used *linearly*, that is, that it appear exactly once in input and once in output. Infinite delays like (5.1) can be avoided relying on a stratification of names into *levels*, akin to the type system for termination of Deng and Sangiorgi [64]. We will rule out divergent computations that involve responsive names infinitely often, but we will do allow divergence in general.

Finally, even when a responsive name is eventually in place as subject of an output action, one has to make sure that such an action becomes eventually available. In other words, one must avoid cyclic waiting like in

$$r(x).\overline{s}\langle x\rangle \mid s(y).\overline{r}\langle y\rangle\,. \tag{5.2}$$

This will be achieved by building a graph of the dependencies among responsive names and then checking for its acyclicity.

Receptiveness and linearity impose relevant limitations on the syntax of well-typed processes: nested free inputs are forbidden, as well as multiple outputs on the same name. On the other hand, the type system is expressive enough to enable a RPC programming style; in particular the usual CPS encoding of primitive recursive functions into π-processes gives rise to well-typed processes [7].

In the second system we propose, Section 5.4, the constraints on receptiveness and linearity are relaxed so as to allow certain forms of nested inputs and multiple outputs. For instance, the new system allows nondeterministic internal choice, which was forbidden in the first one. Relaxation of linearity and receptiveness raises new issues, though. As an example, responsiveness might fail due to "shortage" of inputs, like in (a, b and d responsive):

$$\overline{a}\langle b\rangle|\overline{a}\langle d\rangle|a(x).\overline{x}|b|d \xrightarrow{\tau}\xrightarrow{\tau} \overline{a}\langle d\rangle|d\,.$$

These issues must be dealt with by carefully "balancing" inputs and outputs in typing contexts and in processes. This system is flexible enough to encode into well-typed processes all orchestration patterns of Cook and Misra's ORC language [59], Section 5.5. Due to a rather crude use of levels, however, only certain forms of (tail-)recursion are

Process	P, R	::=	$\mathbf{0}$	*Inaction*
			$\mid \overline{a}\langle b\rangle$	*Output*
			$\mid a(x).P,$ $\quad x \notin \mathrm{in}(P)$	*Input prefix*
			$\mid !a(x).P,$ $\quad x \notin \mathrm{in}(P)$	*Replication*
			$\mid P \mid R$	*Parallel Composition*
			$\mid (\nu\, b)P$	*Restriction*

Table 5.1: Syntax of processes

encodable. In fact, neither the first system is subsumed by the second one, nor vice versa.

The most lengthy and technical proofs of this chapter are reported in Appendix A for readers' convenience.

5.2 Syntax and semantics

In this section we describe the syntax of processes and types and the operational semantics of the calculus. On top of the operational semantics, we define the responsiveness property we are after.

5.2.1 Syntax

We focus on an asynchronous variant of the π-calculus without nondeterministic choice. Indeed, asynchrony is a natural assumption in a distributed environment. Moreover, in the presence of a choice, it would be difficult to guarantee responsiveness of names that belong to discarded branches. Assume a countable set of names $\mathcal{N}$, ranged over $a, b, \ldots, x, y, \ldots$.

Definition 5.1. *The set $\mathcal{P}$ of* processes $P, R, \ldots$ *is defined as the set of terms generated by the syntax in Table 5.1.*

In a non blocking output action $\overline{a}\langle b\rangle$, name a is said to occur in *output subject position* and b in *output object position*. In an input prefix $a(x).P$, and in a replicated input prefix $!a(x).P$, name a is said to occur in *input subject position* and x in *input object position*. We denote by $\mathrm{in}(P)$ the set of names occurring free in input subject position in P. The condition $x \notin \mathrm{in}(P)$, for input and replicated input, means that names can be passed around with the output capability only. This assumption simplifies reasoning on types and does not significantly affect the expressiveness of

Type	$\mathsf{T} ::=$	I		*Inert*
		$\mid \mathsf{T}^{\mathsf{U}}$		*Channel*
Usage	$\mathsf{U} ::=$	$[\omega, k]$,	$k \geqslant 0$	*Omega*
		$\mid [\rho, k]$,	$k \geqslant 0$	*Responsive*

Table 5.2: Syntax of types

the language (see e.g. [28, 108]). As usual, parallel composition, $P|R$, represents the concurrent execution of P and R and restriction, $(\nu\, b)P$, creates a fresh name b with initial scope P.

Binding conventions and notations. Notions of free and bound names ($\mathrm{fn}(\cdot)$ and $\mathrm{bn}(\cdot)$), and alpha-equivalence ($=_\alpha$) arise as expected. In the following, we shall only consider *well-formed* processes, where all bound names are distinct from each other and from free names. Please note that, as in [64], we do *not* identify processes up to alpha-equivalence, hence we will need to introduce an explicit operational rule for alpha-equivalence later on. We will better motivate this choice in page 105.

We shall often abbreviate $a(x).\mathbf{0}$ as $a(x)$, and $(\nu\, a_1) \ldots (\nu\, a_n)P$ as $(\nu\, a_1, \ldots, a_n)P$ or $(\nu\, \widetilde{a})P$, where $\widetilde{a} = a_1, \ldots, a_n$. In a few examples, the object part of an action may be omitted if not relevant for the discussion; e.g., $a(x).P$ may be shortened into $a.P$.

5.2.2 Sorts and types

The set of names $\mathcal{N}$ is partitioned into a family of countable *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. A fixed sorting à la Milner [109] is presupposed: that is, any sort $\mathcal{S}$ has an associated object sort $\mathcal{S}'$, and a name of sort $\mathcal{S}$ can only carry names of sort $\mathcal{S}'$. We only consider processes that are well-sorted in this system. Alpha-equivalence is assumed to be sort-respecting, in the obvious sense. Each sort is associated with a *type* T taken from the set $\mathcal{T}$ defined below. We write $a : \mathsf{T}$ if a belongs to a sort $\mathcal{S}$ with associated type T. The association between types and sorts is such that for each type there is at least one sort of that type.

Definition 5.2 (types). *The set $\mathcal{T}$ of* types $\mathsf{T}, \mathsf{S}, \ldots$ *contains the constant $\boldsymbol{\mathcal{L}}$ and the set of terms generated by the grammar in Table 5.2.*

A channel type $\mathsf{T}^{[\mathsf{u},k]}$ conveys three pieces of information: a type of carried objects T, a *usage* u, that can be *responsive* (ρ) or *ω-receptive* (ω), and an integer *level* $k \geqslant 0$. If $a : \mathsf{T}^{[\mathsf{u},k]}$ and $\mathsf{u} = \rho$ (resp. $\mathsf{u} = \omega$) we say that a is *responsive* (resp. *ω-receptive*).

Informally, responsive names are guaranteed to be eventually used as subject in a communication, while ω-receptive names are guaranteed to be constantly ready to receive. Levels are used to bound the number of times a responsive name can be passed around, so to avoid infinite delay in their use as subject. We also consider a type I of *inert* names that cannot be used as subject of a communication – they just serve as tokens to be passed around. Finally, a type $\mathbf{\perp}$ is introduced to collect those names that cannot be used at all. As we discuss below, $\mathbf{\perp}$ is useful to formulate the subject reduction property while keeping the standard operational semantics.

5.2.3 Operational semantics

The semantics of processes is given by a labeled transition system in the early style, whose rules are presented in Table 5.3. An *action* μ can be of the following forms: free output, $\overline{a}\langle b\rangle$, bound output, $\overline{a}(b)$, input $a(b)$, or internal move $\tau\langle a, b\rangle$. The last mentioned is used for denoting a communication where the – free or bound – names a and b are used respectively as subject and object. This additional annotation is necessary for guaranteeing the linear usage of responsive names as communication subject (see rules (RES) and (RES-ρ)). We define $\mathrm{n}(a(b)) = \mathrm{n}(\overline{a}\langle b\rangle) = \mathrm{n}(\overline{a}(b)) = \mathrm{n}(\tau\langle a, b\rangle) = \{a, b\}$. A substitution σ is a finite partial map from names to names; for any term P, we write $P\sigma$ for the result of applying σ to P, with the usual renaming convention to avoid captures.

The rules are standard, with the difference that the notation $\xrightarrow{\tau\langle a,b\rangle}$ is used to denote a τ-transition. Moreover, $P >_{\alpha,\rho} R$ in (ALPHA) means that $P\rho = R$, where $\rho : V \to (\mathcal{N} \setminus \mathrm{fn}(P))$, with $\mathrm{bn}(P) \subseteq V$, is an injective (sort-respecting) renaming function. Similarly for $\mu >_{\alpha,\rho} \mu'$. Rule (ALPHA) simply states that if P is obtained by applying a renaming ρ to R, then P's behavior corresponds to the ρ renaming of the behavior of R. As an example, suppose $P = (\nu\, a, b)(\overline{b}\langle a\rangle \mid b(x).\overline{x})$ and $P >_{\alpha,\rho} R$, with $\rho(a) = d$ and $\rho(b) = c$. Then $R = (\nu\, d, c)(\overline{c}\langle d\rangle \mid c(x).\overline{x})$, $R \xrightarrow{\tau\langle c,d\rangle} (\nu\, d, c)\overline{d}$, $\tau\langle b, a\rangle >_{\alpha,\rho} \tau\langle c, d\rangle$, $\overline{a} >_{\alpha,\rho} \overline{d}$ and $P \xrightarrow{\tau\langle b,a\rangle} (\nu\, a, b)\overline{a}$. Note also that reduction rules for restriction use information on typing. In rule (RES-ρ), a bound responsive subject a is alpha-renamed to a $\mathbf{\perp}$-name c (a sort of "casting" of a to type $\mathbf{\perp}$.) Informally, this alpha-renaming is necessary because in a well-typed process, due to the linearity constraint on responsive names, name a must vanish after being used as subject. Rule (RES) deals with the remaining cases of restriction. E.g. in $a(x).\overline{x}|(\nu\, c)(\overline{a}\langle b\rangle)$ the RHS can evolve by outputting b on a – this is obtained by applying rules (OUT) and (RES) ($\mu = \overline{a}\langle b\rangle$ and $c \notin \mathrm{n}(\mu)$) – hence the whole process can reduce to $\overline{b}|(\nu\, c)\mathbf{0}$ by performing a communication on a ((IN) and (COM)). Similarly, if we suppose d is an ω-receptive

$$(\text{IN})\ a(x).P \xrightarrow{a(b)} P[b/x] \qquad (\text{REP})\ !a(x).P \xrightarrow{a(b)} !a(x).P|P[b/x]$$

$$(\text{OUT})\ \overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0} \qquad (\text{ALPHA})\ \frac{P >_{\alpha,\rho} R \quad R \xrightarrow{\mu'} R' \quad P' >_{\alpha,\rho} R' \quad \mu >_{\alpha,\rho} \mu'}{P \xrightarrow{\mu} P'}$$

$$(\text{COM}_1)\ \frac{P \xrightarrow{\overline{a}\langle b\rangle} P' \quad R \xrightarrow{a(b)} R'}{P|R \xrightarrow{\tau\langle a,b\rangle} P'|R'} \qquad (\text{PAR}_1)\ \frac{P \xrightarrow{\mu} P' \quad \text{bn}(\mu) \cap \text{fn}(R) = \emptyset}{P|R \xrightarrow{\mu} P'|R}$$

$$(\text{OPEN})\ \frac{P \xrightarrow{\overline{a}\langle b\rangle} P' \quad a \neq b}{(\nu\, b)P \xrightarrow{\overline{a}(b)} P'} \qquad (\text{CLOSE}_1)\ \frac{P \xrightarrow{\overline{a}(b)} P' \quad R \xrightarrow{a(b)} R' \quad b \notin \text{fn}(R)}{P|R \xrightarrow{\tau\langle a,b\rangle} (\nu\, b)(P'|R')}$$

$$(\text{RES})\ \frac{P \xrightarrow{\mu} P' \quad \text{if } a \in \text{n}(\mu) \text{ then } \exists b : \begin{cases} \text{either} & \mu = \tau\langle b, a\rangle \\ \text{or} & \mu = \tau\langle a, b\rangle \text{ and } a \text{ not responsive} \end{cases}}{(\nu\, a)P \xrightarrow{\mu} (\nu\, a)P'}$$

$$(\text{RES-}\rho)\ \frac{P \xrightarrow{\tau\langle a,b\rangle} P' \quad a \text{ responsive} \quad c : \mathbf{L} \quad c \text{ fresh}}{(\nu\, a)P \xrightarrow{\tau\langle a,b\rangle} (\nu\, c)P'[c/a]}$$

Symmetric rules not shown.

Table 5.3: Operational semantics.

name, the process $(\nu\, d)(!d(x).\overline{x}|\overline{d}\langle e\rangle|\overline{d}\langle f\rangle)$ can reduce to $(\nu\, d)(!d(x).\overline{x}|\overline{e}|\overline{d})$ by applying rules (REP), (OUT), (COM) and (RES) ($\mu = \tau\langle d, -\rangle$ and d is not responsive).

Remark 5.1. It is worth to notice that, by ignoring the annotations of names in communication labels, that is by substituting each $\tau\langle a, b\rangle$ with τ, the relation $\xrightarrow{\mu}$ defined by rules in Table 5.3 coincides with the relation defined by the standard operational semantics for the π-calculus.

Notations. We shall often refer to a silent move $P \xrightarrow{\tau\langle a,b\rangle} P'$, sometimes abbreviated as $P \xrightarrow{\tau} P'$, as a *reduction* and we denote by $\xrightarrow{\tau}{}^*$ the reflexive and transitive closure of $\xrightarrow{\tau}$. $P \xrightarrow{[a]} P'$ means $P \xrightarrow{\tau\langle a,b\rangle} P'$ for some free or bound name b. For a string $s = a_1 \cdots a_n \in \mathcal{N}^*$, $P \xrightarrow{[s]} P'$ means $P \xrightarrow{[a_1]} \cdots \xrightarrow{[a_n]} P'$, while $P \xRightarrow{[a]} P'$ means $P \xrightarrow{\tau}{}^* \xrightarrow{[a]} \xrightarrow{\tau}{}^* P'$. We use such abbreviations as $P \xRightarrow{[a]}$ to mean that there exists P' such that $P \xRightarrow{[a]} P'$.

We can now introduce the responsiveness property we are after. Informally, we think of a fair computation as a sequence of communications where for no name a a transition $\xrightarrow{[a]}$ is enabled infinitely often without ever taking place. Then a process uses

a name in a responsive way if that name is eventually, that is, in all fair computations, used as subject of a communication. We then have the following definition. Below, we assume that any bound name occurring in P and r is distinct from any free name in P.

Definition 5.3 (responsiveness). *Let P be a process and $r \in \mathrm{fn}(P)$. We say that P* guarantees responsiveness of c *if whenever $P \xrightarrow{[s]} P'$ ($s \in \mathcal{N}^*$) and r does not occur in s then $P' \overset{[r]}{\Longrightarrow}$.*

5.3 The type system $\vdash_1$

The type system consists of judgments of the form $\Gamma\,;\,\Delta \vdash_1 P$, where Γ and Δ are sets of names. Before introducing the typing rules, we informally present the system and introduce some preliminary definitions.

5.3.1 Overview of the system

Informally, names in Γ are those used by P in input, while in Δ are those used by P in output actions. There are several constraints on the usage of these names by P. We require *receptiveness* of names, hence each name in Γ must occur *immediately* (at top level) in input subject position, exactly once if it is responsive and replicated if it is ω-receptive. A responsive name in Δ must occur in P exactly once either in subject or in object output position, although not necessarily at top level, that is, occurrences in output actions underneath prefixes are allowed. There are no constraints on the use in output actions of ω-receptive names: they may be used an unbounded number of times, including zero. *Linearity* ("exactly once" usage) on responsive names is useful to avoid dealing with "dangling" responsive names, which might arise after a communication, like in

$$(\nu\, r)(r.\mathbf{0}|\overline{r}|\overline{r}) \;\xrightarrow{\tau}\; (\nu\, r)(\mathbf{0}|\mathbf{0}|\overline{r})$$

where r is a responsive name and we ignore the object parts of the communications. If the process on the LHS above were declared well-typed, the transition would violate the subject reduction property, as the process on the RHS above cannot be well-typed: it violates the balancing condition described later in this section.

Linearity and receptiveness alone are not sufficient to guarantee a responsive usage of names. As discussed in Section 5.1, we have also to avoid deadlock situations involving responsive names, like (5.2). This is simply achieved by building a *graph of dependencies* among responsive names of P (defined in the sequel) and checking for

$P\|R \equiv R\|P$	$P\|\mathbf{0} \equiv P$	$(P\|R)\|S \equiv P\|(R\|S)$
$P \equiv R$	if $P >_{\alpha,\rho} R$	$(\nu\, a)\mathbf{0} \equiv \mathbf{0}$ if $a : \mathbb{L}$ or $a : \mathsf{I}$
$(\nu\, a)(P\|R) \equiv (\nu\, a)P\|R$	if $a \notin \mathrm{fn}(R)$	$(\nu\, a)(\nu\, b)P \equiv (\nu\, b)(\nu\, a)P$

Table 5.4: Structural congruence

its acyclicity. We have also to avoid those situations like in (5.1), where a responsive name is indefinitely "ping-pong"-ed among a group of replicated processes. To this purpose, *levels* in types are introduced and the typing rules stipulate that sending a responsive name to a replicated input of level k may only trigger output of level less than k. This is similar to the use of levels in [64] to ensure termination. In our case, we just avoid divergent computations that involve responsive names.

There is one more condition necessary for responsiveness, that is, the sets of input and output names must be *balanced*, so as to ban situations like an output with no input counterpart. This constraint, however, is most easily formulated "on top" of well-typedness, and will be discussed later on.

5.3.2 Preliminary definitions

Formulation of the typing rules requires a few preliminary definitions. A *structural equivalence* is necessary in order to correctly formulate the absence of cyclic waiting on responsive names.

Definition 5.4 (structural congruence). *The* structural congruence $\equiv$ *is the least congruence on processes satisfying the rules in Table 5.4.*

Let us point out a couple of differences from the standard definition [109]. First, there is no rule for replication ($!P \equiv P|!P$), as its right-hand side would not be well-typed. Consider e.g. the process $!a(x).R$, with a ω-receptive name; the process $a(x).R \,|\, !a(x).R$ is not well-typed because ω-receptive names cannot be used as subject of non-replicated inputs. Similarly, in $\mathbf{0} \equiv (\nu\, a)\mathbf{0}$ we require $a : \mathbb{L}$ or $a : \mathsf{I}$. This because the type system requires that bound non inert names be used in input subject position at least once, and this is not the case in $\mathbf{0}$. In what follows we identify two classes of processes: *prime* processes and processes in *normal-form*.

Definition 5.5 (normal-form). *A process P is* prime *if either $P = \overline{a}\langle b\rangle$, or $P = a(x).P'$ or $P = !a(x).P'$. A process P is in* normal form *if $P = (\nu\, \widetilde{d})(P_1|\cdots|P_n)$ ($n \geqslant 0$), every P_i is prime and $\widetilde{d} \subseteq \mathrm{fn}(P_1, ..., P_n)$.*

$\mathrm{os}(\mathbf{0}) = \emptyset$	$\mathrm{os}(!a(b).P) = \emptyset$	$\mathrm{os}(a(b).P) = \mathrm{os}(P)$
$\mathrm{os}(\overline{a}\langle b\rangle) = \{a\}$	$\mathrm{os}((\nu\, a)P) = \mathrm{os}(P)$	$\mathrm{os}(P\vert R) = \mathrm{os}(P) \cup \mathrm{os}(R)$

Table 5.5: $\mathrm{os}(P)$

Every process is easily seen to be structurally equivalent to a process in normal form.

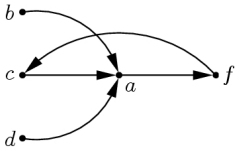
Lemma 5.1. *For each $P \in \mathcal{P}$ there exists $R \in \mathcal{P}$ in normal form such that $P \equiv R$.*

Proof: The proof is straightforward by induction on the structure of P. □

We proceeds by defining the dependency graph discussed above. Informally, its nodes correspond to responsive names of typing contexts and there is an arc from a node a to b exactly when an output action that involves a depends on an input action on b. Although the following definition does not mention processes, one should think of the pairs (Γ_i, Δ_i) below as typing contexts – limited to responsive names – for the P_i's in $P_1 | \cdots | P_n$.

Definition 5.6 (dependency graph). *Let $\{(\Gamma_i, \Delta_i) : i = 1, ..., n\}$ be a set of context pairs. The* dependency graph $\mathrm{DG}(\Gamma_i, \Delta_i)_{i=1,...,n}$ *is a graph (V, T) where: $V = \bigcup_{i=1,...,n}(\Gamma_i \cup \Delta_i)$ is the set of nodes and $T = \bigcup_{i=1,...,n}(\Delta_i \times \Gamma_i)$ is the set of arcs.*

Example 5.3.1. *The (cyclic) dependency graph associated to the process $a.(\overline{b} \,|\, \overline{c} \,|\, \overline{d}) \,|\, f.\overline{a} \,|\, c.\overline{f}$ is depicted below.*



We will have more to say on both structural equivalence and dependency graphs in Remark 5.2 at the end of the section. Like in [64], we will use a function $\mathrm{os}(P)$, defined in Table 5.5, that collects all – either free or bound – names in P that occur as subject of an *active* output action, that is, an output not underneath a replication. It is plain now why we choose to do not identify processes up to alpha-equivalence. If otherwise stated it would have been impossible to give a syntactical definition of $\mathrm{os}(P)$ (and of other functions defined in the following).

Notations. For any name a, we set $\mathrm{lev}(a) = k$ if $a : \mathsf{T}^{[\mathsf{u},k]}$ for some T and u, otherwise $\mathrm{lev}(a)$ is undefined. Given a set of names V, define $V^\rho \stackrel{\triangle}{=} \{\, x \in V \mid x \text{ is responsive} \,\}$ and $V^\omega \stackrel{\triangle}{=} \{\, x \in V \mid x \text{ is } \omega\text{-receptive} \,\}$. For V and W sets of names, we define $V \ominus W \stackrel{\triangle}{=} V \setminus W^\rho$. If $\Delta \cap \Delta' = \emptyset$, we abbreviate $\Delta \cup \Delta'$ as Δ, Δ' and if $a \notin \Delta$, we abbreviate $\Delta \cup \{a\}$ as Δ, a; similarly for Γ.

5.3.3 The typing rules

The type system is displayed in Table 5.6. Recall that each sort has an associated type. Linear usage of responsive names is ensured by rules (T-Nil) and (T-Out), by the disjointness conditions in (T-Par) and by forbidding responsive names to occur free underneath replication (T-Rep). Absence of cyclic waiting involving responsive names is checked in (T-Par) and in (T-Inp) ($a \notin \Delta$). Note the use of levels in rule (T-Rep): communication involving a replicated input subject a and a responsive object can only trigger outputs of level less than $\mathrm{lev}(a)$. This condition is meant to avoid those never-ending "ping-pongs" of responsive names mentioned above. Finally, rule (T-Res) ensures that bound responsive names are used both in input and in output, and ω-receptive names are used at least in input. Rule (T-Res-$\mathbf{1}$) prevents from using a name of type $\mathbf{1}$ and (T-Res-i) deals with inert names. We say that a process P is *well-typed* if there are Γ and Δ such that $\Gamma; \Delta \vdash_1 P$ holds.

Remark 5.2. Avoiding deadlocks on responsive names might be achieved by using levels in rule (T-Inp), in the same fashion as in rule (T-Rep), rather than using graphs in (T-Par). In fact, this would rule out cyclic waiting such as the one in (5.2). We shall pursue this approach in the system of Section 5.4, where there is no way of defining a meaningful notion of dependency graph. However, in the present system, a level-based way of dealing with cyclic waiting would be unnecessarily restrictive, in particular, it would ban as ill-typed the usual encoding of recursive functions into processes (see [7]).

Note that the premise $P = P_1 | \cdots | P_n$, with $P_1, \ldots, P_n$ prime, in rule (T-Par) calls for a typing rule for structural congruence, but despite its presence the type system may be viewed as essentially syntax driven, in the following sense. Given P in normal form, $P = (\nu\, \widetilde{d})(P_1 | \cdots | P_n)$, and ignoring structural equalities that just rearrange the $\widetilde{d}$ or the P_i's, there is at most one rule one can apply with P in the conclusion. This is made formal below.

We define a normal derivation of $\Gamma; \Delta \vdash_1 S$ to be one where rule (T-Str) is applied only where strictly necessary:

$$\text{(T-Out)}\ \frac{a,b \in \Delta \quad a : \mathsf{T}^{\mathsf{U}} \quad b : \mathsf{T} \quad \Delta^{\rho} \ominus \{a,b\} = \emptyset}{\emptyset; \Delta \vdash_1 \overline{a}\langle b \rangle}$$

$$\text{(T-Nil)}\ \frac{\Gamma = \Delta^{\rho} = \emptyset}{\Gamma; \Delta \vdash_1 \mathbf{0}} \qquad \text{(T-inp)}\ \frac{a : \mathsf{T}^{[\rho,k]} \quad b : \mathsf{T} \quad a \notin \Delta \quad \emptyset; \Delta, b \vdash_1 P}{a; \Delta \vdash_1 a(b).P}$$

$$\text{(T-Str)}\ \frac{P \equiv R \quad \Gamma; \Delta \vdash_1 R}{\Gamma; \Delta \vdash_1 P} \qquad \text{(T-Res)}\ \frac{a : \mathsf{T}^{\mathsf{U}} \quad \Gamma, a; \Delta, a \vdash_1 P}{\Gamma; \Delta \vdash_1 (\nu\, a)P}$$

$$\text{(T-Res-}\mathbb{L}\text{)}\ \frac{a : \mathbb{L} \quad \Gamma; \Delta \vdash_1 P}{\Gamma; \Delta \vdash_1 (\nu\, a)P} \qquad \text{(T-Res-I)}\ \frac{a : \mathsf{I} \quad \Gamma; \Delta, a \vdash_1 P}{\Gamma; \Delta \vdash_1 (\nu\, a)P}$$

$$\text{(T-Rep)}\ \frac{a : \mathsf{T}^{[\omega,k]} \quad b : \mathsf{T} \quad \Delta^{\rho} = \emptyset \quad \emptyset\,; \Delta, b \vdash_1 P \quad (\ b \text{ responsive} \ \text{ implies } \forall c \in \mathrm{os}(P) : \mathrm{lev}(c) < k\ )}{a\,; \Delta \vdash_1 !a(b).P}$$

$$\text{(T-Par)}\ \frac{P = P_1 | \cdots | P_n \ \ (n > 1) \quad \forall i : P_i \text{ is prime and } \Gamma_i\,; \Delta_i \vdash_1 P_i \quad \forall i \neq j : \Gamma_i^{\rho} \cap \Gamma_j^{\rho} = \emptyset \text{ and } \Delta_i^{\rho} \cap \Delta_j^{\rho} = \emptyset \quad \mathrm{DG}(\Gamma_i^{\rho}, \Delta_i^{\rho})_{i=1,\ldots,n} \text{ is acyclic}}{\bigcup_{i=1,\ldots,n} \Gamma_i \ ; \ \bigcup_{i=1,\ldots,n} \Delta_i \ \vdash_1 P}$$

Table 5.6: Typing rules of $\vdash_1$

Definition 5.7 (normal derivation). *A* normal derivation *of* $\Gamma; \Delta \vdash_1 S$ *is a derivation such that at each application of rule* (T-Str) *(Table 5.6) the process* P *in the conclusion* is not *in normal-form, while the process* R *in the premise* is *in normal form.*

For each well typed process P there exists a normal derivation.

Lemma 5.2. *Suppose* $\Gamma; \Delta \vdash_1 P$*, then there exists a normal derivation of* $\Gamma; \Delta \vdash_1 P$*.*

5.3.4 Properties of type system $\vdash_1$

In this section we prove that processes well-typed in $\vdash_1$ are also responsive in the sense of Definition 5.3. For the shake of completeness, omitted proofs are reported in a separate appendix (Appendix A, Section A.1, A.2 and A.3).

Subject reduction states that well-typedness is preserved through reductions, and it is our first step towards proving responsiveness.

Theorem 5.1 (subject reduction). *Suppose* $\Gamma; \Delta \vdash_1 P$ *and* $P \xrightarrow{[a]} P'$*. Then* $\Gamma \ominus \{a\}; \Delta \ominus \{a\} \vdash_1 P'$*.*

Our task is to prove that any balanced well-typed process guarantees responsiveness for all responsive names it contains. In the following definition we formally

$\mathrm{wt}(\mathbf{0}) = 0$	$\mathrm{wt}((\nu\, a)P) = \mathrm{wt}(P)$	$\mathrm{wt}(P\|R) = \mathrm{wt}(P) + \mathrm{wt}(R)$
$\mathrm{wt}(!a(b).P) = 0$	$\mathrm{wt}(a(b).P) = \mathrm{wt}(P)$	$\mathrm{wt}(\overline{a}\langle b\rangle) = 0_{\mathrm{lev}(a)}$

Table 5.7: $\mathrm{wt}(P)$

identify balanced processes:

Definition 5.8 (balanced processes). *A process P is* $(\Gamma;\Delta)$-balanced *if* $\Gamma;\Delta \vdash_1 P$, $\Gamma^\rho = \Delta^\rho$ *and* $\Delta^\omega \subseteq \Gamma^\omega$. *It is* balanced *if it is* $(\Gamma;\Delta)$*-balanced for some* Γ *and* Δ.

We need two main ingredients for the proof. The first one is given by the following proposition, stating that if the dependency graph of a process P is acyclic, then P always offers at least one output action involving a responsive name.

Proposition 5.1. *Suppose that* $\Gamma;\Delta \vdash_1 P$, *with* Γ, Δ *and* P *satisfying the conditions in the premise of rule* (T-Par) *and* $\Gamma^\rho = \Delta^\rho$. *Then for some* $j \in \{1,\ldots,n\}$ *we have* $P_j = \overline{a}\langle b\rangle$ *with either* a *or* b *responsive.*

Next, we need a measure of processes that is decreased by reductions involving responsive names. We borrow from [64] the definition of *weight* of P, written $\mathrm{wt}(P)$: this is defined as a vector $\langle w_k, w_{k-1}, \ldots, w_0\rangle$, where $k \geqslant 0$ is the highest level of names in $\mathrm{os}(P)$, and w_i is the number of occurrences in output subject position of names of level i in P. A formal definition is given in Table 5.7. Here, "0_k" is an abbreviation for the vector $\langle 1, 0, \ldots, 0\rangle$ with k components "0" following "1". The vector with just one component that equals "0" is denoted by 0. Sum "+" between two vectors is performed component-wise if they are of the same length; if not, the shorter one is first "padded" by inserting on the left as many 0's as needed.

The set of all vectors can be ordered lexicographically. Assuming two vectors are of equal length (if not, the shorter vector is padded with 0's on the left), we define $\langle w_k, \ldots, w_0\rangle \prec \langle w'_k, \ldots, w'_0\rangle$ if there is i in $0, \ldots, k$ such that $w_j = w'_j$ for all $k \geqslant j > i$ and $w_i < w'_i$. This order is total and well-founded, that is, there are no infinite descending chains of vectors. The next proposition states that the weight of a process is decreased by reductions involving a responsive name, and leads us to Theorem 5.2, which is the main result of the section.

Proposition 5.2. *Suppose* $\Gamma;\Delta \vdash_1 P$ *and* $P \xrightarrow{\tau\langle a,b\rangle} P'$, *with either* a *or* b *responsive. Then* $\mathrm{wt}(P') \prec \mathrm{wt}(P)$.

Theorem 5.2 (responsiveness). *Let* P *be* $(\Gamma;\Delta)$*-balanced and* $r \in \Delta^\rho$. *Then* P *guarantees responsiveness of* r.

PROOF: Assume $P \xrightarrow{[s']} R$, for any R, and $r \notin s'$. We have to show that $R \overset{[r]}{\Longrightarrow}$. By contradiction, assume not. Let P' be a process with a *minimal* $\mathrm{wt}(\cdot)$ satisfying $R \xrightarrow{[s'']} P'$ for some s'': this P' must exist by well-foundedness of $\prec$. Moreover, $r \notin s''$. Let $s = s' \cdot s''$. By subject reduction we have that P' is $(\Gamma'; \Delta')$-balanced, with $\Gamma' = \Gamma \ominus s$ and $\Delta' = \Delta \ominus s$.

Consider now the normal form of the process P' (Lemma 5.5): $P' \equiv (\nu\, \widetilde{d})(P_1 | \cdots | P_n)$, where every P_i is prime and it must be $n > 1$, as r occurs in both input and output and by rule (T-INP) an output $\overline{r}$ cannot occur under an input on r. Assume, for simplicity, that $\widetilde{d}$ does not contain inert names nor names of type $\mathbf{1}$ (in both cases the proof proceeds similarly).

By Lemma 5.2, there exists a normal derivation of $\Gamma'; \Delta' \vdash_1 P'$. We prove that $\Gamma', \widetilde{d}; \Delta', \widetilde{d} \vdash_1 P_1 | \cdots | P_n$, the proof proceeds by distinguishing two cases. Suppose P' is in normal-form (hence $P' = (\nu\, \widetilde{d})(P_1 | \cdots | P_n)$). Then, in the normal derivation, $\Gamma'; \Delta' \vdash_1 P'$ is deduced from $\Gamma', \widetilde{d}; \Delta', \widetilde{d} \vdash_1 P_1 | \cdots | P_n$ by repeated applications of (T-RES). Suppose P' is not in normal-form. Then in the normal derivation the last rule applied is (T-STR) with premise $\Gamma'; \Delta' \vdash_1 (\nu\, \widetilde{d})(P_1 | \cdots | P_n)$, which, in turn, has been deduced from $\Gamma', \widetilde{d}; \Delta', \widetilde{d} \vdash_1 P_1 | \cdots | P_n$ by repeated applications of (T-RES).

In the normal derivation, rule (T-PAR) must have been applied to infer $\Gamma', \widetilde{d}; \Delta', \widetilde{d} \vdash_1 P_1 | \cdots | P_n$, hence it must be: $(\Gamma', \widetilde{d}) = \bigcup_{i=1,\ldots,n} \Gamma_i$, and $(\Delta', \widetilde{d}) = \bigcup_{i=1,\ldots,n} \Delta_i$, and $\Gamma_i; \Delta_i \vdash_1 P_i$, where Δ_i^{ρ} (resp. Γ_i^{ρ}) are pairwise disjoint and $\mathrm{DG}({\Gamma_i}^{\rho}, {\Delta_i}^{\rho})_{i=1,\ldots,n}$ is acyclic. Moreover, from balancing of Γ and Δ and definition of $\ominus$ we deduce balancing of Γ' and Δ', hence $(\Delta', \widetilde{d})^{\rho} = (\Gamma', \widetilde{d})^{\rho}$. By Proposition 5.1 there is a j such that $P_j = \overline{a}\langle b \rangle$ with a or b responsive name. By (T-OUT) and $\Gamma'; \Delta' \vdash_1 P'$ we have $a \in \Delta', \widetilde{d}$. By $(\Delta', \widetilde{d})^{\omega} \subseteq (\Gamma', \widetilde{d})^{\omega}$ and receptiveness of responsive and ω-receptive names ((T-INP) and (T-REP)), there is a k such that $P_k = (!)a(x).P_k'$. This implies $P' \xrightarrow{\tau\langle a,b \rangle} P''$, with $\mathrm{wt}(P'') \prec \mathrm{wt}(P')$, as either a or b is responsive (Proposition 5.2). But this is a contradiction, because P' was assumed to be the process with minimal weight satisfying $R \xrightarrow{[s'']} P'$. Hence $R \overset{[r]}{\Longrightarrow}$. □

Next, we establish an upper bound on the number of steps that are always sufficient for a given responsive name to be used as subject. This upper bound can be given as a function of the syntactic size of P, written $|P|$, and of name levels in P. A similar result was given in [64] for terminating processes. Here, since we deal with processes that in general may not terminate, the upper bound must be given relatively to a notion of *scheduling* of transitions, that is introduced below.

Definition 5.9 (responsive scheduling). *A* responsive scheduling *is a finite or*

infinite sequence of reductions $P_0 \xrightarrow{\tau\langle a_1,b_1\rangle} P_1 \xrightarrow{\tau\langle a_2,b_2\rangle} \cdots$ *where the bound names in* $\{(a_i,b_i)|i \geqslant 1\}$ *are all distinct from the free names in* P *and for each* $i \geqslant 0$*, either* a_i *or* b_i *is responsive.*

The size of a process P, written $|\,P\,|$, is defined as

$$|\,\mathbf{0}\,| = 0 \qquad |\,a(x).P\,| = 1 + |\,P\,| \qquad |\,(\nu\, c : \mathsf{T})P\,| = |\,P\,|$$
$$|\,\overline{a}\langle b\rangle\,| = 1 \qquad |\,!a(x).P\,| = 1 + |\,P\,| \qquad |\,P|R\,| = |\,P\,| + |\,R\,| \;.$$

Structural congruence preserves the size of a process:

Proposition 5.3. *Suppose* $P \equiv R$*, then* $|\,P\,| = |\,R\,|$*.*

Proof: The proof is straightforward by induction on the derivation of $P \equiv R$. □

Theorem 5.3. *Let* P *be* $(\Gamma;\Delta)$*-balanced and* $r \in \Delta^\rho$ *and let* k *be the maximal level of names appearing in active output actions of* P*. Then there is at least one responsive scheduling that contains a reduction with* r *as subject. Moreover, in all such schedulings, the number of reductions preceding the reduction on* r *is upper-bounded by* $|\,P\,|^{k+1}$*.*

The proof relies on Theorem 5.2 (responsiveness), which ensures a communication on r. The maximal number of communications that can precede the reduction on r is evaluated by considering that each reduction can increase the number of outputs – that is of potential reductions – in the continuation, but this increase is limited by the initial size of the process (see Section A.3 for a detailed proof).

5.3.5 An extension: subtyping

The presence of responsive and ω-receptive names, and the presence of levels, naturally induce a subtyping relation that we do not have yet introduced only for presentation convenience. Intuitively, ω-names can be used in place of responsive names, if their levels and carried type comply. Subtyping is contravariant on the type of carried objects. This leads to defining $<$ as the least reflexive and transitive relation on types generated by the following rule:

$$(\text{SUB}) \quad \frac{\mathsf{T} > \mathsf{S} \quad \mathsf{U} < \mathsf{U}'}{\mathsf{T}^{\mathsf{U}} < \mathsf{S}^{\mathsf{U}'}} \qquad \mathsf{U} < \mathsf{U}' \text{ holds in these cases: } \begin{cases} [\omega,k] < [\,\cdot\,,k'] & \text{if } k \leqslant k' \\ [\rho,k] < [\rho,k'] & \text{if } k \leqslant k' \;. \end{cases}$$

In the typing judgements, we have to abandon the sorting system, which does not reconcile well with subtyping, and consider contexts Γ, Δ that explicitly associate names with types. The rule for output is modified as expected. Intuitively, along a channel of type T^{U} we can always send something of type $\mathsf{S} < \mathsf{T}$, thus subtyping

can be used for modifying rules in Table 5.6, and in particular for substituting rule (T-Out) with the following:

$$\frac{\Delta \vdash a : \mathsf{T}^{\mathsf{U}} \quad \Delta \vdash b : \mathsf{S} \quad \mathsf{S} < \mathsf{T} \quad \Delta^{\rho} \ominus \{a, b\} = \emptyset}{\emptyset; \Delta \vdash_1 \overline{a}\langle b \rangle} \ .$$

5.3.6 Type system $\vdash_1$ vs Strong Normalization and Linear Liveness

Closely related to the system presented in this section are a series of papers by Berger, Honda and Yoshida on linearity-based type systems. In [136], they introduce a type system that guarantees termination and determinacy of pi-calculus processes, i.e. *Strong Normalization* (SN). Our techniques of system $\vdash_1$ are actually close to theirs, as far as the linearity conditions and cycle-detection graphs are concerned (see also the type system in [134]). However SN is stronger than responsiveness, in particular SN implies responsiveness on all linear names under a balancing condition. In fact, the system in [136] is stricter than $\vdash_1$, e.g., it does not allow any form of nondeterminism and divergence, as these features would obviously violate SN. Yoshida's type system in [135], in turn a refinement of the systems in [136] and [19], is meant to ensure a *Linear Liveness* property, meaning that the considered process eventually prompts for a free output at a given channel. This property is related to responsiveness, the difference being that Linear Liveness does not imply synchronization, as the corresponding input might not become available. Two kinds of names are considered in [135]: linear (used exactly once) and *affine* (used at most once). Linear subjects carrying linear objects are forbidden and internal mobility is assumed: only restricted names can be passed around.

5.4 The type system $\vdash_2$

The type system presented in Section 5.3 puts rather severe limitations on nesting of input actions and multiple use of names. These limitations stem from the "immediate receptiveness" and linearity conditions imposed on responsive names. For instance, the following encoding of internal choice $\overline{r}\langle a \rangle \oplus \overline{r}\langle b \rangle$, where r is responsive and a, b inert, is not well-typed

$$(\nu c)(\overline{c}\langle a \rangle | \overline{c}\langle b \rangle \mid c(x).\overline{r}\langle x \rangle) \ . \tag{5.3}$$

In fact, c cannot be a ω-receptive name, because the input is not replicated, nor a responsive name, because it is used twice in output.

Limitations are also built-in in process syntax, as for example replicated outputs, that clearly violate linearity, are absent. These might be useful to encode situations like a process receiving from r a value y and storing it into a variable a, where reading from a means doing an input on a:

$$(\nu\, a)\big(r(x).!\overline{a}\langle x\rangle | a(y).P\big)\ . \tag{5.4}$$

For another example, a process that receives two values in a fixed order from two return channels, r_1 and r_2, and then outputs the max along s, is not well-typed

$$r_1(x_1).r_2(x_2).\texttt{if}\ x_1 \geqslant x_2\ \texttt{then}\ \overline{s}\langle x_1\rangle\ \texttt{else}\ \overline{s}\langle x_2\rangle\ . \tag{5.5}$$

In fact, the previous type system does not allow input actions guarded by other inputs.

We present below a new type system, which we indicate with $\vdash_2$, that overcomes the limitations discussed above. In fact, we will trade off flexibility for expressiveness in terms of encodable functions, as only certain patterns of (tail-)recursion will be well-typed in the new system.

5.4.1 Syntax and operational semantics

We extend the syntax of processes by introducing replicated output and the syntax of types by introducing a new responsive usage of names, ρ^+, as follows:

$$\begin{array}{lcl} P & ::= & \cdots \mid !\overline{a}\langle b\rangle \\ \mathsf{U} & ::= & \cdots \mid [\rho^+, k]\ . \end{array}$$

A name $a : \mathsf{T}^{[\rho^+,k]}$ is called *+-responsive*, as it is meant to be used *at least once* as subject of a communication. Therefore now we consider three different usages: ρ (for names used once), ρ^+ (for names used at least once) and ω (for names used an undefined number of times). We point out that responsive names are not subsumed by +-responsive: in particular, as we shall see, the conditions on the type of carried objects are more liberal for responsive names. Operational semantics is enriched by adding the obvious rule for replicated output.

5.4.2 Overview of the system

We give here an informal overview of the type system. Judgments are of the form $\Gamma;\Delta \vdash_2 P$ where in Γ and Δ each +-responsive name a is annotated with a *capability* t, written a^t. A capability t can be one of four kinds: n (*null*), s (*simple*), m (*multiple*) and p (*persistent*). Informally, capabilities have the following meaning

(in the examples below, we ignore object parts of some actions and assume b is a (+-)responsive name):

- a^{n} indicates that a cannot be used at all. This capability has been introduced to uniformly account for +-responsive names that disappear after being used as subjects.
- a^{s} indicates that a appears at least once, but never under a replication nor as subject of a replicated action. Examples: $a.P$, $b.a.P$, $\overline{a}$ and $b.\overline{a}$.
- a^{m} indicates that a appears at least once, even under replication, but never as subject of a replicated action. Examples: $a.P|a.Q$, $!b.a.P$ and $!b.\overline{a}$.
- a^{p} indicates that a only appears as subject of a replicated action. Examples: $!a.P$, $!\overline{a}$, $b.!\overline{a}$ and $!b.!\overline{a}$.

Note that a name a may be given distinct capabilities in input (Γ) and output (Δ). E.g. one may have, again ignoring the object parts, $\Gamma;\Delta \vdash_2 !a.P|\overline{a}|\overline{a}$, where $a^{\mathsf{p}} \in \Gamma$ and $a^{\mathsf{m}} \in \Delta$. Next we illustrate and motivate the constraints on name usages realized by the typing rules and by the balancing conditions discussed later on:

(1) Names with input capability s (simple) occur exactly once in input subject position. This constraint has been introduced for the sake of simplicity. Allowing to use such names more than once in input requires to verify that the number of inputs involving each name is smaller than the number of outputs. E.g. the process (a and b +-responsive names)

$$a \,|\, a.b \,|\, \overline{a} \,|\, \overline{b} \xrightarrow{\tau} a.b \,|\, \overline{b} \not\xrightarrow{\tau} \quad . \tag{5.6}$$

has to be discarded, because a is used twice in input and only once in output.

(2) If $a^{\mathsf{m}} \in \Gamma$ then a can appear more than once in input subject position and it is required that $a^{\mathsf{p}} \in \Delta$. This is necessary to avoid deadlocks arising from not having enough output actions of subject a, like in (5.6). This is avoided if a appears in replicated output subject: $a \,|\, a.b \,|\, !\overline{a} \,|\, \overline{b}$.

(3) If $a^t \in \Gamma$ and a carries (+-)responsive names, then $t = \mathsf{p}$. This is to avoid deadlocks arising from having too many outputs of subject a that carry (+-)responsive names, like in (a +-responsive, b and d (+-)responsive names):

$$\overline{a}\langle b\rangle|\overline{a}\langle d\rangle|a(x).\overline{x}|b|d \xrightarrow{\tau}\xrightarrow{\tau} \overline{a}\langle d\rangle|d \ .$$

(4) Concerning a^{p}, we ban names persistent both in input and in output. This is a necessary condition for avoiding divergences involving +-responsive names like in (a +-responsive)

$$!\overline{a} \,|\, !a(x).P \ .$$

Moreover, names with capability p (persistent) are required to occur *exactly once* in subject position (either in input or in output). This is necessary to avoid deadlock situations due to nondeterminism like in (b and c +-responsive)

$$!a.\overline{b}\,|\,!a.\overline{c}\,|\,\overline{a}\,|\,c\,|\,b \xrightarrow{\tau} !a.\overline{b}\,|\,!a.\overline{c}\,|\,\overline{b}\,|\,c\,|\,b \xrightarrow{\tau} !a.\overline{b}\,|\,!a.\overline{c}\,|\,c \not\xrightarrow{\tau}$$

where a communication on c would never happen. To preserve this conditions at run-time, we have also to forbid

(i) replicated actions guarded by replicated inputs. E.g. it is necessary to avoid situations like

$$!a.!b\,|\,\overline{a} \xrightarrow{\tau} !a.!b\,|\,!b$$

where the RHS violates linearity of +-responsive names used as subjects of replicated inputs;

(ii) persistent names passed around as objects (this for avoiding "collisions" of capabilities).

(5) Names occurring under an (either simple or replicated) input must be of smaller level than the input subject. The role of this condition is twofold, now. Under replicated inputs, it avoids infinite delays, like in the first system. Under simple inputs, it serves to avoid cyclic waiting, like in (a, b (+-)responsive):

$$a.\overline{b}|b.\overline{a}\ .$$

This was achieved by the use of dependency graphs in the first system. As announced in Remark 5.2, however, there appears to be no meaningful extension of this notion of graph in the present system. In particular, acyclicity of the graph might not be preserved by reductions. E.g. consider the process

$$b(x).\overline{a}\langle x\rangle|c(x).a(y).\overline{x}\langle y\rangle|\overline{c}\langle b\rangle\ .$$

Its graph is acyclic, but after a reduction on c the process become

$$b(x).\overline{a}\langle x\rangle|a(y).\overline{b}\langle y\rangle$$

and the corresponding dependency graph has a cycle involving a and b. As a by-product of discarding the dependency graph, we achieve a simplification of the typing rule for parallel composition. However, this rather crude use of levels to ban cyclic waiting is also the cause of the reduced expressiveness in terms of typable functions.

Finally, we introduce a "syntactic" restriction. As +-responsive names used once and more than once in output are treated in the same manner, we reserve capability s for inputs, and use only m and p for outputs. This choice also alleviates some technicalities in the proof of the subject reduction theorem.

5.4.3 The typing rules

We first introduce some additional notation. In what follows, we denote by $\mathrm{is}(P)$ the set of either *bound* or *free* names used in input subject position in P. Contexts Γ and Δ are sets of annotated names of the form a^t, where t is a capability. Each name occurs at most once in a context. +-responsive names are annotated with one of the four capabilities n, s (only in Γ), m or p, while non-+-responsive names are always annotated with a default "$-$" capability; when convenient a^- is abbreviated simply as a. Union and intersection of two contexts, written $\Gamma_1 \cup \Gamma_2$ and $\Gamma_1 \cap \Gamma_2$, are defined only if the contexts agree on capabilities of common names, that is whenever $a^{t_i} \in \Gamma_i$ for $i = 1, 2$ then $t_1 = t_2$. We write Γ_1, Γ_2 in place of $\Gamma_1 \cup \Gamma_2$ if $\Gamma_1 \cap \Gamma_2 = \emptyset$; while Γ_1, a^t abbreviates $\Gamma_1, \{a^t\}$. For any context Γ and capability t, we define $\Gamma^t \stackrel{\triangle}{=} \{a | a^t \in \Gamma\}$. The set of names $\Gamma^{\rho^+} \stackrel{\triangle}{=} \{\, a \mid a \text{ is +-responsive and } a^t \in \Gamma \text{ for some } t \neq \mathsf{n} \,\}$ and Γ^ρ, Γ^ω (defined similarly) will also be useful. The typing rules are presented in Table 5.8. We briefly comment on the rules by considering the five points discussed above in turn.

(1) is ensured in (T_+-Par) by checking the disjointness of Γ_1^{s} and Γ_2^{s} and in (T_+-Inp), because $a \notin \Gamma$;

(2) is ensured in (T_+-Par) by $\Gamma^{\mathsf{m}} \cap \Delta^{\mathsf{m}} = \emptyset$. Note that a simpler constraint would be to require $\Gamma^{\mathsf{m}} \subseteq \Delta^{\mathsf{p}}$, but, subject reduction at the labeled transition system level (which is necessary for proving subject reduction) would be violated by the process below

$$!a.b \xrightarrow{a} !a.b \mid b$$

where rule (T_+-Par) would require b to be used as subject of a persistent output in the rhs;

(3) is ensured in (T_+-Inp) by checking that +-responsive names used as subject of non-replicated inputs cannot carry (+-)responsive objects;

(4) all rules for input ensure that received names cannot be used as subjects of replicated outputs (by checking the capability of the received objects); moreover, (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$) ensure that inputs on persistent names cannot be guarded by replicated inputs (by checking $\Gamma^{\mathsf{p}} = \emptyset$). Rules for outputs check that persistent names cannot be passed around. Finally, (T_+-Par) verifies the linear usage of persistent names in input and output subject (by checking the

disjointness of $\Gamma_1{}^{\mathsf{p}}$ and $\Gamma_2{}^{\mathsf{p}}$ and of $\Delta_1{}^{\mathsf{p}}$ and $\Delta_2{}^{\mathsf{p}}$) and bans the usage of names with persistent capability both in input and output (by checking the disjointness of Γ^{p} and Δ^{p});

(5) is ensured by rules (T_+-Inp), (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$) by comparing the levels of the input prefix against the levels of each nested inputs and outputs.

Finally, linear usage of responsive names is ensured by the typing rules for replicated inputs by checking the emptiness of Δ^{ρ} and Γ^{ρ}, by rule (T_+-Par), by checking $\Gamma_1{}^{\rho} \cap \Gamma_2{}^{\rho} = \Delta_1^{\rho} \cap \Delta_2^{\rho} = \emptyset$, and by ($\mathrm{T}_+$-Inp) ($a \notin \Gamma$).

5.4.4 Properties of type system $\vdash_2$

Subject reduction carries over to the new system, modulo a small notational change. For Γ a typing context and V a set of names let us denote by $\Gamma \ominus^+ V$ the typing context obtained by removing from Γ each a^t such that $a \in V$. Let us denote by $\mathrm{on}(P)$ the set of names occurring *free* in output, subject or object, position in P.

Theorem 5.4 (subject reduction for system $\vdash_2$). *$\Gamma; \Delta \vdash_2 P$ and $P \xrightarrow{[a]} P'$ imply $\Gamma'; \Delta' \vdash_2 P'$, with $\Gamma' = \Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P'))$ and $\Delta' = \Delta \ominus^+ (\{a\} \setminus \mathrm{on}(P'))$.*

The analogous of Proposition 5.1 and 5.2 for system $\vdash_2$ holds (formal proofs can be found in Section A.4). In what follows we consider the extension of $\mathrm{wt}(\cdot)$ to the system $\vdash_2$, written $\mathrm{wt}^+(\cdot)$, defined in Table 5.9.

Proposition 5.4. *Suppose P is $(\Gamma; \Delta)$-strongly balanced with $\Delta^{\rho} \cup \Gamma^{\rho^+} \neq \emptyset$. Then $P \xrightarrow{\tau\langle a,b\rangle}$ with either a or b (+-)responsive name.*

Proposition 5.5. *$\Gamma; \Delta \vdash_2 P$ and $P \xrightarrow{\tau\langle a,b\rangle} P'$ with either a or b (+-)responsive, implies $\mathrm{wt}^+(P') \prec \mathrm{wt}^+(P)$.*

The balancing requirements are now more stringent. They include those for responsive and ω-receptive names necessary in the first system (condition 1 below). Concerning +-responsive names, "perfect balancing" between input and output is required only for those names that carry (+-)responsive names (condition 2). Moreover, the same requirements apply also to restricted +-responsive names (condition 3).

In the following, we need some additional notations. Given a set of names V let us define $V^{\dagger} = \{a \in V \mid a : \mathsf{T} \text{ and } \mathsf{T} \text{ is of the form } (\mathsf{S}^{[\mathsf{u},k]})^{[\mathsf{u}',h]} \text{ with } \mathsf{u} \in \{\rho, \rho^+\}\}$. Define $\mathrm{r}_{\mathrm{i}}^+(P)$ (resp. $\mathrm{r}_{\mathrm{o}}^+(P)$) as the set of restricted +-responsive names in P occurring in an input (resp. output) action in P, even underneath a replication. We have the following definition and results.

$$(\mathrm{T}_+\text{-Inp})\ \frac{a : \mathsf{T}^{[\mathsf{u},k]} \text{ with } \mathsf{u} \neq \omega \quad b : \mathsf{T} \quad \forall c \in \mathrm{os}(P) \cup \mathrm{is}(P) : \mathrm{lev}(c) < k \quad \Gamma^\omega = \emptyset \quad a \text{ +-responsive implies } b \text{ not (+-)responsive} \quad \Gamma; \Delta, b^{t'} \vdash_2 P \quad t \neq \mathsf{n},\mathsf{p} \quad t' \neq \mathsf{n},\mathsf{p}}{\Gamma, a^t; \Delta \vdash_2 a(b).P}$$

$$(\mathrm{T}_+\text{-Rep})\ \frac{a : \mathsf{T}^{[\omega,k]} \quad b : \mathsf{T} \quad \Delta^\rho = \Delta^{\rho^+} = \emptyset \quad \emptyset; \Delta, b^{t'} \vdash_2 P \quad t' \neq \mathsf{n},\mathsf{p} \quad b \text{ (+-)responsive implies } \forall c \in \mathrm{os}(P) \cup \mathrm{is}(P) : \mathrm{lev}(c) < k}{a^-; \Delta \vdash_2 !a(b).P}$$

$$(\mathrm{T}_+\text{-Rep}^\mathsf{p})\ \frac{a : \mathsf{T}^{[\rho^+,k]} \quad b : \mathsf{T} \quad \Gamma^\ell = \emptyset \text{ for } \ell \in \{\rho,\omega,\mathsf{s},\mathsf{p}\} \quad \Delta^{\ell'} = \emptyset \text{ for } \ell' \in \{\mathsf{p},\rho\} \quad \Gamma; \Delta, b^t \vdash_2 P \quad t \neq \mathsf{n},\mathsf{p} \quad \forall c \in \mathrm{os}(P) \cup \mathrm{is}(P) : \mathrm{lev}(c) < k}{\Gamma, a^\mathsf{p}; \Delta \vdash_2 !a(b).P}$$

$$(\mathrm{T}_+\text{-Out})\ \frac{a : \mathsf{T}^\mathsf{U} \quad b : \mathsf{T} \quad \Delta^\rho = \Delta^{\rho^+} = \emptyset \quad t' \neq \mathsf{n},\mathsf{p} \quad t \neq \mathsf{n},\mathsf{p}}{\emptyset; \Delta, a^t, b^{t'} \vdash_2 \overline{a}\langle b\rangle}$$

$$(\mathrm{T}_+\text{-Out}^\mathsf{p})\ \frac{a : \mathsf{T}^{[\rho^+,k]} \quad b : \mathsf{T} \quad \Delta^\rho = \Delta^{\rho^+} = \emptyset \quad b \text{ not (+-)responsive}}{\emptyset; \Delta, a^\mathsf{p}, b^- \vdash_2 !\overline{a}\langle b\rangle}$$

$$(\mathrm{T}_+\text{-Nil})\ \frac{\Delta^\rho = \Delta^{\rho^+} = \emptyset}{\emptyset; \Delta \vdash_2 \mathbf{0}} \qquad (\mathrm{T}_+\text{-Res})\ \frac{a : \mathsf{T}^\mathsf{U} \quad \Gamma, a^t; \Delta, a^{t'} \vdash_2 P}{\Gamma; \Delta \vdash_2 (\nu\, a)P}$$

$$(\mathrm{T}_+\text{-Res-}\mathbb{L})\ \frac{a : \mathbb{L} \quad \Gamma; \Delta \vdash_2 P}{\Gamma; \Delta \vdash_2 (\nu\, a)P} \qquad (\mathrm{T}_+\text{-Res-I})\ \frac{a : \mathsf{I} \quad \Gamma; \Delta, a^- \vdash_2 P}{\Gamma; \Delta \vdash_2 (\nu\, a)P}$$

$$(\mathrm{T}_+\text{-Weak-}\Gamma)\ \frac{\Gamma; \Delta \vdash_2 P}{\Gamma, a^\mathsf{n}; \Delta \vdash_2 P} \qquad (\mathrm{T}_+\text{-Weak-}\Delta)\ \frac{\Gamma; \Delta \vdash_2 P}{\Gamma; \Delta, a^\mathsf{n} \vdash_2 P}$$

$$(\mathrm{T}_+\text{-Par})\ \frac{\Gamma = \Gamma_1 \cup \Gamma_2 \quad \Delta = \Delta_1 \cup \Delta_2 \quad \Gamma_i; \Delta_i \vdash_2 P_i \quad (i = 1,2) \quad \Gamma_1^\ell \cap \Gamma_2^\ell = \emptyset \text{ for } \ell \in \{\rho,\mathsf{s},\mathsf{p}\} \quad \Delta_1^{\ell'} \cap \Delta_2^{\ell'} = \emptyset \text{ for } \ell' \in \{\rho,\mathsf{p}\} \quad \Gamma^\mathsf{p} \cap \Delta^\mathsf{p} = \emptyset \quad \Gamma^\mathsf{m} \cap \Delta^\mathsf{m} = \emptyset}{\Gamma\,; \Delta \ \vdash_2 \ P_1 | P_2}$$

Table 5.8: Typing rules of $\vdash_2$.

Definition 5.10 (strongly balanced processes). *A process P is* $(\Gamma;\Delta)$-strongly balanced *if $\Gamma;\Delta \vdash_2 P$ and the following conditions hold:*

(1) $\Gamma^\rho = \Delta^\rho$ *and* $\Delta^\omega \subseteq \Gamma^\omega$*;*

(2) $\Gamma^{\rho^+} \subseteq \Delta^{\rho^+}$ *and* $(\Delta^{\rho^+})^\dagger \subseteq (\Gamma^{\rho^+})^\dagger$*;*

(3) $\mathrm{r_i^+}(P) \subseteq \mathrm{r_o^+}(P)$ *and* $(\mathrm{r_o^+}(P))^\dagger \subseteq (\mathrm{r_i^+}(P))^\dagger$.

P is said strongly balanced *if it is $(\Gamma;\Delta)$-strongly balanced for some Γ and Δ.*

$\mathrm{wt}^+(\mathbf{0}) = 0$	$\mathrm{wt}^+(!\overline{a}\langle b\rangle) = 0$	$\mathrm{wt}^+(!a(b).P) = 0$
$\mathrm{wt}^+(\overline{a}\langle b\rangle) = 0_{\mathrm{lev}(a)}$		$\mathrm{wt}^+(P\|R) = \mathrm{wt}^+(P) + \mathrm{wt}^+(R)$
$\mathrm{wt}^+((\nu a)P) = \mathrm{wt}^+(P)$		$\mathrm{wt}^+(a(b).P) = \mathrm{wt}^+(P) + 0_{\mathrm{lev}(a)}$

Table 5.9: $\mathrm{wt}^+(P)$

The proof of the following theorem is non-trivial, as strong balancing is preserved through reductions only up to certain transformations on processes. The lemma below ensures that such transformations re-establish strong balancing.

Lemma 5.3. *Suppose P $(\Gamma;\Delta)$-strongly balanced and $P \xrightarrow{\tau\langle a,b\rangle} P'$ with P' non strongly balanced. Let be $\Gamma';\Delta' \vdash_2 P'$. Then*

(1) $a \in ({\Gamma'}^{\rho^+} \setminus {\Delta'}^{\rho^+}) \cup (\mathrm{r_i}^+(P') \setminus \mathrm{r_o}^+(P'))$;
(2) $P \equiv (\nu\, \widetilde{d})(!a(x).R \,|\, R')$ *and* $a \notin \mathrm{in}(R, R')$;
(3) $P' \equiv (\nu\, \widetilde{d})(!a(x).R \,|\, R[{}^{b}\!/\!_{x}] \,|\, R'')$ *and* $a \notin \mathrm{in}(R, R'', R[{}^{b}\!/\!_{x}])$;
(4) $P'' = (\nu\, \widetilde{d})(R[{}^{b}\!/\!_{x}] \,|\, R'')$ *is strongly balanced.*

Theorem 5.5 (responsiveness for system $\vdash_2$). *Suppose P is $(\Gamma;\Delta)$-strongly balanced and $r \in \Delta^{\rho} \cup {\Gamma^{\rho}}^{+}$. Then P guarantees responsiveness of r.*

Proof: Suppose that $P \xrightarrow{[s]} P'$, with P' having a minimal weight among processes reachable from P with $r \notin s$ (this P' must exist by well-foundedness of $\prec$). Let $s = a_1 \cdots a_n$, and consider the sequence of reductions leading to P':

$$P = P_0 \xrightarrow{[a_1]} P_1 \xrightarrow{[a_2]} \cdots \xrightarrow{[a_n]} P_n = P' \tag{5.7}$$

By $\Gamma;\Delta \vdash_2 P$ and subject reduction we have that $\Gamma_i;\Delta_i \vdash_2 P_i$ for $i = 0, ..., n$, where $\Gamma_0 = \Gamma$ and $\Delta_0 = \Delta$ and $\Gamma_i = \Gamma_{i-1} \ominus^+ (\{a_i\} \setminus \mathrm{in}(P_i))$ and $\Delta_i = \Delta_{i-1} \ominus^+ (\{a_i\} \setminus \mathrm{on}(P_i))$ for $i > 0$. We prove that $P' \xRightarrow{[r]}$ by induction on the number k of non-strongly balanced processes in the sequence of reductions (5.7), that is

$$k = \big|\{i \,|\, 0 \leqslant i \leqslant n \text{ and } P_i \text{ is not } (\Gamma_i, \Delta_i)\text{-strongly balanced }\}\big|.$$

$k = 0$**:** Then P' is strongly balanced. Since $r \in ({\Delta_n}^{\rho} \cup {\Gamma_n}^{\rho^+})$ (as $r \notin s$), by Proposition 5.4, $P' \xrightarrow{\tau\langle a,b\rangle} P''$, with either a or b (+-)responsive, and, by Proposition 5.5, $\mathrm{wt}^+(P'') \prec \mathrm{wt}^+(P')$. Hence $a = r$, because P' was assumed to have minimal weight among the processes reachable from P without using r as subject.

$k > 0$**:** Let P_j $(j > 0)$ be the leftmost non-strongly balanced process in the sequence (5.7). Consider the reduction $P_{j-1} \xrightarrow{[a_j]} P_j$. Process P_{j-1} is strongly balanced while P_j is not, thus, by Lemma 5.3 (1, 2), $a_j \in ({\Gamma_j}^{\rho^+} \setminus {\Delta_j}^{\rho^+}) \cup$

$(\mathrm{r_i^+}(P_j) \setminus \mathrm{r_o^+}(P_j))$ and $P_{j-1} \equiv (\nu\, \widetilde{d})(!a_j(x).R \mid S)$, with $a_j \notin \mathrm{in}(R,S)$. Again by Lemma 5.3 (3), $P_j \equiv (\nu\, \widetilde{d})(!a_j(x).R \mid R[c/x] \mid S')$ with $a_j \notin \mathrm{in}(R, R[c/x], S')$. Moreover $P' \equiv (\nu\, \widetilde{d'})(!a_j(x).R \mid P''')$ with $a_j \notin \mathrm{in}(P''')$. Now, the process $P'_j = (\nu\, \widetilde{d})(R[c/x] \mid S')$, obtained by erasing the term $!a_j(x).R$ from P_j, is strongly balanced (Lemma 5.3 (4)), and it holds $P'_j \xrightarrow{[a_{j+1}]} \cdots \xrightarrow{[a_n]} P'_n = P''$, with $P'' \equiv (\nu\, \widetilde{d'})\, P'''$. This sequence has $\leqslant k-1$ unbalanced processes, and moreover P'' has minimal weight among the processes reachable from P'_j without using r as subject because $\mathrm{wt}^+(P'') = \mathrm{wt}^+(P')$ (by definition of $\mathrm{wt}^+(\cdot)$ we have $\mathrm{wt}^+((\nu\, \widetilde{d'})(!a_j(x).R \mid P''')) = \mathrm{wt}^+((\nu\, \widetilde{d'})P'''))$. Then, by inductive hypothesis, $P'' \stackrel{[r]}{\Longrightarrow}$, which implies $P' \stackrel{[r]}{\Longrightarrow}$.

□

Example 5.4.1. Let us now examine a few examples. In what follows, unless otherwise stated we assume that x, y are of type inert, that a, b, c are $+$-responsive and that r, s are responsive. Conditions on levels are ignored when obvious.

Process (5.3) at the beginning of the section is well-typed with c of capability multiple in output and simple in input; it is strongly balanced if put in parallel with an appropriate context of the form $r(x).P$. Process (5.4) is well-typed with a of capability persistent in output and simple in input (also, P must be assumed strongly balanced, and not containing free persistent inputs or names of level greater than a's); it is strongly balanced if put in parallel with $\overline{r}\langle x\rangle$. Process (5.5) is well-typed assuming r_1 and r_2 of capability simple in input and x_1, x_2 natural number variables (the obvious extension of the system with $if - then - else$ and naturals is here assumed); again, it is strongly balanced if put in parallel with an appropriate context.

The next two examples involve non-linear usages of $+$-responsive names arising from replication and reference passing. We mention these examples also because they will help us to compare our system to existing type systems (see § 5.4.5 below). The first example involves only replication, object parts play no role:

$$!a.\overline{b} \mid \overline{a} \mid b\ . \tag{5.8}$$

The above process is strongly balanced under the assumption that a has capability persistent in input and multiple in output, and b has capability simple in input and multiple in output; also, the level of b must be less than a's. In the next example, an agent "looks up" a directory a to get the address of a service b, and then calls this service:

$$!a(z).\overline{z}\langle b\rangle \mid (\nu\, r)(\overline{a}\langle r\rangle \mid r(w).\overline{w}) \mid b\ . \tag{5.9}$$

This process is strongly balanced under the assumption that: a is persistent in input and multiple in output; b is simple in input and multiple in output; also, it must be $\mathrm{lev}(b) < \mathrm{lev}(r) < \mathrm{lev}(a)$. The variant where b is replaced by $!b$ is also strongly balanced; in this case b is persistent in input.

5.4.5 Type system $\vdash_2$ vs Lock-Freedom

There is a series of works by Kobayashi where type systems for livelock and deadlock freedom are proposed [95, 96, 97]. In [95, 96] the proposed systems guarantee that, under suitable fairness assumptions, certain actions are lock-free, i.e. are deemed to succeed in synchronization, if they become available. The system in [97] is a further refinement, but the resulting system cannot be used to ensure some interesting properties like termination of processes or ensuring some kind of communications will eventually succeed no matter whether the process diverges.

The systems in [95, 96] can be used for ensuring responsive usage of responsive names but they do not always work for +-responsive ones. Let's rapidly introduce the systems and discuss their limits. In Kobayashi's works channel types are defined in terms of *usages*: roughly, CCS-like expressions on the alphabet $\{I, O\}$, that define the order in which each channel must be used in input (I) and in output (O). Each I/O action is annotated with an *obligation* level, related to when the action must become available, and a *capability* level, related to when the action must succeed in synchronization if it becomes available. A level can be a natural number or infinity, the latter used to annotate actions that are not guaranteed either to become available (infinite obligation) or to succeed in synchronization (infinite capability). This scheme is fairly general, allowing e.g. for typing of shared-memory structures such as locks and semaphores, which are outside the scope of our systems. Our responsive types can be encoded into lock-freedom types as $I_{c_1}^{o_1} \,|\, O_{c_2}^{o_2}$, with finite obligation and capability levels (resp. o_i and c_i for $i = 1, 2$). The parallel composition $I \,|\, O$ ensures a linear usage. On the other hand, it appears that our +-responsive types cannot, in general, be encoded into lock-freedom types. More precisely, one can exhibit processes well-typed in our system two and containing +-responsive names that cannot be assigned a finite capability in Kobayashi's systems. For example, both the process (5.8) and the "service-lookup" (5.9) are well-typed (in fact, strongly balanced) in our system two, under a typing context where b is +-responsive. They are not in the systems of [95, 96], under any type context that assigns to b a finite capability. The reason is that, in [95, 96], a finite-capability input on b is required to be balanced by an instance of a finite-obligation output on $\overline{b}$. But this instance cannot be statically determined

in the given processes because of the replicated input on a. In the latest version of Kobayashi's `TyPiCal` tool [98], released after the publication of [7], processes of this form can however be handled.

Another difference from [95, 96] is that these systems partly rely on a form of dynamic analysis which is performed on types: the *reliability* condition on usages, which roughly plays the same role played in our systems by balancing, is checked via a reduction to the reachability problem for Petri nets. As previously remarked, our systems are entirely static. Moreover, in [95, 96] services where the number of reductions before synchronization depends on the argument that is passed (e.g. the encoding of recursive functions) are not easily dealt with, unless extensions with dependent types are considered (see [95]).

5.5 Encoding the Structured Orchestration Language

ORC [59] is one of the recent proposed languages for WS orchestration that supports a structured model of concurrent and distributed programming. This model assumes that basic services, like sequential computation and data manipulation, are implemented by primitive sites, and provides constructs to orchestrate the concurrent invocation of sites to achieve a given goal. In this section we briefly introduce ORC and show that it can be encoded into π-calculus processes well-typed in system $\vdash_2$.

5.5.1 ORC: syntax and operational semantics

For the sake of simplicity, we consider a monadic version of this calculus, and we suppose that inert names, $c, c', \ldots$, are the only *data values* that can be exchanged among ORC services. We also consider a countable set of *variables* $x, y, \ldots$.

ORC terms, ranged over $f, g, \ldots$, are defined by the grammar in Table 5.10.

In the table, M is a *site* name, p is a parameter (either a variable x or a name c) and for every expression name E there exists a declaration $E(x) \stackrel{\triangle}{=} f$, where x is the formal parameter and $\mathrm{fv}(f) \subseteq \{x\}$. The primitives can be informally explained as follows. Each closed expression *publishes* (returns) a (finite or infinite) sequence of zero or more values. A site call $M(c)$ always publishes a predefined value $F_M(c)$. An expression call $E(c)$ publishes the values returned by $f[c/x]$ if $E(x) \stackrel{\triangle}{=} f$. The expression $let(c)$ publishes the value c. In $f > x > g$, the execution of f is started, and every value c published by f triggers a new instance of g, $g[c/x]$; the sequence of values produced by all these instances of g running in parallel is published. In $f|g$ a sequence obtained by interleaving values produced by f and g is published. In

Parameter	p	::=	x	*Variable*
			$\mid c$	*Value*
Term	f, g	::=	$\mathbf{0}$	*Inaction*
			$\mid M(p)$	*Site call*
			$\mid E(p)$	*Expression call*
			$\mid let(p)$	*Publication*
			$\mid f > x > g$	*Sequential composition*
			$\mid f \mid g$	*Symmetric parallel composition*
			$\mid g \ where \ x :\in f$	*Asymmetric parallel composition*

Table 5.10: ORC's syntax.

$$(\text{PUB}) \ \frac{}{let(c) \overset{!c}{\hookrightarrow} \mathbf{0}} \qquad (\text{SITE}) \ \frac{}{M(c) \overset{\tau}{\hookrightarrow} let(F_M(c))}$$

$$(\text{PAR1}) \ \frac{f \overset{\lambda}{\hookrightarrow} f'}{f \mid g \overset{\lambda}{\hookrightarrow} f' \mid g} \qquad (\text{PAR2}) \ \frac{g \overset{\lambda}{\hookrightarrow} g'}{f \mid g \overset{\lambda}{\hookrightarrow} f \mid g'}$$

$$(\text{SEQ1}) \ \frac{f \overset{\lambda}{\hookrightarrow} f' \quad \lambda \neq !c}{f > x > g \overset{\lambda}{\hookrightarrow} f' > x > g} \qquad (\text{SEQ2}) \ \frac{f \overset{!c}{\hookrightarrow} f'}{f > x > g \overset{\tau}{\hookrightarrow} (f' > x > g) \mid g[c/x]}$$

$$(\text{WH1}) \ \frac{f \overset{\lambda}{\hookrightarrow} f' \quad \lambda \neq !c}{g \ where \ x :\in f \overset{\lambda}{\hookrightarrow} g \ where \ x :\in f'} \qquad (\text{WH2}) \ \frac{f \overset{!c}{\hookrightarrow} f'}{g \ where \ x :\in f \overset{\tau}{\hookrightarrow} g[c/x]}$$

$$(\text{WH3}) \ \frac{g \overset{\lambda}{\hookrightarrow} g'}{g \ where \ x :\in f \overset{\lambda}{\hookrightarrow} g' \ where \ x :\in f} \qquad (\text{DEF}) \ \frac{E(x) \triangleq f}{E(p) \overset{\tau}{\hookrightarrow} f[p/x]}$$

where in (SITE) $F_M(c)$ is any function on data values.

Table 5.11: ORC operational semantics.

$g \ where \ x :\in f$ the values produced by g are published; however, the execution of f and g is started in parallel, and each subterm of g that depends on x is blocked until f produces the first value c, which causes x to be replaced by c; subsequent values published by f are discarded. The operational semantics is formally defined in Table 5.11. Labels, λ, λ', range over publications, $!c$, and synchronizations, τ. For simplicity, we assume that a site M receives anything, then publishes a predefined value $F_M(c)$ and returns. We write $f \overset{!c}{\Rightarrow}$ if $f \overset{\tau}{\hookrightarrow}^* \overset{!c}{\hookrightarrow}$, that is if f publishes the value c possibly after some internal reductions.

$$
\begin{array}{rclcrcl}
[\![let(x)]\!]_s & = & x(y).\overline{s}\langle y\rangle & & [\![let(c)]\!]_s & = & \overline{s}\langle c\rangle \\
[\![E(x)]\!]_s & = & x(y).\overline{E}\langle y, s\rangle & & [\![E(c)]\!]_s & = & \overline{E}\langle c, s\rangle \\
[\![M(x)]\!]_s & = & x(y).[\![M(y)]\!]_s & & [\![M(c)]\!]_s & = & (\nu r)(\overline{M}\langle c, r\rangle \mid r(y).\overline{s}\langle y\rangle) \\
[\![f \mid g]\!]_s & = & [\![f]\!]_s \mid [\![g]\!]_s & & [\![f > x > g]\!]_s & = & (\nu t)\big([\![f]\!]_t \mid !t(y).(\nu x)(!\overline{x}\langle y\rangle \mid [\![g]\!]_s)\big) \\
\multicolumn{7}{c}{[\![g \ \mathit{where}\ x :\in f]\!]_s \ = \ (\nu r)\big([\![f]\!]_r \mid (\nu x)(r(y).!\overline{x}\langle y\rangle \mid [\![g]\!]_s)\big)}
\end{array}
$$

Table 5.12: Encoding of the ORC language.

5.5.2 Encoding

ORC terms are translated into π-calculus by the function $[\![\cdot]\!]_s$ defined in Table 5.12; s is used here as a result channel. Encoding of a declaration $E(x) \triangleq f$ is given by $!E(x, s).[\![f]\!]_s$. The encoding of the site M is $!M(x, s).\overline{s}\langle F_M(c)\rangle$. The encodings of $let(p)$, $E(p)$ and $M(p)$ for $p = c$ correspond to outputting c on the result channel s and invoking expression E and site M with parameters c and s, respectively. When $p = x$, it is first necessary to retrieve the content of variable x (by reading on it) before proceeding by either outputting, calling E or calling M. The encoding of the parallel composition of two terms, corresponds to the parallel composition of both encodings. The remaining two cases are more interesting. In $[\![f > x > g]\!]_s$ the execution of $[\![f]\!]_t$ is started and each published value is sent on t. For each of these values a new copy of $[\![g]\!]_s$ is started with a new "local variable" x containing such a value. Similar comments for $[\![g \ \mathit{where}\ x :\in f]\!]_s$, but in this case the executions of f and g are started in parallel, and only the first value published by f is considered (thanks to the non-replicated input on r). Note that after the first publication f's execution is not stopped, but it does not interfere with the execution of g because the name r is no longer available.

The encoded terms are well typed under the typing assumptions in Table 5.13. Levels are left unspecified, but suitable values for them can be easily inferred by inspection.

The following result can be used for reasoning about responsiveness of ORC expressions; the proof is reported in Section A.5. In what follows, given an ORC term f, D_f stands for the parallel composition of the encodings of all declarations and sites involved in the definition of f, and $\widetilde{d} = \mathrm{fn}(D_f)$.

Proposition 5.6. *Let f be a closed* ORC *term and suppose D_f is well typed. Under the typing assumptions of Table 5.13, $[\![f]\!]_s$ is well-typed and $F \triangleq (\nu\widetilde{d})([\![f]\!]_s \mid D_f \mid !s(x).\mathbf{0})$, with s and $\widetilde{d}$ +-responsive, is strongly balanced. Moreover, $f \stackrel{!c}{\Longrightarrow}$ if and only if $F \xrightarrow{\tau\langle s,c\rangle}$.*

Name	c, y	x	s	r	t	E	M
Type	I	$\mathsf{I}^{[\rho^+,k_x]}$	$\mathsf{I}^{[\rho^+,h]}$	$\mathsf{I}^{[\rho^+,h']}$	$\mathsf{I}^{[\rho^+,h']}$	$(\mathsf{I}, \mathsf{I}^{[\rho^+,h]})^{[\rho^+,k_E]}$	$(\mathsf{I}, \mathsf{I}^{[\rho^+,h']})^{[\rho^+,k_M]}$
I-Cap.		m	s or p	s	p	p	p
O-Cap.	$-$	p	m	m	m	m	m

with: $k_x > h, k_E, k_M$, and $k_E > h$, and $k_M > h, h'$ and $h' > h, k_x$

Table 5.13: Typing assumptions.

Example 5.5.1. We show how to encode an ORC orchestration pattern into a well-typed π-calculus process. Consider two sites, *CNN* and *BBC*, and suppose that, when invoked, they reply by publishing a news-page. Consider also a site $Mail(m, a)$, which receives a message m and an e-mail address a, and notifies (publishes an uninteresting value) after sending m to a. The ORC function below emails n times the first news-page received from *CNN* or *BBC* to address a, and publishes the current value of n after every sending and upon termination:

$$\begin{aligned} MailNews(n,a) \stackrel{\triangle}{=}\ & if\ n = 0 \quad then \quad let(n) \\ & \qquad\quad else \quad (Mail(t,a) \gg let(n))\ where\ t :\in (CNN|BBC) \\ & \qquad\qquad\qquad\quad | \ MailNews(n-1,a)\ . \end{aligned}$$

Consider the extension of the calculus with natural values, Nat, polyadic communication and an $if - then - else$ construct. Suppose the encodings of sites *CNN*, *BBC* and *Mail* are, respectively, $!CNN(x).\overline{x}\langle N\rangle$, $!BBC(x).\overline{x}\langle N'\rangle$ and $!Mail(x,a,r).(\overline{a}\langle x\rangle \,|\, \overline{r})$, where N and N' represent pieces of news. The function *MailNews* can be encoded as follows:

$$\begin{aligned} MN \stackrel{\triangle}{=}\ & !Mn(n,a,s).\, if \quad n = 0\ then \quad \overline{s}\langle n\rangle \\ & \quad else \ \Big((\nu r)\Big(\overline{CNN}\langle r\rangle \mid \overline{BBC}\langle r\rangle \mid (\nu t)\big(r(y).!\overline{t}\langle y\rangle \mid (\nu r')(t(x).\overline{Mail}\langle x,a,r'\rangle \\ & \qquad\qquad | \ !r'(x)\ .\overline{s}\langle n\rangle)\big)\Big) \mid \overline{MN}\langle n-1,a,s\rangle \Big) \end{aligned}$$

where the received channel s is used for publishing values. Consider an ordering relation "$<$" between (possibly open) integer expressions, e, and variables, x, as follows: $e < x$ if for each evaluation ρ under which e is defined, $e\rho < \rho(x)$. E.g., $x - 1 < x$. We define the "$\lhd$" relation over actions defined as follows: $\overline{c}\langle d\rangle \lhd a(d')$ if either $\mathrm{lev}(c) < \mathrm{lev}(a)$ or $\mathrm{lev}(c) = \mathrm{lev}(a)$ and $d = e < x = d'$, where d, d' denote either names or (open)

expressions. The type system $\vdash_2$ can be easily extended by considering "$\lhd$", in place of "$<$", in rules (T_+-Inp), (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$), and the obvious extensions of Theorem 5.4 and 5.5 can be easily proved. MN is well-typed supposing s, r, r' and t +-responsive, $\mathrm{lev}(Mn) > \mathrm{lev}(CNN)$, $\mathrm{lev}(Mn) > \mathrm{lev}(BBC)$ and $\mathrm{lev}(CNN), \mathrm{lev}(BBC) > \mathrm{lev}(r) > \mathrm{lev}(t) > \mathrm{lev}(Mail) > \mathrm{lev}(r') > \mathrm{lev}(s)$.

Example 5.5.2. We show now that some ORC terms are not encodable into well-typed processes. Consider the term $f = Inc(0)$, where the expression Inc is recursively defined as $Inc(x) \triangleq Succ(x) > y > Inc(y)$ and $Succ$ is the successor function $Succ(x) \triangleq x + 1$. The term F below is not well-typed.

$$
\begin{aligned}
F &\triangleq (\nu\, Succ, Inc)(\llbracket f \rrbracket_s \mid !s(x).\mathbf{0} \mid D_f) \\
\llbracket f \rrbracket_s &\triangleq \overline{Inc}\langle 0, s\rangle \\
D_f &\triangleq\ !Succ(y,s).\overline{s}\langle y+1\rangle \\
&\qquad \mid !Inc(x,r).(\nu\, s)\big(\overline{Succ}\langle x, s\rangle \mid !s(y).(\nu\, w)(!\overline{w}\langle y\rangle \mid w(z).\overline{Inc}\langle z, r\rangle)\big)
\end{aligned}
$$

In fact, $!Inc(x,r).(\nu\, s)\big(\overline{Succ}\langle x, s\rangle \mid !s(y).(\nu\, w)(!\overline{w}\langle y\rangle \mid w(z).\overline{Inc}\langle z, r\rangle)\big)$ is not well-typed ($\overline{Inc}\langle z, r\rangle \not\lhd Inc(x,r)$) and the premise of Proposition 5.6 is not satisfied.

5.6 Conclusions

We have presented two type systems each of which is used for statically enforcing responsive usage of names in π-calculus processes. The first system combines techniques for linearity, receptiveness and deadlock and livelock-freeness. The resulting system allows one to guarantee responsiveness and to give an upper bound on the number of reductions preceding a responsive one. The usual encoding of primitive recursive functions into π-calculus processes (see e.g., [64]) is well typed in this system [7]. In the second system, receptiveness and linearity are relaxed at the price of stronger requirements on levels and balancing. We lose some expressive power in terms of encodable functions. Only tail-recursive functions are encodable into well-typed processes, but we are able to type interesting processes, such as translation of ORC terms. This means that, by supposing all sites always respond, we can use the type system $\vdash_2$ for checking if choreographies defined by using the ORC language describe the behavior of responsive services. Both proposals are syntax driven, so that type checking should be straightforward and efficient to implement.

In Section 5.3.5, we have sketched how to deal with subtyping in the first system, but we do not have pursued this direction because it would require heavy annotations on contexts. On the other side, the definition of a suitable subtyping relation

for the second system deserve further investigation, mainly due to the presence of +-responsive names and capability annotations. Similarly, the definition of an inference system requires additional care due to the presence of levels and annotations.

Chapter

SIX

AtCCS: A concurrent calculus with atomic transactions

In this chapter, we depart from the main topic of the thesis to look into another kind of functional properties of WS. *While in the previous chapters we mainly looked at static properties by using "static tools", like type systems, here we concentrate on error recovery aspects and use "dynamic tools", like behavioral equivalences. In particular, we study the* Software Transactional Memory *(*STM*) model, an original approach for controlling accesses to shared resources in concurrent applications, from a process algebra perspective. We define* AtCCS*: an extension of asynchronous* CCS *with atomic blocks of actions. We show that the addition of atomic transactions results in a very expressive calculus, enough to easily encode other concurrent primitives such as (preemptive versions of) guarded choice and multiset-synchronization à la join-calculus. The correctness of the encodings is proved using a suitable notion of bisimulation equivalence. The equivalence is then applied to prove interesting "laws of transactions" and to obtain a simple normal form for atomic blocks. Finally, we propose a may-testing semantics for AtCCS and prove that it is not appropriate for reasoning on atomic processes.*

6.1 Introduction

When studying and designing distributed and concurrent systems, it is necessary to tackle the problem of *failures*. Typical problems of these systems arise from the presence of a shared memory. A memory access control is needed for avoiding that unwarranted accesses would give rise to inconsistent states and compromise future computations. This kinds of problems have been already studied for a long time – e.g., in the databases setting – and *transactions* have been introduced for coping with

them. In the specific case of SOA and WS the problem is not only to deal with a shared memory, but also with "shared services". Shared services in the sense that the required service may involve more than one service and more than one WS and the failure of at least one of them would cause a global failure. As an example, consider a trip booking service that, when invoked, tries to book both flight and hotel by invoking two other suitable services. If one of the two services fails, it is necessary to cancel the reservation made by the other (undo its actions): this for guaranteeing a consistent global state.

The WS-Transaction specification [91] defines mechanisms for transactional interoperability between WS and provide a means to compose transactional qualities of service into WS applications. It identifies two types of transactions: WS-*atomic transactions* and WS-*business activities*. The first class identifies short-lived and distributed activities characterized by the all-or-nothing semantics, which is guaranteed by using well-known commit protocols, like the two-phase commit. In this case the ACID properties of transactions, in their databases' sense, are guaranteed. In the case of (long-lived) business activities each "transactive" block of actions is associated with a program, called *compensation*, that has to be run if a failure is detected. Its goal is undoing the visible actions that may have been performed – in pretty much the same way that exception handling in programming languages – and restoring a consistent state. In the case of compensating transactions, atomicity, isolation and durability are obviously violated. Both types of transaction specifications are useful for coordinating the transactional behavior of (distributed) services and the (distributed) recovery of errors. But in both cases nothing is said about *local* computation. In general, the usage of local transactional mechanisms, such as *locks*, is assumed for ensuring correct local execution of transactions.

In this chapter we focus on the local aspect of transactions and investigate the *optimistic* approach found in *Software Transactional Memory* (STM) [84]. The STM model is an original approach for controlling concurrent accesses to resources without using explicit lock-based synchronization mechanisms. Similarly to database transactions, the STM model provides a way to group sequences of read and write actions inside atomic blocks whose whole effect should occur atomically. This model has several advantages. Most notably, it dispenses the programmer from the need to explicitly manipulate locks, a task widely recognized as difficult and error-prone. Moreover, atomic transactions provide a clean conceptual basis for concurrency control, which should ease the verification of concurrent programs. Finally, the model is effective:

there exist several STM implementations for designing software for multiprocessor systems; these applications exhibit good performances in practice compared to equivalent, hand-crafted, code using locks.

We study the STM model from a process algebra perspective and define an extension of asynchronous CCS with atomic blocks of actions. We call this calculus AtCCS. The choice of a dialect of CCS is motivated by an attention to economy: to focus on STM primitives, we study a calculus as simple as possible and dispense with orthogonal issues such as values, mobility of names or processes, etc. We believe that our work could be easily transferred to a richer setting. Our goal is not only to set a formal ground for reasoning on STM implementations but also to understand how this model fits with other concurrency control mechanisms. We also view this calculus as a testbed for extending process calculi with atomic transactions.

The idea of providing hardware support for software transactions originated from works by Herlihy and Moss [84] and was later extended by Shavit and Touitou [127] to software-only transactional memory. Transactions are used to protect the execution of an atomic block. Intuitively, each thread that enters a transaction takes a snapshot of the shared memory (the global state). The evaluation is optimistic and all actions are performed on a copy of the memory (the local state). When the transaction ends, the snapshot is compared with the current state of the memory. There are two possible outcomes: if the check indicates that concurrent writes have occurred, the transaction aborts and is rescheduled; otherwise, the transaction is committed and its effects are propagated instantaneously. Very recently, Harris *et al.* [81] have proposed a (combinator style) language of transactions that enables arbitrary atomic operations to be composed into larger atomic expressions. We base the syntax of AtCCS on the operators defined in [81].

We show that the addition of atomic transactions results in a very expressive calculus, enough to easily encode other concurrent primitives such as (preemptive versions of) guarded choice and multiset-synchronization à la join-calculus (Section 6.2). The correctness of our encodings is proved using a suitable notion of asynchronous bisimulation equivalence that allows compositional reasoning on transactions and is shown to be a congruence (Section 6.3). The equivalence is applied to prove interesting "laws of transactions" and to obtain a simple normal form for atomic blocks. We also use our transactions to give straightforward solutions to two celebrated examples of concurrent problems: the leader election and dining philosophers problems. Finally, in Section 6.4 we show that a may-testing equivalence is not strong enough when ones want to verify interesting properties of processes – such as atomicity in our case.

The proofs of the main results of this chapter are reported in Appendix B.

Related works Transactions, failures and atomicity have been studied for a long time and there are a lot of works treating such matters in different ways.

We can list several works that combine ACID transactions with process calculi. Gorrieri et al [79] have modeled concurrent systems with atomic behaviors using an extension of CCS. They use a two-level transition system (a high and a low level) where high actions are decomposed into atomic sequences of low actions. To enforce isolation, atomic sequences must go into a special invisible state during all their execution. Contrary to our model, this work does not follow an optimistic approach: transactions are executed sequentially, without interleaving with other actions, as though in a critical section. Another related calculus is RCCS, a reversible version of CCS [61, 62] based on an earlier notion of process calculus with backtracking [20]. In RCCS, each process has access to a log of its synchronization's history and may always wind back to a previous state. This calculus guarantees the ACD properties of transactions (isolation is meaningless since RCCS do not use a shared memory model). A similar approach is followed in [57], where two extensions of the π-calculus are proposed: the pik-calculus and the pike-calculus. Both calculi incorporate various abstractions for fault tolerance, from which several forms of distributed transactions and distributed computation can be built. Each transactional block is provided with a log that is used when rollback is needed. Moreover, each transaction can access – in read-only fashion – to any other logs. A relation of causality among logs, hence among transactions, is used for dealing with commits and aborts. In some works, serializability is used as a criterion to evaluate the correctness of transaction semantics. Busi and Zavattaro [41] introduce the semantics of JavaSpaces by following a pessimistic approach. Locks are acquired when entries are accessed, preventing other transactions from using these values until the owning transaction commits. This pessimistic approach has two notable disadvantages: deadlock and scalability. In [93], a transactional semantics – based on an optimistic concurrency model – for a transactional variant of Linda [40] is defined. Nested and multithreaded transactions are allowed and a log-based approach is followed for guaranteeing serializabilty of sequences of actions. A similar approach is followed in [131], where a framework for specifying the semantics of nested and multithreaded transactions in an object calculus is given. The framework is parametrized by the definition of a transactional mechanism and allows the study of multiple models, such as the usual lock-based approach. In this work, STM is close to a model called *versioning semantics*. Like in our approach, this model is based on the use of logs and is characterized by an optimistic approach where log consistency is checked at

commit time. STM are used in practice in [67], where Donnelly and Fluet introduce a concurrency abstraction combining first-class synchronization message-passing events with all-or-nothing transactions. They introduce the notion of Transactional Events (TE) and give a formal semantics for TE Haskell: a language inspired from Concurrent ML and Concurrent Haskell, implemented by using STM Haskell. TE have the compositional structure of a monad-with-plus and synchronization among transactions is allowed. The dynamic semantics guarantees the all-or-nothing execution of TE. As stated by the authors, this work raises interesting questions about the relationship between TE and process calculi and about a right notion of behavioral equivalence – that would be used also for proving the monad-with-plus laws. Fewer works consider behavioral equivalences for transactions. A foundational work is [25], that gives a theory of transactions specifying atomicity, isolation and durability in the form of an equivalence relation on processes, but it provides no formal proof system.

Linked to the upsurge of works on WS and on long running Web transactions, a larger body of works is concerned with formalizing *compensating transactions*. We give a brief survey of works that formalize compensable processes using process calculi. These works can be grouped into two classes: (1) *interaction based compensation* [36, 26, 37, 101], which are extensions of process calculi (like π or join-calculus) for describing transactional choreographies where composition take place dynamically and where each service describes its possible interactions and compensations; (2) *compensable flow composition* [38, 43, 44], where ad hoc process algebras are designed from scratch to describe the possible flow of control among services. These calculi are oriented towards the orchestration of services and service failures. This second approach is also followed in [18, 22] where two frameworks for composing transactional services are presented.

6.2 The calculus

In this section we present syntax and operational semantics of the calculus, which is essentially asynchronous CCS [110], without choice and relabeling operators, equipped with atomic blocks (transactions) and constructs for composing transactional sequences of actions.

6.2.1 Syntax

We let $\mathcal{N}$, ranged over a, b, ..., be an infinite set of *names*. As in CCS, names model communication channels used in process synchronization, but they also occur

Action	α, β	::=	$rd(a)$	*Read a from memory*
			$\mid wt(a)$	*Write a into the memory*
Atomic expression	M, N	::=	end	*End*
			$\mid retry$	*Retry*
			$\mid \alpha.M$	*Prefix*
			$\mid M \ orElse \ M$	*Alternative*
Ongoing atomic block	A, B	::=	$(M)_{\sigma;\delta}$	M
			$\mid A \ orElse \ A$	*Ongoing alternative*
Process	P, R	::=	$\mathbf{0}$	*Nil*
			$\mid \overline{a}$	*Output*
			$\mid a.P$	*Input*
			$\mid !a.P$	*Replicated input*
			$\mid atom(M)$	*Atomic block*
			$\mid P \mid P$	*Parallel composition*
			$\mid P \setminus^n a, \ n \geqslant 0$	*Hiding*
			$\mid \{\!\mid A \mid\!\}_M$	*Ongoing atomic block*

Table 6.1: Syntax

as objects of read and write actions in atomic transactions.

Definition 6.1. *The set $\mathcal{P}$ of processes, ranged over P, R, ..., $\mathcal{M}$ of atomic expressions, ranged over M, N, ..., and $\mathcal{A}$ of ongoing atomic expressions, ranged over A, B, C, ... are defined by the grammar in Table 6.1.*

Atomic expressions are used to define sequences of actions whose effect should happen atomically. Actions $rd(a)$ and $wt(a)$ represent attempts to input and output to the channel a. Instead of using snapshots of the state for managing transaction, we use a log-based approach. During the evaluation of an atomic block, actions are recorded in a private log δ (a sequence $\alpha_1, \ldots, \alpha_n$) and have no effects outside the scope of the transaction until it is committed. The action *retry* aborts an atomic expression unconditionally and starts its execution afresh. The termination action *end* signals that an expression is finished and should be committed. If the transaction can be committed, all actions in the log are performed at the same time and the transaction is closed, otherwise the transaction aborts. Finally, transactions can be composed using

the operator *orElse*, which implements preemptive alternatives between expressions. In $M \ orElse \ N$, the expression N is executed if M aborts and has the behavior of M otherwise. This allows processes to wait for many things at once.

Ongoing atomic blocks are essentially atomic expressions enriched with an evaluation state σ and a log δ of the currently recorded actions. A state σ is a multiset of names that represents the output actions visible to the transaction when it was initiated. This notion of state bears some resemblance with tuples space in coordination calculi, such as Linda [40]; but here we consider only read and write primitives for modifying the state. When a transaction ends, the state σ recorded in the block $(M)_{\sigma;\delta}$ – the state at the initiation of the transaction – can be compared with the current state – the state when the transaction ends – to check if other processes have concurrently made changes to the global state, in which case the transaction should be aborted.

Processes model concurrent systems of communicating agents. We have the usual operators of CCS: the empty process, $\mathbf{0}$, the parallel composition $P \mid R$, and the input prefix $a.P$. There are some differences though. The calculus is asynchronous, meaning that a process cannot block on output actions. Also, we use replicated input $!a.P$ instead of recursion – this does not change the expressiveness of the calculus – and we lack the choice and relabeling operators of CCS. The hiding operator $P \setminus^n a$ bounds the scope of name a to P. The integer n stands for the number of outputs on a that can be accessed by P. Finally, the main addition is the presence of the operator $atom(M)$, which models a transaction that safeguards the expression M. The process $\{\!|A|\!\}_M$ represents the ongoing evaluation of an atomic block M: the subscript is used to keep the initial code of the transaction, in case it is aborted and executed afresh, while A holds the remaining actions that should be performed.

Notations. In what follows, we consider processes up to alpha-renaming of bound names and usually omit trailing *end* in atomic expressions. We write $\sigma \uplus \{a\}$ for the multiset σ enriched with the name a and $\sigma \setminus \sigma'$ for the multiset obtained from σ by removing elements found in σ', that is a multiset σ'' such that $\sigma = \sigma' \uplus \sigma''$. The symbol $\emptyset$ stands for the empty multiset while $\{a^n\}$ is the multiset composed of exactly n copies of a, where $\{a^0\} = \emptyset$.

We denote the empty log as ϵ. Given a log δ, we use the notation $\text{WT}(\delta)$ for the multiset of names which appear as objects of a write action in δ. Similarly, we use the notation $\text{RD}(\delta)$ for the multiset of names that are objects of read actions. Formal definitions of the functions WT and RD are given in Table 6.2.

$$\begin{array}{cc} \textsc{wt}(\epsilon) = \textsc{rd}(\epsilon) = \epsilon & \\ \textsc{wt}(wt(a).\delta) = \textsc{wt}(\delta) \uplus \{a\} & \textsc{wt}(rd(a).\delta) = \textsc{wt}(\delta) \\ \textsc{rd}(rd(a).\delta) = \textsc{rd}(\delta) \uplus \{a\} & \textsc{rd}(wt(a).\delta) = \textsc{rd}(\delta)\ . \end{array}$$

Table 6.2: WT and RD

Example 6.2.1 (composing synchronization). Before we describe the meaning of processes, we try to convey the semantics of AtCCS (and the usefulness of the atomic block operator) using a simple example. Consider a concurrent system with two memory cells, M_1 and M_2, used to store integers. We consider here a straightforward extension of the calculus with "value-passing." In this setting, we can model a cell with value v by an output $\overline{m_i}!v$ and model an update by a process of the form $m_i?x.(\overline{m_i}!v' \mid \dots)$. With this encoding, the channel name m_i acts as a lock protecting the shared resource M_i.

Assume now that the values of the cells should be synchronized to preserve a global invariant on the system. For instance, we model a flying aircraft, each cell store the pitch of an aileron and we need to ensure that the aileron stay aligned, that is that the values of the cells are equal. A process testing the validity of the invariant is for example P_1 below (we suppose that a message on the reserved channel *err* triggers an alarm). There are multiple design choices for resetting the value of both cells to 0, e.g. P_2 and P_3.

$$P_1 \stackrel{\triangle}{=} m_1?x.m_2?y.\text{if } x \mathrel{!}= y \text{ then } \overline{err}!$$

$$P_2 \stackrel{\triangle}{=} m_2?x.m_1?y.\big(\overline{m_1}!0 \mid \overline{m_2}!0\big) \qquad P_3 \stackrel{\triangle}{=} m_1?x.\big(\overline{m_1}!0 \mid m_2?y.\overline{m_2}!0\big)$$

Each choice exemplifies a problem with lock-based programming. The composition of P_1 with P_2 leads to a race condition where P_1 acquire the lock on M_1, P_2 on M_2 and each process gets stuck. The composition of P_1 and P_3 may break the invariant because the value of M_1 is updated too quickly. A solution in the first case is to strengthen the invariant and enforce an order for acquiring locks, but this solution is not viable in general and opens the door to *priority inversion* problems. Another solution is to use an additional (master) lock to protect both cells, but this approach obfuscate the code and significantly decreases the concurrency of the system.

Overall, this simple example shows that synchronization constraints do not compose well when using locks. This situation is consistently observed (and bears a resemblance to the inheritance anomaly problem found in concurrent object-oriented languages). The approach advocated here is to use atomic transactions. In our exam-

$$(\textsc{out})\ \overline{a}\,;\sigma \to \mathbf{0}\,;\sigma \uplus \{a\} \qquad (\textsc{rep})\ !a.P\,;\sigma \uplus \{a\} \to P \mid !a.P\,;\sigma$$

$$(\textsc{in})\ a.P\,;\sigma \uplus \{a\} \to P\,;\sigma \qquad (\textsc{com})\ \frac{P\,;\sigma \to P'\,;\sigma \uplus \{a\} \quad R\,;\sigma \uplus \{a\} \to R'\,;\sigma}{P \mid R\,;\sigma \to P' \mid R'\,;\sigma}$$

$$(\textsc{parL})\ \frac{P\,;\sigma \to P'\,;\sigma'}{P \mid R\,;\sigma \to P' \mid R\,;\sigma'} \qquad (\textsc{hid})\ \frac{P\,;\sigma \uplus \{a^n\} \to P'\,;\sigma' \uplus \{a^m\} \qquad a \notin \sigma, \sigma'}{P \setminus^n a\,;\sigma \to P' \setminus^m a\,;\sigma'}$$

$$(\textsc{parR})\ \frac{R\,;\sigma \to R'\,;\sigma'}{P \mid R\,;\sigma \to P \mid R'\,;\sigma'} \qquad (\textsc{atSt})\ atom(M)\,;\sigma \to \{\!|(M)_{\sigma;\epsilon}|\!\}_M\,;\sigma$$

$$(\textsc{atPass})\ \frac{A \to A'}{\{\!|A|\!\}_M\,;\sigma \to \{\!|A'|\!\}_M\,;\sigma} \qquad (\textsc{atRe})\ \{\!|(retry)_{\sigma';\delta}|\!\}_M\,;\sigma \to atom(M)\,;\sigma$$

$$(\textsc{atFail})\ \frac{\textsc{rd}(\delta) \not\subseteq \sigma}{\{\!|(end)_{\sigma';\delta}|\!\}_M\,;\sigma \to atom(M)\,;\sigma}$$

$$(\textsc{atOk})\ \frac{\textsc{rd}(\delta) \subseteq \sigma \quad \sigma = \sigma'' \uplus \textsc{rd}(\delta) \quad \textsc{wt}(\delta) = \{a_1, \ldots, a_n\}}{\{\!|(end)_{\sigma';\delta}|\!\}_M\,;\sigma \to \overline{a}_1 \mid \ldots \mid \overline{a}_n\,;\sigma''}$$

Table 6.3: Operational semantics processes.

ple, the problem is solved by simply wrapping the two operations in a transaction, like in the process $atom\big(rd(m_2?y).\,wt(m_2!0).\,rd(m_1?x).\,wt(m_1!0)\big)$, which ensures that all cell updates are effected atomically.

6.2.2 Reduction semantics

The semantics of AtCCS is stratified in two levels: there is one reduction relation for processes and a second for atomic expressions. With a slight abuse of notation, we use the same symbol ($\to$) for both relations.

Semantics of processes. Table 6.3 gives the semantics of processes. A reduction is of the form $P\,;\sigma \to P'\,;\sigma'$ where σ is the state of P. The state σ records the names of all output actions visible to P when reduction happens. It grows when an output is reduced, (OUT), and shrinks in the case of inputs, (IN) and (REP). A parallel composition evolves if one of the component evolves or if both can synchronize, rules (PARL), (PARR) and (COM). In a hiding $P\backslash^n a$, the annotation n is an integer denoting the number of outputs on a that are visible to P. Intuitively, in a "configuration" $P \setminus^n a\,;\sigma$, the outputs visible to P are those in $\sigma \uplus \{a^n\}$. This extra annotation is necessary because the scope of a is restricted to P, hence it is not possible to have outputs on a in the global state. Rule (HID) allows reductions, also involving name a, to happen inside a hiding. For instance, we have $(P \mid \overline{a}) \setminus^n a\,;\sigma \to P \setminus^{n+1} a\,;\sigma$.

$$\text{(ARDOK)}\quad \frac{\text{RD}(\delta) \uplus \{a\} \subseteq \sigma}{(rd(a).M)_{\sigma;\delta} \to (M)_{\sigma;\delta.\, rd(a)}} \qquad \text{(ARDF)}\quad \frac{\text{RD}(\delta) \uplus \{a\} \not\subseteq \sigma}{(rd(a).M)_{\sigma;\delta} \to (retry)_{\sigma;\delta}}$$

$$\text{(AWR)}\quad (wt(a).M)_{\sigma;\delta} \to (M)_{\sigma;\delta.\, wt(a)}$$

$$\text{(AOI)}\quad (M_1\ orElse\ M_2)_{\sigma;\delta} \to (M_1)_{\sigma;\delta}\ orElse\ (M_2)_{\sigma;\delta}$$

$$\text{(AOF)}\quad (retry)_{\sigma;\delta}\ orElse\ B \to B \qquad \text{(AOE)}\quad (end)_{\sigma;\delta}\ orElse\ B \to (end)_{\sigma;\delta}$$

$$\text{(AOL)}\quad \frac{A \to A'}{A\ orElse\ B \to A'\ orElse\ B} \qquad \text{(AOR)}\quad \frac{B \to B'}{A\ orElse\ B \to A\ orElse\ B'}$$

Table 6.4: Operational semantics atomic expressions.

The remaining reduction rules govern the evolution of atomic transactions. Like in the case of (COM), all those rules, but (ATOK), leave the global state unchanged. Rule (ATST) deals with the initiation of an atomic block $atom(M)$: an ongoing block $\{\!|(M)_{\sigma;\epsilon}|\!\}_M$ is created which holds the current evaluation state σ and an empty log ϵ. An atomic block $\{\!|A|\!\}_M$ reduces when its expression A reduces – according to the semantics in Table 6.4, rule (ATPASS). Rules (ATRE), (ATFAIL) and (ATOK) deal with the completion of a transaction. After a finite number of reductions, the evaluation of an ongoing expression will necessarily result in a fail state, $(retry)_{\sigma;\delta}$, or a success, $(end)_{\sigma;\delta}$. In the first case, rule (ATRE), the transaction is aborted and started again from scratch. In the second case, we need to check if the log is *consistent* with the current evaluation state. We consider a log as consistent if the read actions of δ can be performed on the current state. If the check fails, rule (ATFAIL), the transaction aborts. Otherwise, rule (ATOK), we commit the transaction: the names in $\text{RD}(\delta)$ are taken from the current state and a bunch of outputs on the names in $\text{WT}(\delta)$ are generated.

Remark 6.1. The synchronization rule (COM) may seem redundant here, because it can be simulated by applying (OUT) followed by either (REP) or (IN). The presence of (COM) is fundamental in the definition of the labeled semantics for AtCCS (see Proposition 6.1) and we have preferred to introduce (COM) here for ease of presentation.

Semantics of atomic expressions. Table 6.4 gives the semantics of ongoing atomic expressions. We recall that, in an expression $(rd(a).M)_{\sigma;\delta}$, the subscript σ is the *initial state*, that is a copy of the state at the time the block has been created and δ is the log of actions performed since the initiation of the transaction.

Rule (ARDOK) states that a read action $rd(a)$ is recorded in the log δ if all read

actions in $\delta.\, rd(a)$ can be performed in the initial state. If it is not the case, the ongoing expression fails, rule (ARDF). This test may be interpreted as a kind of optimization: if a transaction cannot commit in the initial state then, should it commit at the end of the atomic block, it would mean that the global state has been concurrently modified during the execution of the transaction. Note that we consider the initial state σ and not $\sigma \uplus \mathrm{WT}(\delta)$, which means that, in an atomic block, write actions are not directly visible and cannot be consumed by a read action. This is coherent with the fact that outputs on $\mathrm{WT}(\delta)$ only take place after commit of the block. Rule (AWR) states that a write action always succeeds and is recorded in the current log.

The remaining rules govern the semantics of the *retry*, *end* and *orElse* constructs. These constructs are borrowed from the STM combinators used in the implementation of an STM system in Concurrent Haskell [81]. We define these operators with an equivalent semantics, with the difference that, in our case, a state is not a snapshot of the shared memory but a multiset of visible outputs. A composition $M \;\; orElse \;\; N$ corresponds to the interleaving of the behaviors of M and N, which are independently evaluated with respect to the same evaluation state (but have distinct logs), (AOL) and (AOR). The *orElse* operator is preemptive: the ongoing block $M \;\; orElse \;\; N$ ends, (AOE), if either M ends or M aborts and N ends, (AOF).

Remark 6.2. The semantics defined here is akin to an optimistic concurrency protocol in which the validity of read and write performed within an atomic block is determined by verifying if the log is consistent with the global state at commit time. We do not care if such state is different from the evaluation state of the atomic block. Evaluation states are introduced only with the aim of increasing "performances": an atomic block may be immediately retried when it tries to inputs a non-available name instead of waiting the commit time for re-starting its evaluation.

Example 6.2.2 (leader election). Our first example is a simple (non-blocking) solution to the well-known *leader election* problem. Consider a system composed by n processes and a token, named t, that is modeled by an output $\bar{t}$. A process becomes a leader by getting (making an input on) t. As usual, all participants run the same process (except for the value i of their identity). We suppose that there is only one copy of the token in the system and that leadership of process i is communicated to the other processes by outputting on a reserved name win_i. A participant that is not a leader outputs on $lose_i$. The protocol followed by the participants is defined by the following process:

$$L_i \triangleq \big(atom\big(rd(t).\, wt(k).\, end \;\; orElse \;\; wt(k').\, end\big) \mid k.\overline{win_i} \mid k'.\overline{lose_i}\big) \setminus^0 k \setminus^0 k' \;.$$

In this encoding, the atomic block is used to protect the concurrent accesses to t. If the process L_i commits its transaction and grabs the token, it immediately release an output on its private channel k. The transactions of the other participants may either fail or commit while releasing an output on their private channel k'. Then, the elected process L_i may proceed with a synchronization on k that triggers the output $\overline{win}_i$. The semantics of $atom(\cdot)$ ensures that only one transaction can acquire the lock and commit the atomic block, then no other process has acquired the token and we are guaranteed that there could be at most one leader. For simplicity, we propose a *one*-round solution. An extension to a multiple-round system is straightforward: it is enough to release the token after outputting win_i and consuming $lose_j$ for all $j \neq i$.

This expressivity result is mixed blessing. Indeed, it means that any implementation of the atomic operator should be able to solve the leader election problem, which is known to be very expensive in the case of loosely-coupled systems or in presence of failures (see e.g. [116] for a discussion on the expressivity of process calculi and electoral systems). On the other hand, atomic transactions are optimistic and are compatible with the use of probabilistic approaches. Therefore it is still reasonable to expect a practical implementation of AtCCS.

Example 6.2.3 (guarded choice). We consider an operator for choice, $\mu_1.P_1 + \cdots + \mu_n.P_n$, such that every process is prefixed by an action μ_i that is either an output $\overline{a}_i$ or an input a_i. The semantics of choice is characterized by the following three reduction rules (we assume that R is also a choice):

$$\text{(C-INP)} \quad a.P + R\,;\sigma \uplus \{a\} \to P\,;\sigma \qquad \text{(C-OUT)} \quad \overline{a}.P + R\,;\sigma \to P\,;\sigma \uplus \{a\}$$

$$\text{(C-PASS)} \quad \frac{a \notin \sigma \quad R\,;\sigma \to R'\,;\sigma'}{a.P + R\,;\sigma \to R'\,;\sigma'}$$

A minor difference with the behavior of the choice operator found in CCS is that our semantics gives precedence to the leftmost process (this is reminiscent of the preemptive behavior of *orElse*). Another characteristic is related to the asynchronous nature of the calculus, see rule (C-OUT): since an output action can always interact with the environment, a choice $\overline{a}.P + R$ may react at once and release the process $\overline{a} \mid P$.

Like in the example of the leader election problem, we can encode a choice $\mu_1.P_1 + \cdots + \mu_n.P_n$ using an atomic block that will mediate the interaction with the actions $\mu_1, \ldots, \mu_n$. We start by defining a straightforward encoding of input/output actions into atomic actions: $[\![\overline{a}]\!] = wt(a)$ and $[\![a]\!] = rd(a)$. Then the encoding of choice is the

process

$$[\![\mu_1.P_1 + \cdots + \mu_n.P_n]\!] \stackrel{\triangle}{=} \big(atom\big([\![\mu_1]\!].[\![\overline{k}_1]\!].end \;\; orElse \;\; \cdots \;\; orElse \;\; [\![\mu_n]\!].[\![\overline{k}_n]\!].end\big) \\ \mid k_1.[\![P_1]\!] \mid \cdots \mid k_n.[\![P_n]\!]\big) \setminus^0 k_1 \ldots \setminus^0 k_n$$

The principle of the encoding is essentially the same that in our solution to the leader election problem. Actually, using the encoding for choice, we can rewrite our solution in the following form: $L_i \stackrel{\triangle}{=} t.\overline{win}_i + \overline{lose}_i.\mathbf{0}$. Using the rules in Table 6.3, it is easy to see that our encoding of choice is compatible with rule (C-INP), meaning that:

$$\begin{aligned}[\![a.P + R]\!]\,;\sigma \uplus \{a\} &\rightarrow^* \big(\{\!|(end)_{\sigma\uplus\{a\};rd(a).\,wt(k_1)}|\!\}_M \mid k_1.[\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots\,;\sigma \uplus \{a\} \\ &\rightarrow \big(\overline{k}_1 \mid k_1.[\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots\,;\sigma \\ &\rightarrow^* \big([\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots\,;\sigma\end{aligned}$$

where the processes in parallel with $[\![P]\!]$ are harmless. In the next section, we define a weak bisimulation equivalence $\approx_{\mathsf{a}}$ that can be used to garbage collect harmless processes in the sense that, e.g. $(P \mid k.R) \setminus^0 k \approx_{\mathsf{a}} P$ if P has no occurrences of k. Hence, we could prove that $[\![a.P + R]\!]\,;\sigma \uplus \{a\} \rightarrow^* \approx_{\mathsf{a}} [\![P]\!]\,;\sigma$, which is enough to show that our encoding is correct with respect to rule (C-INP). The same is true for rules (C-OUT) and (C-PASS).

Example 6.2.4 (join pattern). A multi-synchronization $(a_1 \times \cdots \times a_n).P$ may be viewed as an extension of input prefix in which communication requires a synchronization with the n outputs $\overline{a}_1, \ldots, \overline{a}_n$ at once. that is, we have the reduction:

$$\text{(J-INP)} \quad (a_1 \times \cdots \times a_n).P\,;\sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow\; P\,;\sigma$$

This synchronization primitive is fundamental to the definition of the Gamma calculus of Banâtre and Le Métayer and of the Join calculus of Fournet and Gonthier. It is easy to see that the encoding of a multi-synchronization (input) is a simple transaction:

$$[\![\big(a_1 \times \cdots \times a_n\big).P]\!] \;\stackrel{\triangle}{=}\; \big(atom([\![a_1]\!].\cdots.[\![a_n]\!].[\![\overline{k}]\!].end) \mid k.[\![P]\!]\big) \setminus^0 k$$

with k fresh name, and that we have

$$[\![\big(a_1 \times \cdots \times a_n\big).P]\!]\,;\sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow^*\; \big(\mathbf{0} \mid [\![P]\!]\big) \setminus^0 k\,;\sigma$$

where the process $\big(\mathbf{0} \mid [\![P]\!]\big) \setminus^0 k$ is behaviorally equivalent to $[\![P]\!]$, that is:

$$[\![\big(a_1 \times \cdots \times a_n\big).P]\!]\,;\sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow^* \approx_{\mathsf{a}}\; [\![P]\!]\,;\sigma$$

Based on this encoding, we can define two interesting derived operators: a mixed version of multi-synchronization, $(\mu_1 \times \cdots \times \mu_n).P$, that mixes input and output actions; and a replicated version, that is analogous to replicated input.

$$[\![(\mu_1 \times \cdots \times \mu_n).P]\!] \triangleq (atom([\![\mu_1]\!].\cdots.[\![\mu_n]\!].[\![\overline{k}]\!].\,end) \mid k.[\![P]\!]) \setminus^0 k$$
$$[\![!\,(\mu_1 \times \cdots \times \mu_n).P]\!] \triangleq (\overline{r} \mid !r.\,atom([\![\mu_1]\!].\cdots.[\![\mu_n]\!].[\![\overline{r}]\!].[\![\overline{k}]\!].\,end) \mid !k.[\![P]\!]) \setminus^0 r \setminus^0 k$$

By looking at the possible reductions of these (derived) operators, we can define derived reduction rules. Assume δ is the log $[\![\mu_1]\!].\cdots.[\![\mu_n]\!]$, we have a simulation result comparable to the case for multi-synchronization, namely:

$$[\![(\mu_1 \times \cdots \times \mu_n).P]\!]\,;\sigma \uplus \mathrm{RD}(\delta) \;\rightarrow^* \approx_{\mathsf{a}}\; [\![P]\!]\,;\sigma \uplus \mathrm{WT}(\delta)$$
$$[\![!(\mu_1 \times \cdots \times \mu_n).P]\!]\,;\sigma \uplus \mathrm{RD}(\delta) \;\rightarrow^* \approx_{\mathsf{a}}\; [\![!(\mu_1 \times \cdots \times \mu_n).P]\!] \mid [\![P]\!]\,;\sigma \uplus \mathrm{WT}(\delta)$$

To obtain join-definitions, we need to combine a sequence of replicated multi-synchronizations using the choice composition defined precedently and we need hiding to close the scope of the definition. Actually, we can encode even more flexible constructs mixing choice and join-patterns. For the sake of simplicity, we only study examples of such operations. The first example is the (linear) join-pattern $(a \times b).P \wedge (a \times c).R$, that may fire P if the outputs $\{a, b\}$ are in the global state σ and otherwise fire R if $\{a, c\}$ is in σ – actually, real implementations of join-calculus have a preemptive semantics for pattern synchronization. The second example is the derived operator $(a \times b) + (b \times c \times \overline{a}).P$, such that P is fired if outputs on $\{a, b\}$ are available or if outputs on $\{b, c\}$ are available (in which case an output on a is also generated). These examples can be easily interpreted using atomic transactions:

$$[\![(a \times b).P \wedge (a \times c).R]\!] \triangleq \big(atom\big([\![a]\!].[\![b]\!].[\![\overline{k}_1]\!].\,end \;\; orElse \;\; [\![a]\!].[\![c]\!].[\![\overline{k}_2]\!].\,end\big) \mid k_1.P \mid k_2.R\big) \setminus^0 k_1 \setminus^0 k_2$$
$$[\![(a \times b + b \times c \times \overline{a}).P]\!] \triangleq \big(atom\big([\![a]\!].[\![b]\!].[\![\overline{k}]\!].\,end \;\; orElse \;\; [\![b]\!].[\![c]\!].[\![\overline{a}]\!].[\![\overline{k}]\!].\,end\big) \mid k.P\big) \setminus^0 k$$

In the next section we define the notion of bisimulation used for reasoning on the soundness of our encodings. We also define an equivalence relation for atomic expressions that is useful for reasoning on the behavior of atomic blocks.

6.3 Bisimulation semantics

A first phase before obtaining a bisimulation equivalence is to define a Labeled Transition System (LTS) for AtCCS processes.

6.3.1 Labeled semantics

It is easy to derive labels from the reduction semantics given in Table 6.3. For instance, a reduction of the form $P\,;\sigma \to P'\,;\sigma \uplus \{a\}$ is clearly an *output transition* and we could denote it using the transition $P \xrightarrow{\overline{a}} P'$, meaning that the effect of the transition is to add a message on a to the global state σ. In the following, we formalize the notion of label and transition. Let *Out* be the set of output actions of the form $\overline{a}$, with $a \in \mathcal{N}$. Besides outputs, which corresponds to an application of rule (OUT), we also need *block actions*, which are multisets of the form $\{\!|a_1, \ldots, a_n|\!\}$ corresponding to the commit of an atomic block, that is to the deletion of a bunch of names from the global state in rule (ATOK). Let $Ib = \{\{\!|a_1, \ldots, a_n|\!\} | a_i \in \mathcal{N}\}$, ranged over θ, γ, ..., be the set of (atomic) block actions. Block actions include the usual labels found in LTS for CCS and are used for labeling input and synchronizations: an input action a, which intuitively corresponds to rules (IN) and (REP), is a shorthand for the singleton block action $\{\!|a|\!\}$; the silent action τ, which corresponds to rule (COM), is a shorthand for the empty block action $\emptyset$. In the following, we use the symbols $\mu, \mu', \ldots$ to range over labels, $\mu ::= \overline{a} \,\big|\, \theta \,\big|\, \tau \,\big|\, a$.

Definition 6.2 (labeled semantics). *The labeled semantics for AtCCS is the smallest relation $P \xrightarrow{\mu} P'$ satisfying the two following clauses:*

(1) *we have $P \xrightarrow{\overline{a}} P'$ if there is a state σ such that $P\,;\sigma \to P'\,;\sigma \uplus \{a\}$;*

(2) *we have $P \xrightarrow{\theta} P'$ if there is a state σ such that $P\,;\sigma \uplus \theta \to P'\,;\sigma$.*

Note that, in the case of the (derived) action τ, we obtain from clause (2) that $P \xrightarrow{\tau} P'$ if there is a state σ such that $P\,;\sigma \to P'\,;\sigma$. As usual, silent actions label transitions that do not modify the environment – in our case the global state – and so are invisible to an outside observer. Unlike CCS, the calculus has more examples of silent transitions than mere internal synchronization, e.g. the initiation and evolution of an atomic block, rules (ATST) and (ATPASS). Consequently, a suitable (weak) equivalence for ATCCS should not distinguish e.g. the processes $atom(retry)$, $atom(end)$, $(a.\overline{a})$ and $\mathbf{0}$. The same is true with input transitions. For instance, we expect to equate the processes $a.\mathbf{0}$ and $atom(rd(a).\, end)$.

Our labeled semantics for ATCCS is not based on a set of transition rules, as it is usually the case. Nonetheless, we can recover an axiomatic presentation of the semantics using the tight correspondence between labeled transitions and reductions characterized by the following proposition.

Proposition 6.1. *Consider two processes P and R. The following implications are true:*

(COM) *if* $P \xrightarrow{a} P'$ *and* $R \xrightarrow{\overline{a}} R'$ *then* $P \mid R \xrightarrow{\tau} P' \mid R'$;

(PAR) *if* $P \xrightarrow{\mu} P'$ *then* $P \mid R \xrightarrow{\mu} P' \mid R$ *and* $R \mid P \xrightarrow{\mu} R \mid P'$;

(HID) *if* $P \xrightarrow{\mu} P'$ *and* $a \in \mathcal{N}$ *does not appear in* μ *then* $P \setminus^n a \xrightarrow{\mu} P' \setminus^n a$;

(HIDOUT) *if* $P \xrightarrow{\overline{a}} P'$ *then* $P \setminus^n a \xrightarrow{\tau} P' \setminus^{n+1} a$;

(HIDAT) *if* $P \xrightarrow{\mu} P'$ *and* $\mu = \theta \uplus \{a^m\}$, *where* $a \in \mathcal{N}$ *does not appear in the label* θ, *then* $P \setminus^{n+m} a \xrightarrow{\theta} P' \setminus^n a$.

PROOF: In each case, we have a transition of the form $P \xrightarrow{\mu} P'$. By definition, there are states σ and σ' such that $P\,;\sigma \to P'\,;\sigma'$. The property is obtained by a simple induction on this reduction (a case analysis on the last reduction rule is enough). □

Notations. We denote by $\Rightarrow$ the *weak transition relation*, that is the reflexive and transitive closure of $\xrightarrow{\tau}$. We denote by $\overset{\mu}{\Rightarrow}$ the relation $\Rightarrow$ if $\mu = \tau$ and $\Rightarrow \xrightarrow{\mu} \Rightarrow$ otherwise. If s is a sequence of labels $\mu_0 \cdots \mu_n$, we denote $\xrightarrow{s}$ the relation such that $P \xrightarrow{s} P'$ if and only if there is a process R such that $P \xrightarrow{\mu_0} R$ and $R \xrightarrow{\mu_1 \cdots \mu_n} P'$ and $\xrightarrow{s}$ is the identity relation when s is the empty sequence ϵ. We also define a weak version $\overset{s}{\Rightarrow}$ of this relation in the same way.

6.3.2 Asynchronous bisimulation

Equipped with a LTS, we can define a weak *asynchronous bisimulation* relation, denoted $\approx_{\mathsf{a}}$, in the style of [12].

Definition 6.3 (weak asynchronous bisimulation). *A symmetric relation* $\mathcal{R}$ *is a* weak asynchronous bisimulation *if whenever* $P\mathcal{R}S$ *then the following holds:*

(1) *if* $P \xrightarrow{\overline{a}} P'$ *then there is* S' *such that* $S \overset{\overline{a}}{\Rightarrow} S'$ *and* $P'\mathcal{R}S'$;

(2) *if* $P \xrightarrow{\theta} P'$ *then there is a process* S' *and a block action* γ *such that* $S \overset{\gamma}{\Rightarrow} S'$ *and* $\left(P' \mid \prod_{a \in (\gamma \setminus \theta)} \overline{a}\right) \mathcal{R} \left(S' \mid \prod_{a \in (\theta \setminus \gamma)} \overline{a}\right)$.

We denote with $\approx_{\mathsf{a}}$ *the largest weak asynchronous bisimulation.*

Assume $P \approx_{\mathsf{a}} S$ and $P \xrightarrow{\tau} P'$, the (derived) case for silent action entails that there is S' and θ such that $S \overset{\theta}{\Rightarrow} S'$ and $P' \mid \prod_{a \in \theta} \overline{a} \approx_{\mathsf{a}} S'$. If θ is the silent action, $\theta = \{\,\}$, we recover the usual condition for bisimulation, that is $S \Rightarrow S'$ and $P' \approx_{\mathsf{a}} S'$. If θ is an input action, $\theta = \{a\}$, we recover the definition of asynchronous bisimulation of [12]. Due to the presence of block actions γ, the definition of $\approx_{\mathsf{a}}$ is slightly more complicated than in [12], but it is also more compact – there are only two cases – and more symmetric. Hence, we expect to be able to reuse known methods and tools

for proving the equivalence of AtCCS processes. Another indication that $\approx_{\mathsf{a}}$ is a good choice for reasoning about processes is that it is a congruence.

Theorem 6.1. *Weak asynchronous bisimulation $\approx_{\mathsf{a}}$ is a congruence.*

Proof: It suffices to prove that $\approx_{\mathsf{a}}$ is preserved by every operator of the calculus; the proof is reported in Appendix B, Section B.1. □

We need to define a specific equivalence relation to reason on transactions. Indeed, the obvious choice that equates two expressions M and N if $atom(M) \approx_{\mathsf{a}} atom(N)$ does not lead to a congruence. For instance, we have $atom(rd(a).wt(a).end) \approx_{\mathsf{a}} atom(end)$ while $atom(rd(a).wt(a).end \;\; orElse \;\; wt(b).end) \not\approx_{\mathsf{a}} atom(end \;\; orElse \;\; wt(b).end)$. The first transaction may output a message on b while the second always end silently.

We define an equivalence relation between atomic expressions $\backsimeq$, and a *weak atomic preorder* $\sqsupseteq$, that relates two expressions if they end (or abort) for the same states. We also ask that equivalent expressions should perform the same changes on the global state when they end. We say that two logs δ, δ' have *same effects*, denoted $\delta =_\sigma \delta'$ if $\sigma \setminus \mathrm{RD}(\delta) \uplus \mathrm{WT}(\delta) = \sigma \setminus \mathrm{RD}(\delta') \uplus \mathrm{WT}(\delta')$.

Definition 6.4 (weak atomic equivalence). *$M \sqsupseteq_\sigma N$ if and only if*

(1) *either $(N)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma,\delta}$;*

(2) *or $(N)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma,\delta}$ and $(M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta'}$.*

$M \backsimeq_\sigma N$ if and only if

(1) *either $(M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma,\delta}$ and $(N)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma,\delta'}$;*

(2) *or $(M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta}$ and $(N)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma,\delta'}$ with $\delta =_\sigma \delta'$.*

Two atomic expressions M, N are atomic equivalent*, denoted $M \backsimeq N$, if and only if $M \backsimeq_\sigma N$ for every state σ. Similarly, we have $M \sqsupseteq N$ if and only if $M \sqsupseteq_\sigma N$ for every state σ.*

Note that $atom(M) \approx_{\mathsf{a}} atom(N)$ does not imply $M \backsimeq N$. E.g. $atom(rd(a).wt(a).end) \approx_{\mathsf{a}} atom(end)$ but $rd(a).wt(a).end \not\backsimeq end$ – in fact $rd(a).wt(a).end \not\backsimeq_\emptyset end$. As stated by the following proposition, the vice versa is true; the proof is reported in Appendix B, Section B.1.

Proposition 6.2. *$M \backsimeq N$ implies $atom(M) \approx_{\mathsf{a}} atom(N)$.*

Even though of the definitions of $\sqsupseteq$ and $\backsimeq$ depend on a universal quantification over states, testing the equivalence of two expressions is not expensive. First, we can rely on a monotonicity property of reduction: if $\sigma \subseteq \sigma'$ then for all M the effect of $(M)_{\sigma,\delta}$ is included in those of $(M)_{\sigma',\delta}$. Moreover, we define a normal form for expressions later in

Laws for atomic expressions:			
(COMM)	$\alpha.\beta.M$	$\preceq$	$\beta.\alpha.M$
(DIST)	$\alpha.(M\ orElse\ N)$	$\preceq$	$(\alpha.M)\ orElse\ (\alpha.N)$
(ASS)	$M_1\ orElse\ (M_2\ orElse\ M_3)$	$\preceq$	$(M_1\ orElse\ M_2)\ orElse\ M_3$
(IDEM)	$M\ orElse\ M$	$\preceq$	M
(ABSRT1)	$\alpha.\,retry$	$\preceq$	$retry$
(ABSRT2)	$retry\ orElse\ M$	$\preceq$	$M \preceq M\ orElse\ retry$
(ABSEND)	$end\ orElse\ M$	$\preceq$	end
Laws for processes:			
(ASY)	$a.\overline{a}$	$\approx_{\mathsf{a}}$	$\mathbf{0}$
(A-ASY)	$atom(rd(a).\,wt(a).\,end)$	$\approx_{\mathsf{a}}$	$\mathbf{0}$
(A-1)	$atom(rd(a).\,end)$	$\approx_{\mathsf{a}}$	$a.\mathbf{0}$

Table 6.5: Algebraic laws of transactions.

this section (see Proposition 6.3) that greatly simplifies the comparison of expressions. Another indication that $\preceq$ is a good choice of equivalence for atomic expressions is that it is a congruence; the proof is reported in Appendix B, Section B.1.

Theorem 6.2. *Weak atomic equivalence $\preceq$ is a congruence.*

On the Algebraic Structure of Transactions. The equivalence relations $\preceq$ and $\approx_{\mathsf{a}}$ can be used to prove interesting laws of atomic expressions and processes. We list some of these laws in Table 6.5, proofs are reported in Appendix B, Section B.2. The behavioral rules for atomic expressions are particularly interesting since they exhibit a rich algebraic structure for $\mathcal{M}$. For instance, rules (COMM) and (DIST) state that action prefix $\alpha.M$ is a commutative operation that distribute over $orElse$. We also have that $(\mathcal{M}, orElse, retry)$ is an idempotent semigroup with identity $retry$, rules (ASS), (ABSRT2) and (IDEM), and that end annihilates $\mathcal{M}$, rule (ABSEND). Most of these laws appear in [81] but are not formally proved.

Actually, we can show that the structure of $\mathcal{M}$ is close to that of a bound join-semilattice. We assume unary function symbols $a(\,)$ and $\overline{a}(\,)$ for every name a (a term $\overline{a}(M)$ is intended to represent a prefix $wt(a).M$) and use the symbols $\sqcup, 1, 0$ instead of $orElse, end, retry$. With this presentation, the behavioral laws for atomic expression are almost those of a semilattice. By definition of $\sqsupseteq$, we have that $M \sqcup M' \preceq M$ if

and only if $M \sqsupseteq M'$ and for all M, N we have $1 \sqsupseteq M \sqcup N \sqsupseteq M \sqsupseteq 0$.

$$\mu(\mu'(M)) \;\backsimeq\; \mu'(\mu(M)) \qquad \mu(M \sqcup N) \;\backsimeq\; \mu(M) \sqcup \mu(N) \qquad \mu(0) \;\backsimeq\; 0$$
$$0 \sqcup M \;\backsimeq\; M \;\backsimeq\; M \sqcup 0 \qquad 1 \sqcup M \;\backsimeq\; 1$$

It is possible to prove other behavioral laws to support our interpretation of $orElse$ as a join. However some important properties are missing, most notably, while $\sqcup$ is associative, it is not commutative. For instance, $a(\overline{b}(1)) \sqcup 1 \not\backsimeq 1$ while $1 \backsimeq 1 \sqcup a(\overline{b}(1))$, rule (absEnd). This observation could help improve the design of the transaction language: it will be interesting to enrich the language so that we obtain a real lattice.

Normal Form for Transactions. Next, we show that behavioural laws can be used to rearrange an atomic expression to put it into a simple *normal form*. This procedure can be understood as a kind of compilation that transform an expression M into a simpler form.

Informally, an atomic expression M is said to be in *normal form* if it does not contain nested $orElse$ – all occurrences are at top level – and if there are no *redundant branches*. A redundant branch is a sequence of actions that will never be executed. For instance, the read actions in $rd(a).end$ are included in $rd(a).rd(b).end$, then the second branch in the composition $\big(rd(a).end\big)\ orElse\ \big(rd(a).rd(b).end\big)$ is redundant: obviously, if $rd(a).end$ fails then $rd(a).rd(b).end$ cannot succeed. We overload the functions defined on logs and write $\text{RD}(M)$ for the multiset of names occurring in read actions in M. We define $\text{WT}(M)$ similarly. In what follows, we abbreviate $(M_1\ orElse\ \cdots\ orElse\ M_n)$ with the expression $\bigsqcup_{i=1,\ldots,n} M_i$.

Definition 6.5 (normal-form). *We say that an expression M is in* normal form *if it is of the form $\bigsqcup_{i=1,\ldots,n} K_i$ where it holds that*

(1) *K_i is a sequence of action prefixes $\alpha_{j_1}.\cdots.\alpha_{j_{n_i}}.end$, with $i = 1, \ldots, n$ and $n_i \geqslant 0$;*

(2) *$\text{RD}(K_i) \not\subseteq \text{RD}(K_j)$ for all $i < j$, with $i, j \in 1, \ldots, n$.*

Condition (1) requires the absence of nested $orElse$ and condition (2) prohibits redundant branches, moreover it also ensures that all branches, but the last one, has at least a read action.

Proposition 6.3. *For every expression M there is a normal form M' such that $M \backsimeq M'$.*

Proof: Laws (comm), (dist) and (ass) in Table 6.5 can be applied for eliminating nested $orElse$. Next, we use the fact that if K is a redundant branch of M then $M \sqsupseteq K$. More details regarding the proof are reported in Section B.3. □

Our choice of using bisimulation for reasoning about atomic transactions may appear arbitrary. We have already debated over the need to consider asynchronous bisimulation $\approx_{\mathsf{a}}$ instead of (simple) bisimulation $\approx$. In the next section, we study a testing equivalence for AtCCS, more particularly an asynchronous may testing semantics [65].

Example 6.3.1 (dining philosophers). In this example we give yet another solution to the well-known dining philosophers problem. We use atomic blocks of actions in the implementation of the system and we show that the obtained process behaves as its specification, without using backtracking and without falling into situations of deadlock. Suppose to have four philosophers, $I = \{0, 1, 2, 3\}$ is the considered set of indexes. In what follows we write $i + i'$ for the sum modulo 4 of i and i' and in $P + R$ we consider $+$ as the usual nondeterministic CCS choice between P and R. Suppose t is a set of indexes corresponding to thinking philosophers, which are ready to eat; and e corresponds to eating philosophers, which are ready to think. $P_{t;e}$ is the specification of the system: $t \cup e = I$ (each philosopher eats or thinks), $t \cap e = \emptyset$ (none can eat and think at the same time) and for no $i \in I$ it holds that $i, i + 1 \in e$ (two adjacent philosophers cannot eat simultaneously).

$$\begin{array}{rcl} P_{t;e} & \stackrel{\triangle}{=} & \sum_{i \notin t} t_i.P_{t \cup i;e-i} \\ & + & \sum_{\{i=0,1 \text{ if } e=\emptyset\}} \tau.(e_i.P_{t-i;i} + e_{i+2}.P_{t-(i+2);(i+2)}) \\ & + & \sum_{\{i \in t \,|\, i-1, i+1 \notin e,\, i+2 \in e\}} \tau.S_i \\ S_i & \stackrel{\triangle}{=} & e_i.P_{\{i-1,i+1\};\{i,i+2\}} \\ & + & t_{i+2}.\big(e_{i+2}.S_i \,+\, e_i.P_{\{i-1,i+1,i+2\};\{i\}}\big) \end{array}$$

The actions of eating, e_i, and thinking, t_i, of the philosopher i can be observed as inputs. There are no restrictions for eating philosophers that wants to start thinking, first branch. If none is eating, non-deterministically philosophers with either odd or pair indexes are allowed to eat, second branch. If philosopher i is already eating, its neighbors $i - 1$ and $i + 1$ cannot eat, while it is allowed to its opposite $i + 2$, third branch. The system specification will never fall into deadlocks and there can be at most two simultaneously eating philosophers (with indexes i and $i + 2$).

A philosopher D_i, for $i \in I$, can be implemented as follows:

$$D_i \stackrel{\triangle}{=} atom(rd(c_{i-1}).\, rd(c_i).\, end).e_i.t_i.(\overline{c}_{i-1} \,|\, \overline{c}_i).$$

Process D_i attempts to get the chopsticks, on his right and left, by using an atomic block for reading both c_{i-1} and c_i. If the commit of the atomic block cannot be

performed, then at least one of its neighbors, D_{i-1} or D_{i+1} is already eating, because at least one of the chopsticks is not available, thus D_i will retry to get both chopsticks. Otherwise he can eat, thus he will acquire the chopsticks and eat by inputting e_i. After eating, he can decide to start thinking, thus he reads t_i, and after both chopsticks can be released.

The global system is given by the parallel composition of the philosopher D_i and the output of the 4 chopsticks, which are hidden to observers.

$$D \triangleq (D_0 \,|\, D_1 \,|\, D_2 \,|\, D_3 \,|\, \overline{c}_0 \,|\, \overline{c}_1 \,|\, \overline{c}_2 \,|\, \overline{c}_3) \setminus^0 c_0, c_1, c_2, c_3.$$

In what follows we show that $P_{I;\emptyset} \approx_{\mathsf{a}} D$ holds. Before we define a useful abbreviation. Suppose A, B, C, D, $E \subseteq \{0, 1, 2, 3\}$, are sets of indexes such that $A \cup B \cup C = \{0, 1, 2, 3\}$, $A \cap B = A \cap C = B \cap C = \emptyset$ and $D \cup E \subseteq \{0, 1, 2, 3\}$ with $D \cap E = \emptyset$. We define $D\{A; B; C; D; E\}$ as follows:

$$\begin{array}{rcl} D\{A; B; C; D; E\} & \triangleq & (\prod_{\{i \in A\}} D_i \mid \prod_{\{i \in B\}} e_i.t_i.(\overline{c}_{i-1} \,|\, \overline{c}_i) \\ & & \mid \prod_{\{i \in C\}} t_i.(\overline{c}_{i-1} \,|\, \overline{c}_i) \\ & & \mid \prod_{\{i \in D\}} \overline{c}_i) \setminus^1 c_{i,\, i \in E} \setminus^0 c_{i,\, i \in I \setminus E}. \end{array}$$

That is a system where the philosophers in A are in the initial state; philosophers in B are ready to eat (they have already acquired both chopsticks); philosophers in C are ready to think (they have already eaten); indexes in D correspond to available chopsticks not yet outputted; indexes in E correspond to chopsticks outputted, thus chopsticks that are available (part of the actual state).

In the following $\mathcal{P}(S)$ represents the powerset of S. $P_{I;\emptyset}\, \mathcal{R}\, D\{I; \emptyset; \emptyset; I; \emptyset\}$ where the bisimulation $\mathcal{R}$ is defined as follows:

$$\begin{array}{l} \mathcal{R} = \Big\{ \big(P_{I;\emptyset}, D\{I; \emptyset; \emptyset; I \setminus S; S\}\big) \,\big|\, S \in \mathcal{P}(I) \Big\} \\ \cup \Big\{ \big(P_{I-i;i}, D\{I - i; \emptyset; \{i\}; \{i+1, i+2\} \setminus S; S\}\big) \,\big|\, S \in \mathcal{P}(\{i+1, i+2\}),\, i = 0, 1, 2, 3 \Big\} \\ \cup \Big\{ \big(S_{i+2}, D\{\{i-1, i+1\}; \{i+2\}; \{i\}; \emptyset; \emptyset\}\big) \,\big|\, i = 0, 1, 2, 3 \Big\} \\ \cup \Big\{ \big(P_{\{i-1,i+1\};\{i,i+2\}}, D\{\{i-1, i+1\}; \emptyset; \{i, i+2\}; \emptyset; \emptyset\}\big) \,\big|\, i = 0, 1 \Big\} \\ \cup \Big\{ \big((e_i.P_{I-i;i} + e_{i+2}.P_{I-(i+2);(i+2)}), D\{\{i-1, i+1\}; \{i+2, i\}; \emptyset; \emptyset; \emptyset\}\big) \,\big|\, i = 0, 1 \Big\} \\ \cup \Big\{ \big((e_i.P_{I-i;i} + e_{i+2}.P_{I-(i+2);(i+2)}), D\{\{i-1, i, i+1\}; \{i+2\}; \emptyset; \{i-1, i\} \setminus S; S\}\big) \\ \qquad \big|\, S \in \mathcal{P}(\{i-1, i\}),\, i = 0, 1, 2, 3 \Big\}\,. \end{array}$$

6.4 May-testing semantics

Using a testing equivalence instead of bisimulation is sometimes more convenient. Nonetheless, testing equivalences have the drawback that their definition depends on a universal quantification over arbitrarily many processes. We define a may-testing equivalence for AtCCS and give an alternative characterization using a trace-based equivalence. We also expose some shortcomings of may testing related to the (folklore) fact that it cannot distinguish the points of choice in a process. Actually, we define for every atomic block $atom(M)$ a corresponding process without transactions, but using choice, that is (may-testing) indistinguishable from $atom(M)$.

We define the notion of observers and successful computations. An *observer* O is a particular type of process which does not contain atomic blocks and that can perform a distinct output $\overline{w}$ (the success action). We denote $\mathcal{O}bs$ the set of all observers. A *computation* from a process P and an observer O is a sequence of transitions of the form $P \mid O = P_0 \mid O_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_k \mid O_k \xrightarrow{\tau} \cdots$, which is of either infinite or finite size, say n, such that $P_n \mid O_n$ cannot evolve. A computation from $P \mid O$ is *successful* if there is an index m such that O_m has a success action, that is $O_m \xrightarrow{\overline{w}}$. In this case, we say that P *may* O. Two processes are may testing equivalent if they have the same successful observers.

Definition 6.6 (may-testing preorder). *Given two processes P and R, we write $P \sqsubseteq_{may} R$ if for every observer O in $\mathcal{O}bs$ we have P may O implies R may O. We use $\simeq_{may}$ to denote the equivalence obtained as the kernel of the preorder $\sqsubseteq_{may}$.*

Universal quantification on observers make it difficult to work with the operational definition of the may preorder. Following [30], we study a trace-based characterization for our calculus. The following preorder over traces will be used for defining the alternative characterization of the may-testing preorder. Proofs of this section are almost standard (see e.g. [30]) and for the sake of completeness are reported in Section B.4.

In what follows, a *trace* s is an element of $(Out \cup Ib)^*$, that is a sequence of actions $\mu_1 \cdots \mu_n$ where we only consider outputs and block actions and leave aside τ and input actions, which are derivable.

Definition 6.7 (preorder over traces). *Let $\preccurlyeq_0$ be the least relation on traces that satisfies the following laws:*

$$(\text{TO1})\quad s_1\, s_2 \preccurlyeq_0 s_1\, \{\!|a|\!\}\, s_2 \qquad (\text{TO2})\quad s_1\, s_2\, \{\!|a|\!\}\, s_3 \preccurlyeq_0 s_1\, \{\!|a|\!\}\, s_2\, s_3$$

$$(\text{TO3})\quad s_1\, s_2 \preccurlyeq_0 s_1\, \{\!|a|\!\}\, \overline{a}\, s_2 \qquad (\text{TO4})\quad \{\!|a_1, \ldots, a_n|\!\} \;{}_0\!\succcurlyeq \preccurlyeq_0 \{\!|a_1|\!\} \cdots \{\!|a_n|\!\}$$

The preorder $\preccurlyeq$ is the reflexive and transitive closure of $\preccurlyeq_0$.

Following the terminology of [30], (TO1), (TO2) and (TO3) are the laws for *deletion*, *postponement* and *annihilation* of input action. We add rule (TO4) which allows to substitute block actions with the corresponding sequences of inputs. The preorder $\preccurlyeq$ is preserved by prefixing. We can now define a preorder over processes.

Definition 6.8 (alternative preorder). *For processes P and Q, we set $P \ll_{may} Q$ if for all weak transition $P \stackrel{s}{\Rightarrow} P'$ there is a trace s' and a process Q' such that $s' \preccurlyeq s$ and $Q \stackrel{s'}{\Rightarrow} Q'$.*

We now prove coincidence of $\ll_{may}$ and $\sqsubseteq_{may}$. Some definitions and preliminary results are needed. For every label μ we define the *complement* $\overline{\mu}$ such that: the complement of an output action $\overline{a}$ is a block action $\{\!|a|\!\}$ and the complement of a block action $\{\!|a_1, \ldots, a_n|\!\}$ is a trace $\overline{a}_1 \cdots \overline{a}_n$. For every trace $s = \mu_1 \cdots \mu_n$, the cotrace $\overline{s} = \overline{\mu}_1 \cdots \overline{\mu}_n$ is obtained by concatenating the complements of the actions in s. The following lemma relates the preorder $\preccurlyeq$ with the operational semantics of processes.

Lemma 6.1. *Assume that $s' \preccurlyeq s$ and $P \stackrel{\overline{s}}{\Rightarrow} P'$, then there is a process P'' such that $P \stackrel{\overline{s}'}{\Rightarrow} P''$.*

The next step is to define a special class of observers. For every trace s, we inductively define an observer $\mathcal{O}(s) \in \mathcal{O}bs$ as follows:

$$\mathcal{O}(\epsilon) \stackrel{\triangle}{=} \overline{w}, \quad \mathcal{O}(\overline{a}\, s) \stackrel{\triangle}{=} a.\mathcal{O}(s), \quad \mathcal{O}(\{\!|a_1, \ldots, a_n|\!\}\, s) \stackrel{\triangle}{=} \Big(\prod_{i=1,\ldots,n} \overline{a}_i \Big) \mid \mathcal{O}(s) \ .$$

The following property shows that the sequence of visible actions from $\mathcal{O}(s)$ is related to traces simulated by s.

Lemma 6.2. *Consider two traces s and r. If there is a process R such that $\mathcal{O}(s) \stackrel{\overline{r}}{\Rightarrow} \stackrel{\overline{w}}{\Rightarrow} R$ then $r \preccurlyeq s$.*

We can now prove a full abstraction theorem between may testing $\sqsubseteq_{may}$ and the alternative preorder $\ll_{may}$.

Theorem 6.3. *For all processes P and R, we have $P \sqsubseteq_{may} R$ if and only if $P \ll_{may} R$.*

Next, we show that may-testing semantics is not precise enough to tell apart atomic transactions from sequences of input actions. We consider an atomic expression M in normal form. Assume $M = \bigsqcup_{i=1,\ldots,n} K_i$, the following lemma state that the observing behavior of M is obtained by considering, for every branch K_i, a transition labeled by the block action containing $\text{RD}(K_i)$ followed by output transitions on the names in $\text{WT}(K_i)$.

Lemma 6.3. *Assume $M = \bigsqcup_{i=1,\ldots,n} K_i$ is an expression in normal form. For every index i in $\{1,\ldots,n\}$ we have $atom(M) ; \sigma_i \rightarrow^* \{\!|(end)_{\sigma_i;\delta}|\!\}_M ; \sigma_i$ where $\sigma_i = \text{RD}(K_i) = \text{RD}(\delta)$ and $\text{WT}(\delta) = \text{WT}(K_i)$.*

As a corollary of Lemma 6.3, we obtain that the possible behavior of $atom(M)$ can be described as $atom(M) \xrightarrow{\text{RD}(K_i)} \prod_{b\in\text{WT}(K_i)} \overline{b}$ for every $i = 1,\ldots,n$.

We now prove that for every atomic transaction $atom(M)$ there is a CCS process $[\![M]\!]$ that is may-testing equivalent to M. By CCS process, we intend a term of AtCCS without atomic transactions that may include occurrences of the choice operator $P+R$. By Proposition 6.3, we can assume that M is in normal form, that is $M = \bigsqcup_{i=1,\ldots,n} K_i$. The interpretation of a sequence of actions $K = \alpha_1.\cdots.\alpha_n.end$ is the process $[\![K]\!] = a_1.\cdots.a_k.\big(\overline{b_1} \mid \cdots \mid \overline{b_l}\big)$ where $\{a_1,\ldots,a_k\} = \text{RD}(K)$ and $\{b_1,\ldots,b_l\} = \text{WT}(K)$. (In particular we have $[\![end]\!] = \mathbf{0}$.) The translated of M, denoted $[\![M]\!]$, is the process $[\![K_1]\!]+\cdots+[\![K_n]\!]$. The following theorem proves that may-testing semantics is not able to distinguish the behavior of an atomic process from the behavior of its translation, which means that may-testing is blind to the presence of transactions.

Proposition 6.4. *For every expression M in normal form we have $atom(M) \simeq_{may} [\![M]\!]$.*

PROOF: The proof uses the characterization of may testing in term of the alternative preorder. We show separately that $atom(M) \ll_{may} [\![M]\!]$ and $[\![M]\!] \ll_{may} atom(M)$. A complete proof can be found at the end of Section B.4. □

We observe that a process $[\![M]\!]$ is a choice between processes of the form $a.P$ or $\big(\prod_{i\in I} \overline{b}_i\big)$. Therefore, using internal choice and a slightly more convoluted encoding, it is possible to use only input guarded choice $a.P + b.R$ in place of full choice in the definition of $[\![M]\!]$.

Example 6.4.1. Consider the atomic process $M = rd(a).rd(b).wt(d).rd(c)$ and its translation $[\![M]\!] = a.b.c.\overline{d}$; $atom(M) \simeq_{may} [\![M]\!]$ because:

(⇒): $atom(M) \ll_{may} [\![M]\!]$ (that is $atom(M) \sqsubseteq_{may} [\![M]\!]$): $atom(M) \stackrel{s}{\Rightarrow}$ with either $s = \{\!|a,b,c|\!\}$ or $s = \{\!|a,b,c|\!\}\overline{d}$ and $[\![M]\!] \stackrel{s'}{\Rightarrow}$ with either $s' = \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\} \preccurlyeq \{\!|a,b,c|\!\}$ or $s' = \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\}\overline{d} \preccurlyeq \{\!|a,b,c|\!\}\overline{d}$, (TO4);

(⇐): $[\![M]\!] \ll_{may} atom(M)$ (that is $[\![M]\!] \sqsubseteq_{may} atom(M)$): $[\![M]\!] \stackrel{s}{\Rightarrow}$ with $s \in \{\{\!|a|\!\}, \{\!|a|\!\}\{\!|b|\!\}, \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\}, \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\}\overline{d}\}$ and $atom(M) \xrightarrow{s'}$ with either $s' = \epsilon \preccurlyeq \{\!|a|\!\}, \{\!|a|\!\}\{\!|b|\!\}$, (TO1), or $s' = \{\!|abc|\!\} \preccurlyeq \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\}$ or $s' = \{\!|abc|\!\}\overline{d} \preccurlyeq \{\!|a|\!\}\{\!|b|\!\}\{\!|c|\!\}\overline{d}$, (TO4).

6.5 Conclusions

In this chapter we have studied a formal model for atomicity based on logs. We have defined AtCCS, a process calculus based on a shared memory system, which extends the asynchronous CCS by adding atomic blocks of actions. The calculus deals with atomicity constraints by checking logs consistency at commit time instead of by using locks. We have shown that the calculus is expressive enough to encode interesting concurrency primitives, such as preemptive versions of guarded-choice and multiset-synchronization. We have also introduced new solutions to the well-known leader election and dining philosophers problems. We have defined two equivalences, for processes and atomic expressions, and we have shown that both are congruences. These equivalences are used to prove the correctness of the encodings, to prove interesting "behavioral laws" and to define a simple normal-form for transactions. AtCCS can be viewed as a starting point in the definition of a transactional calculus for WS. A relevant limitation of the calculus is that it allows only "basic" atomic processes, which are simple sequences of actions. Only the $orElse$ construct can add complexity to atomic blocks: neither concurrency, nor restriction and nesting are allowed. In the future, it would be interesting to overcome these limitations and to enrich the language with parallel composition and synchronization of atomic expressions and compensating and nested transactions. The final calculus would allow to completely modelize and analize the transactional behavior of WS, as described by using the WS-Transaction specification [91].

Chapter

SEVEN

Conclusions

We have attempted to give a process-algebraic account of some important aspects of SOA and WS.

In XPi we target *communication*. We model WS as communication-centered applications. We focus our attention on message exchange and processing and on services interaction. The type system we define is actually very basic, but sufficient to regulate messaging and ensuring that communications never produce typing errors. In other words, that clients and servers always understand with each other.

In Astuce we deal with the problems related to REST WS, and in particular we focus on *distribution* of resources (documents). Our intent is to study the approach usually followed in search engines by considering some of its basic aspects. To this end, we propose a process calculus based on distribution of documents and concurrent pattern-matching evaluations. Again, our attention is focused on the processing model. The type system we propose is inspired by existing works and is based on regular expression types. The presence of a type system here is less fundamental than in XPi. In fact, since pattern-matching, by definition, can fail, the type system cannot be used for ensuring the absence of failures. Its aim is to guarantee the validity of documents and of retrieved information.

Finally, we study two relevant non-functional aspects of WS: *responsiveness* and *transactionality*. The type systems introduced for ensuring responsiveness of services allow one to analyze the behavior of processes and statically guarantee that a reply will eventually follow each request. That is, they guarantee users that the required service will be supplied. The definition and implementation of inference systems for both proposals are left as future works. In AtCCS we address the problem of dealing with failures by guaranteeing the atomic execution of transactions. We define

transactions as (basic) blocks of actions where only the *orElse* construct can add complexity: neither concurrency, nor restriction and nesting are allowed. We have chosen to follow this simple approach with the aim of defining a powerful operational semantics for STM and a "clean" notion of equivalence for processes and atomic transactions.

In this thesis we have focused only on some important aspects of WS, and we have defined formal methods for reasoning on them. But there are still a lot of aspects we have not considered and a lot of work to do in this direction.

For instance, we have not directly addressed neither the coordination and orchestration aspects, nor the security-related problems. A great deal of these aspects is expressible as liveness and safety properties written in suitable temporal logics. For example, we can use logics for guaranteeing that a certain order in a sequence of calls and communications is respected – hence for coreography – , for guaranteeing that each access to confidential information is preceded by authentication of the user and so on. At present, we are investigating this possibility. We would like to introduce methods for guaranteeing (behavioral) properties of services more involved than just responsiveness, lock-freedom and termination. For instance, it would be useful to guarantee that a service operation, let us say *ship_good*, will be invoked only after a call to another operation, let us say *get_money* (a safety property), or that a call to a service, let us say *pay*, is always followed by a notification *get_good* (a liveness property). In this respect, behavioral types come into play. Taking inspiration from the type system of Igarashi and Kobayashi [92], the idea is to define methods for statically abstracting "propositional" approximations (in terms of CCS, Petri Nets, and so on) of "first-order" process calculi, such as π-calculus and Join-calculus. More precisely, let us consider a context associating values and free names of processes with *tags* belonging to a finite set. Tags may represent particular events an external observer is interested in. If we observe the behavior of a process "through" this context – e.g. by substituting values and names in transition labels with the corresponding tags – we obtain an abstraction of the behavior of the whole initial system under this context. Our aim is to define means to statically compute these abstractions. In particular we would like to obtain suitable (over-)approximations of the behavior of processes and to verify certain properties on these approximations, being assured that the same properties also hold for the abstracted processes.

We have already started working in this direction in [9], where we propose behavioral type systems for abstracting π-calculus and Join-calculus processes. Further

studies could aim at defining suitable combinations of type checking and model checking techniques for property verification and at defining suitable inference systems. It would also be worthwhile to study this problems in terms of abstract interpretation [60] and comparing this approach to the one we follow in [9].

References

[1] W.M.P. Aalst, K.M. Hee and G.J. Houben. Modelling workflow management systems with high-level petri nets. In *Proceedings of the second Workshop on Computer Supported Cooperative Work, Petrinets and related formalisms*, 31–50, 1994.

[2] M. Abadi and A.D. Gordon. Mobile values, new names, and secure communication. In *Proceedings of POPL*, 104–115, ACM Press, 2001.

[3] M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Academic Press, 1999.

[4] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *Proceedings of VLDB*, 1078–1090, Morgan Kaufmann, 2002.

[5] L. Acciai and M. Boreale. XPi: a Typed Process Calculus for XML Messaging. In *Proceedings of FMOODS*, *Lecture Notes in Computer Science*, 3535:47–66, Springer-Verlag, 2005.

[6] L. Acciai, M. Boreale and S. Dal Zilio. A Typed Calculus for Querying Distributed XML Documents. In *Proceedings of TGC*, 2006. *Lecture Notes in Computer Science*, 4661:167–182, Springer-Verlag, 2007. A long version appears as LIF Research Report 29, 2006.

[7] L. Acciai and M. Boreale. Responsiveness in Process Calculi. In *Proceedings of ASIAN*, 2006. To appear in *Lecture Notes in Computer Science*.

[8] L. Acciai, M. Boreale and S. Dal Zilio. A Concurrent Calculus with Atomic Transactions. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 4421:48–63, Springer-Verlag, 2007.

[9] L. Acciai and M. Boreale. Type abstractions of name passing processes. In *Proceedings of FSEN*, 2007. To appear in *Lecture Notes in Computer Science*.

[10] M. Altinel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination information. In *Proceedings of VLDB*, 53–64, Morgan Kaufmann, 2000.

[11] R. Amadio. An Asynchronous Model of Locality, Failure And Process Mobility. In *Proceedings of COORDINATION*, *Lecture Notes in Computer Science*, 1282:374–391, Springer-Verlag, 1997.

[12] R. Amadio, I. Castellani and D. Sangiorgi. On Bisimulation for the Asybchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, Elsevier, 1998.

[13] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana. Business Process Execution Language for Web Services, v1.1, 2003. Available at `http://www.oasis-open.org/committees/download.php/2046/BPEL%20V1-1%20May%205%202003%20Final.pdf`.

[14] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. Acta Informatica, 29(8):737–760, 1992.

[15] B. Benatallah, F. Casati, J. Ponge and F. Toumani. On temporal abstractions of Web Services Protocols. In *Proceedings of CAiSE (Short Paper)*, 2005.

[16] V. Benzaken, G. Castagna and A. Frisch. Cduce: An XML-Centric General-Purpose Language. In *Proceedings of ICFP*, 51–63, ACM Press, 2003.

[17] D. Berardi, F. De Rosa, L. De Santis, and M. Mecella. A conceptual model for e-services based on fsa. In *Proceedings of IDPT* 2003.

[18] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull and M. Mecella. Automatic composition of transition-based Web Services with Messaging. In *Proceedings of VLDB*, 613–624, ACM Press, 2005.

[19] M. Berger, K. Honda and N. Yoshida. Sequentiality and the π-calculus. In *Proceedings of TCLA*, *Lecture Notes in Computer Science*, 2044:29–45, Springer-Verlag, 2001.

[20] J.A. Bergstra, A. Ponse and J.J. van Wamel. Process Algebra with Backtracking. In *Proceedings of REX Workshop*, *Lecture Notes in Computer Science*, 803:46–91, Springer-Verlag, 1994.

[21] G.M. Bierman and P. Sewell. Iota: A concurrent XML scripting language with applications to Home Area Networking. Technical Report 577, University of Cambridge Computer Laboratory, 2003.

[22] S. Bhiri, O. Perrin and C. Godart. Ensuring Required Failure Atomicity of Composite Web Services. In *Proceedings of WWW*, 138–147, *ACM* Press, 2005.

[23] Biztalk Server Home. `http://www.microsoft.com/biztalk/`.

[24] S. Bjorg and L.G. Meredith. Contracts and Types. *Communication of the ACM*, 46(10):41–47, October 2003.

[25] A.P. Black, V. Cremet, R. Guerraoui and M. Odersky. An equational theory for Transactions. In *Proceedings of FSTTCS*, *Lecture Notes in Computer Science*, 2914:38–49, Springer-Verlag, 2003.

[26] L. Bocchi, C. Laneve and G. Zavattaro. A calculus for long-running transactions. In *Proceedings of FMOODS*, *Lecture Notes in Computer Science*, 2884:124–138, Springer-Verlag, 2003.

[27] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard. Web Services Architecture. W3C Working Group Note, 11 February 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`.

[28] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2):205–226, Elsevier, 1998.

[29] M. Boreale and D. Sangiorgi. Bisimulation in Name-Passing Calculi without Matching. In *Proceedings of LICS*, 165–175, *IEEE Computer Society Press*, 1998.

[30] M. Boreale, R. De Nicola and R. Pugliese. Trace and Testing Equivalence on Asynchronous Processes. *Information and Computation*, 172(2):139–164, 2002.

[31] 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 *Proceedings of WS-FM*, *Lecture Notes in Computer Science*, 4184:38–57, Springer-Verlag, 2006.

[32] S. Bose, V. Chaluvadi, L. Fegaras and D. Levine. Query Processing of Streamed XML Data In *Proceedings of CIKM*, 126–133, ACM Press, 2002.

[33] S. Bose, V. Chaluvadi, L. Fegaras and D. Levine. A Query Algebra for Fragmented XML Stream Data In *Proceedings of DBPL*, *Lecture Notes in Computer Science*, 2921:195–215, Springer-Verlag, 2003.

[34] A. Brown, C. Laneve and G. Meredith. PiDuce: a process calculus with native XML datatypes. In *Proceedings of EPEW/WS-FM*, *Lecture Notes in Computer Science*, 3670:18–34, Springer-Verlag, 2005.

[35] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2), 1998.

[36] R. Bruni, C. Laneve and U. Montanari. Orchestrating Transactions in Join Calculus. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 2421:321–337, Springer-Verlag, 2002.

[37] R. Bruni, H.C. Melgratti and U. Montanari. Nested Commits for Mobile Calculi: extending Join. *IFIP TCS*, 563–576, Kluwer, 2004.

[38] R. Bruni, H.C. Melgratti and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proceedings of POPL*, 209–220, *ACM* Press, 2005.

[39] R. Bruni, M.J. Butler, C. Ferreira, C.A.R. Hoare, H.C. Melgratti and U. Montanari. Comparing two approaches to compensable flow composition. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 3653:383–397, Springer-Verlag, 2005.

[40] N. Busi, R. Gorrieri and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, Elsevier, 1998.

[41] N. Busi and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. *Electronic Notes in Theoretical Computer Science*, 54, 2001.

[42] M.J. Butler, C.A.R. Hoare and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years Communicating Sequential Processes*, *Lecture Notes in Computer Science*, 3525:133–150, Springer-Verlag, 2004.

[43] M.J. Butler, C.A.R. Hoare and C. Ferreira. An operational semantics for StAC, a language for modeling long running transactions. In *Proceedings of COORDINATION*, *Lecture Notes in Computer Science*, 2949:87–104, Springer-Verlag, 2004.

[44] M.J. Butler, C. Ferreira and M.Y. Ng. Precise modeling of Compensating Business Transactions and its application to BPEL. In *J. UCS*, 11(5):712–743, 2005.

[45] L. Cardelli and G. Ghelli. TQL: A Query Language for Semistructured Data Based on the Ambient Logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.

[46] L. Cardelli, P. Gardner and G. Ghelli. A Spatial Logic for Querying Graphs. In *Proceedings of ICALP*, *Lecture Notes in Computer Science*, 2380:597–610, Springer-Verlag, 2002.

[47] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, Elsevier, 2000.

[48] S. Carpineti and C. Laneve. A Basic Contract Language for Web Services. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 3924:197–213, Springer-Verlag, 2006.

[49] S. Carpineti, G. Castagna, C. Laneve and L. Padovani. A formal account of contracts for web services. In *Proceedings of WS-FM*, *Lecture Notes in Computer Science*, 4184:148–162, Springer-Verlag, 2006.

[50] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy and M. Shan. Adaptive and dynamic service composition in *eflow*. In *Proceedings of CAiSE*, *Lecture Notes in Computer Science*, 1789:13–31, Springer-Verlag, 2000.

[51] G. Castagna. Pattern and types for querying XML documents. In *Proceedings of DBPL*, *Lecture Notes in Computer Science*, 3774:1–26, Springer-Verlag, 2005.

[52] G. Castagna, R. De Nicola and D. Varacca. Semantic subtyping for the π-calculus. In *Proceedings of LICS*, 92–101, IEEE Computer Society Press, 2005.

[53] G. Castagna, N. Gesbert and L. Padovani. A theory of contracts for web services. In *Proceedings of Plan-X*, 2007.

[54] S. Chaki, S.K. Rajamani and J. Rehof. Types as Models: Model Checking Message-Passing Programs. In *Proceedings of POPL*, 45–57, ACM Press, 2002.

[55] C.Y. Chan, P. Felber, M. Garofalakis and R. Rastogi. Efficient filtering of XML documents with XPath expressions. The *VLDB Journal* 11(4):354–379, 2002.

[56] S.S. Chawathe and F. Peng. XPath Queries on Streaming Data. In *Proceedings of SIGMOD*, 431–442, ACM Press, 2003.

[57] T. Chothia and D. Duggan. Abstractions for Fault-Tolerant Global Computing. *Theoretical Computer Science*, 322(3):567–613, Elsevier, 2004.

[58] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web Services Description Language 1.1. W3C Note, 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[59] W.R. Cook and J. Misra. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, 2006. `http://www.cs.utexas.edu/~wcook/projects/orc/`.

[60] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixedpoints. In *Proceedings of POPL*, 238–252, ACM Press,1977.

[61] V. Danos and J. Krivine. Reversible Communicating System. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 3170:292–307, Springer-Verlag, 2004.

[62] V. Danos and J. Krivine. Transactions in RCCS. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 3653:398–412, Springer-Verlag, 2005.

[63] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Cluster. In *Proceedings of OSDI*, 137–150, 2004.

[64] Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.

[65] R. De Nicola and M.C.B. Hennessy. Testing Equivalence for Processes. *Theoretical Computer Science*, 34:83–133, Elsevier, 1984.

[66] Y. Diao, P. Fisher and M.J. Franklin. Yfilter: efficient and scalable filtering of XML documents. In *Proceedings of 18th ICDE*, IEEE Computer Society Press, 2002.

[67] K. Donnelly and M. Fluet. Transactional events. In *Proceedings of ICFP*, 124–135, ACM Press, 2006.

[68] A. Ferrara. Web services: a process algebra approach. In *Proceedings of ICSOC*, 242–251, ACM Press, 2004.

[69] W. Ferreira, M. Hennessy and A.S. Jeffrey. A theory of weak bisimulation for core CML. *Journal of Functional Programming*, 8(5):447–491, 1998.

[70] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD. Dissertation. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`, 2000.

[71] H. Foster, S. Uchitel, J. Magee and J. Kramer. Model-based Verification of Web Service Compositions. In *Proceedings of ASE*, 152–163, IEEE Computer Society Press, 2003.

[72] C. Fournet and G. Gouthier. The Reflexive Chemical Abstract Machine and the Join Calculus. In *Proceedings of POPL*, ACM press, 372–385, 1996.

[73] A. Frisch and K. Nakano. Streaming XML transformations using term rewriting. In *Proceedings of PLAN-X*, 2007.

[74] X. Fu, T. Bultan and J. Su. Analysis of Interacting BPEL Web Services. In *Proceedings of WWW*, 621–630, ACM Press, 2004.

[75] P. Gardner and S. Maffeis. Modelling Dynamic Web Data. *Theoretical Computer Science*, 342(1):104–131, Elsevier, 2005.

[76] S.J. Gay and M. Hole. Types and Subtypes for Client-Server Interactions. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 1576:74–90, Springer-Verlag, 1999.

[77] S.J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[78] A.D. Gordon and P.D. Hankin. A concurrent object calculus: reduction and typing. In *Proceedings of HLCL*. Electronic Notes Theoretical Computer Science, 16(3), 1998.

[79] R. Gorrieri, S. Marchetti and U. Montanari. A^2CCS: Atomic Actions for CCS. *Theoretical Computer Science*, 72(2-3):203–223, Elsevier, 1990.

[80] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates In *Proceedings of SIGMOD*, 419–430, ACM Press, 2003.

[81] T. Harris, S. Marlow, S.P. Jones and M. Herlihy. Composable Memory Transactions. In *Proceedings of PPOPP*, 48–60, *ACM* Press, 2005.

[82] M. Hennessy, J. Rathke and N. Yoshida. safeDpi: a language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

[83] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173(1):82–120, 2002.

[84] M. Herlihy and J.E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of ISCA*, 289–300, 1993.

[85] P. Helland. Autonomous computing: Fiefdoms and Emissaries. Microsoft Webcast, `http://download.microsoft.com/documents/uk/msdn/architecture/connected/Fiefdoms_Emissaries.ppt`, 2002.

[86] K. Honda, V. T. Vasconcelos and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 1381:122–138, Springer-Verlag, 1998.

[87] H. Hosoya, J. Vouillon and B.J. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2004.

[88] H. Hosoya and B.J. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2003.

[89] H. Hosoya and B.J. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transaction on Internet Technology*, 3(2):117–148, 2003.

[90] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGOMOD Record*, 34(2):86–95, 2005.

[91] IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA. Web Services Transactions specifications. Available at `http://www-128.ibm.com/developerworks/library/specification/ws-tx/`, 2004.

[92] A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, Elsevier, 2004.

[93] S. Jagannathan and J. Vitek. Optimistic Concurrency Semantics for Transactions in Coordination Languages. In *Proceedings of COORDINATION*, *Lecture Notes in Computer Science*, 2949:183–198, Springer-Verlag, 2004.

[94] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[95] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[96] N. Kobayashi. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informartica*, 42(4-5):291–347, 2005. cta Inf.

[97] N. Kobayashi. A New Type System for deadlock-Free Processes. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 4137:233–247, Springer-Verlag, 2006.

[98] N. Kobayashi. The `TyPiCal` tool, available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.

[99] N. Kobayashi, B.C. Pierce and D.N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[100] N. Kobayashi, S. Saito and E. Sumii. Implicitly-Typed Deadlock-Free Process Calculus. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 1877:499–503, Springer-Verlag, 2000.

[101] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proceedings of FoSSaCS*, *Lecture Notes in Computer Science*, 3441:282–298, Springer-Verlag, 2005.

[102] A. Lapadula, R. Pugliese and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 4421:33–47, Springer-Verlag, 2007.

[103] A. Lapadula, R. Pugliese and F. Tiezzi. A WSDL-Based Type System for WS-BPEL. In *Proceedings of COORDINATION*, *Lecture Notes in Computer Science*, 4038:145–163, Springer-Verlag, 2006.

[104] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. To appear in Journal of Logic and Algebraic Programming (JLAP), Elsevier press.

[105] B. Ludäscher, P. Mukhopadhyay and Y. Papakonstantinou. A Tranducer-Based XML Query Processor. In *Proceedings of VLDB*, 227–238, Morgan Kaufmann, 2002.

[106] M. Mecella, F. Parisi-Presicce and B. Pernici. Modeling E -service Orchestration through Petri Nets. In *Proceedings of TES*, *Lecture Notes in Computer Science*, 2444:38–47, Springer-Verlag, 2002.

[107] M. Merro. Locality and polyadicity in asynchronous name-passing calculi. In *Proceedings of FoSSaCS*, *Lecture Notes in Computer Science*, 1784:238–251. Springer-Verlag, 2000.

[108] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of ICALP*, *Lecture Notes in Computer Science*, 1443:856-867, Springer-Verlag, 1998. Full version in *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[109] R. Milner. The polyadic π-calculus: a tutorial. *Logic and Algebra of Specification*, 203–246, Springer-Verlag, 1993.

[110] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, Elsevier, 1983.

[111] R. Milner, J. Parrow and D. Walker. A calculus of Mobile Processes, part I and II. *Information and Computation*, 100:1–40 and 41–78, 1992.

[112] K. Nakano. An implementation scheme for XML transformation languages through derivation stream processors. In *Proceedings of ASIAN*, *Lecture Notes in Computer Science*, 3302:74–90, Springer-Verlag, 2004.

[113] K. Nakano. Streamlining functional XML processing. In *Proceedings of 1st DIKU-IST Joint Workshop on Foundations of Software*, 2005.

[114] K. Nakano and S-C. Mu. A Pushdown Machine for Recursive XML Processing. In *Proceedings of APLAS*, *Lecture Notes in Computer Science*, 4279:340–356, Springer-Verlag, 2006.

[115] D. Olteanu, T. Furche and F. Bry. An efficient single-pass query evaluator for XML data structure. In *Proceedings of SAC*, 627–631, ACM Press, 2004.

[116] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[117] B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Process. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[118] J. Ponge. A New Model For Web Services Timed Business Protocols. In *Proceedings of Atelier "Conception des systémes d'information et services Web"*, 2006.

[119] G. Pu, X. Zhao, S. Wang and Z. Qiu. Towards the Semantics and Verification of BPEL4WS. *Electronic Notes Theoretical Computer Science*, 151(2):33–52, Elsevier, 2006.

[120] Z. Qiu, S. Wang, G. Pu and X. Zhao. Semantics of BPEL4WS-Like Fault and Compensation Handling. In *Proceedings of FM*, *Lecture Notes in Computer Science*, 3582:350–365, Springer-Verlag, 2005.

[121] S.K. Rajamani and J. Rehof. A Behavioral Module System for the Pi-Calculus. In *Proceedings of SAS*, *Lecture Notes in Computer Science*, 1877:375–394, 2001.

[122] Relax-NG Technical Committee. Relax-NG. Available at `http://relaxng.org/`.

[123] A. Sahuguet. *ubQL: A Distributed Query Language to Program Distributed Query Systems*. PhD thesis, University of Pennsylvania, 2002.

[124] D. Sangiorgi. Bisimulation in higher-order calculi. *Information and Computation*, 131(2):141–178, 1996.

[125] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2):457–493, Elsevier, 1999.

[126] D. Sangiorgi and R. Milner. Barbed bisimulation. In *Proceedings of ICALP*, *Lecture Notes in Computer Science*, 623:685–695, Springer-Verlag, 1992.

[127] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

[128] K. Takeuchi, K. Honda and M. Kubo. An Interaction-based Language and its Typing System. In *Proceedings of PARLE*, *Lecture Notes in Computer Science*, 817:398–413, Springer-Verlag, 1994.

[129] S. Thatte. XLang: Web Services for Business Process, Microsoft Corporation, 2001.

[130] V.T. Vasconcelos, A. Ravara and S.J. Gay. Session Types for Functional Multithreading. In *Proceedings of CONCUR*, *Lecture Notes in Computer Science*, 3170:497–511, Springer-Verlag, 2004.

[131] J. Vitek, S. Jagannathan, A. Welc and A.L. Hosking. A semantic Framework for designer transactions. In *Proceedings of ESOP*, *Lecture Notes in Computer Science*, 2986:249–263, Springer-Verlag, 2004.

[132] World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Proposed Recommendation 21 November 2006. Available from `http://www.w3.org/TR/2006/PR-xquery-20061121/`.

[133] World Wide Web Consortium. XML Path Language (XPath). W3C Recommendation 16 November 1999. Available from `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[134] N. Yoshida. Graph Types for Monadic Mobile Processes. In *Proceedings of 16th FST/TCS*, *Lecture Notes in Computer Science*, 1180:371–386, Springer-Verlag, 1996.

[135] N. Yoshida. Type-Based Liveness in the Presence of Nontermination and Nondeterminism. MCS Technical Report, 2002-20, University of Leicester, 2002.

[136] N. Yoshida, M. Berger and K. Honda. Strong Normalisation in the π-calculus. *Information and Computation*, 191(2):145–202, 2004.

[137] W3C XML schema Working Group. XML Schema 1.1 Part 2: Datatypes. W3C Working Draft 17 February2 006. Available at `http://www.w3.org/TR/xmlschema11-2/`. XML Schema 1.1 Part 1: Structures. W3C Working Draft 31 August 2006. Available at `http://www.w3.org/TR/xmlschema11-1/`.

[138] W3C XML protocol Working Group. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation 16 August 2006, edited in place 29 September 2006. Available at `http://www.w3.org/TR/xml/`.

Appendix

A

Proofs of Chapter 5

A.1 Proof of Theorem 5.1

Proof of the subject reduction theorem for the type system $\vdash_1$ is quite standard; as usual a preliminary result on substitution is needed. In what follows, recall that variables cannot appear in input subject position.

Proposition A.1.1 (substitution). *Suppose* $\Gamma; \Delta, x \vdash_1 P$, $x, b : \mathcal{T}$ *and* $b \notin \Gamma$ *then*

(1) $b \notin \Delta$ *imply* $\Gamma; \Delta, b \vdash_1 P[b/x]$;

(2) $b \in \Delta$ *and* b *is either* ω*-receptive or inert name imply* $\Gamma; \Delta \vdash_1 P[b/x]$.

Proof: In both cases the proof proceeds by induction on the derivation of $\Gamma; \Delta, x \vdash_1 P$.

(1) Consider the last typing rule applied in the derivation. The interesting case is (T-Out), in the other cases the proof proceeds by applying the inductive hypothesis. Concerning rule (T-Par), $b \notin \Gamma \cup \Delta$ in the premis ensures that acyclicity of the graph and disjointness of Δ_i^ρ, for $i = 1, \ldots, n$, are preserved.

(T-Out) $\emptyset; \Delta, x \vdash_1 \overline{a}\langle c \rangle$ implies $a : \mathsf{S}^{\mathsf{U}}$, $c : \mathsf{S}$ and $(\Delta, x)^\rho \ominus \{a, c\} = \emptyset$. We distinguish the following cases:

$a, c \neq x$**:** $(\Delta, b)^\rho \ominus \{a, c\} = \emptyset$;

$a = x$**:** $\overline{x}\langle c \rangle[b/x] = \overline{b}\langle c \rangle$, $\mathsf{T} = \mathsf{S}^{\mathsf{U}}$ and $(\Delta, b)^\rho \ominus \{b, c\} = \emptyset$;

$c = x$**:** $\overline{a}\langle x \rangle[b/x] = \overline{a}\langle b \rangle$, $\mathsf{T} = \mathsf{S}$ and $(\Delta, b)^\rho \ominus \{a, b\} = \emptyset$;

in each case, by (T-Out), $\emptyset; \Delta, b \vdash_1 \overline{a}\langle c \rangle[b/x]$.

(2) The result follows by a straightforward induction on typing rules. Recall that rule (T-Par) does not impose linearity on the usage of ω-receptive and inert names in output.

□

The following lemma ensures that structural congruent processes have the same behavior. Note that the presence of (T-Str) spares us from introducing a subject congruence proposition.

Lemma A.1.1. *If $P \equiv R$ is deduced without applying alpha-equivalence and $P \xrightarrow{\alpha} P'$ then $R \xrightarrow{\alpha} R'$ and $P' \equiv R'$.*

Proof: The proof is straightforward by induction on the derivation of $P \equiv R$. □

The following proposition makes a step forward in proving the subject reduction theorem.

Proposition A.1.2. *Suppose $\Gamma; \Delta \vdash_1 P$; then*

(1) *$P \xrightarrow{a(b)} P'$, $a : \mathsf{T}^{\mathsf{U}}$ and*

   (a) if $b : \mathsf{T}$ and $b \notin \Delta$ then $\Gamma \ominus \{a\}; \Delta, b \vdash_1 P'$;

   (b) if $b : \mathsf{T}$ with either $\mathsf{T} = \mathsf{S}^{[\omega,k]}$ or $\mathsf{T} = \mathsf{I}$ and $b \in \Delta$ then $\Gamma \ominus \{a\}; \Delta \vdash_1 P'$;

(2) *$P \xrightarrow{\overline{a}\langle b\rangle} P'$ implies $\Gamma; \Delta \ominus \{a, b\} \vdash_1 P'$;*

(3) *$P \xrightarrow{\overline{a}(b)} P'$ implies either $\Gamma, b; (\Delta, b) \ominus \{a, b\} \vdash_1 P'$ if $b : \mathsf{T}^{\mathsf{U}}$ or $\Gamma; (\Delta, b) \ominus \{a, b\} \vdash_1 P'$ if $b : \mathsf{I}$.*

Proof:

(1) By induction on the derivation of $P \xrightarrow{a(b)} P'$; the proof proceeds by distinguishing the last transition rule applied:

   (in): $a(x).P \xrightarrow{a(b)} P[b/x]$. $a; \Delta \vdash_1 a(x).P$ implies, by (T-Inp), $\emptyset; \Delta, x \vdash_1 P$ and

      (a): if $b \notin \Delta$, by Proposition A.1.1 (1) (substitution), it follows that $\emptyset; \Delta, b \vdash_1 P[b/x]$;

      (b): otherwise, if $b : \mathsf{T}$ with either $\mathsf{T} = \mathsf{S}^{[\omega,k]}$ or $\mathsf{T} = \mathsf{I}$ and $b \in \Delta$, by Proposition A.1.1 (2) (substitution), it follows that $\emptyset; \Delta \vdash_1 P[b/x]$.

   (rep): $!a(x).P \xrightarrow{a(b)} !a(x).P|P[b/x]$. $\Gamma; \Delta \vdash_1 !a(x).P$, with $\Gamma = \{a\}$, implies, by rule (T-Rep), $\Delta^\rho = \emptyset$, $a : \mathsf{T}^{[\omega,k]}$, $x : \mathsf{T}$ and $\emptyset; \Delta, x \vdash_1 P$. As previously seen, Proposition A.1.1 (1,2) (substitution) can be applied (depending on $b \in \Delta$ or not) for deducing $\emptyset; \Delta' \vdash_1 P[b/x]$, with either $\Delta' = \Delta, b$ or $\Delta' = \Delta$. $\Gamma \ominus \{a\} = \Gamma$, hence $\Gamma \ominus \{a\}; \Delta \vdash_1 !a(x).P$; finally, by either (T-Par) (if P is in normal-form) or (T-Str), (T-Res), (T-Res-i), (T-Res-$\mathbf{L}$) and (T-Par) (otherwise), $\Gamma \ominus \{a\}; \Delta' \vdash_1 !a(x).P|P[b/x]$, because $\Gamma^\rho = \Delta^\rho = \emptyset$;

   (alpha): the proof proceeds by applying the inductive hypothesis;

   (par_1): $P|R \xrightarrow{a(b)} P'|R$ implies $P \xrightarrow{a(b)} P'$. We distinguish two cases:

- if $P \mid R$ is not in normal-form rule (T-STR) is applied on the last step of the normal derivation of $\Gamma; \Delta \vdash_1 P \mid R$. The proof proceeds by applying Lemma A.1.1, (ALPHA) and the inductive hypothesis.
- If $P \mid R$ is in normal-form rule (T-PAR) is applied for deducing $\Gamma; \Delta \vdash_1 P \mid R$. Suppose for simplicity that P and R are both prime, in the general case $P = P_1 | \ldots | P_n$ and $R = R_1 | \ldots | R_m$, with each P_i and R_j prime, the proof proceeds similarly.
  By (T-PAR), $\Gamma = \Gamma_P \cup \Gamma_R$, $\Delta = \Delta_P \cup \Delta_R$, $\Gamma_P; \Delta_P \vdash_1 P$ and $\Gamma_R; \Delta_R \vdash_1 R$. Moreover, the dependency graph is acyclic and the sets $\Gamma_P^\rho, \Gamma_R^\rho$ and $\Delta_P^\rho, \Delta_R^\rho$ are disjoint. By inductive hypothesis, $P \xrightarrow{a(b)} P'$ implies that $\Gamma_P \ominus \{a\}; \Delta'_P \vdash_1 P'$ with either $\Delta'_P = \Delta_P, b$ or $\Delta'_P = \Delta_P$.
  Suppose P' prime. In this case, $\Gamma \ominus \{a\}; \Delta' \vdash_1 P' \mid R$, with either $\Delta' = \Delta, b$ or $\Delta' = \Delta$, in fact in both cases (1a) and (1b) acyclicity of the graph is preserved: nested free inputs are not allowed and no new arcs from b can be added to the graph. Moreover, disjointness of ${\Delta'_P}^\rho$, ${\Delta_R}^\rho$ and ${\Gamma'_P}^\rho, {\Gamma_R}^\rho$ is guaranteed (if b is responsive then $b \notin \Delta = \Delta_R \cup \Delta_P$). If P' is not prime, rules (T-STR), (T-RES), (T-RES-I), (T-RES-$\mathbb{1}$) and (T-PAR) are applied for achieving the same result.

(RES): $(\nu\, c)P \xrightarrow{a(b)} (\nu\, c)P'$ implies $P \xrightarrow{a(b)} P'$ with $a, b \neq c$. Suppose $c : \mathsf{T}^{\mathsf{U}}$ (in the other cases the proof proceeds similarly). By (T-RES), $\Gamma, c; \Delta, c \vdash_1 P$ and by induction hypothesis, $P \xrightarrow{a(b)} P'$ implies $(\Gamma, c) \ominus \{a\}; \Delta', c \vdash_1 P'$, with either $\Delta' = \Delta, b$ or $\Delta' = \Delta$. $\Gamma \ominus \{a\}; \Delta' \vdash_1 (\nu\, c)P'$ follows by (T-RES) and $c \neq a$.

(2) By induction on the derivation of $P \xrightarrow{\overline{a}\langle b\rangle} P'$, the proof proceeds by distinguishing the last transition rule applied:

(OUT): $\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$. $\emptyset; \Delta \vdash_1 \overline{a}\langle b\rangle$ implies, by rule (T-OUT), $(\Delta \ominus \{a, b\})^\rho = \emptyset$ and $\emptyset; \Delta \ominus \{a, b\} \vdash_1 \mathbf{0}$ by rule (T-NIL);

(ALPHA): the proof proceeds by applying the inductive hypothesis by applying rule (T-STR);

(PAR$_1$): $P|R \xrightarrow{\overline{a}\langle b\rangle} P'|R$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$. The proof proceeds as already seen for (1), by distinguishing the last rule applied in the normal derivation of $\Gamma; \Delta \vdash_1 P \mid R$. Note that in case (T-PAR), acyclicity of the dependency graph and disjointness of input and output contexts are preserved because the new contexts are obtained by subtracting both a and b (if responsive);

(RES): $(\nu\, c)P \xrightarrow{\overline{a}\langle b\rangle} (\nu\, c)P'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ with $a, b \neq c$. Suppose $c : \mathsf{T}^{\mathsf{U}}$ (in the other cases the proof proceeds similarly). In the normal derivation of $\Gamma; \Delta \vdash_1 (\nu\, c)P$ rule (T-RES) is the last applied with premise $\Gamma, c; \Delta, c \vdash_1 P$, hence by applying the inductive hypothesis, $P \xrightarrow{\overline{a}\langle b\rangle} P'$ implies $\Gamma, c; (\Delta, c) \ominus \{a, b\} \vdash_1 P'$. $\Gamma; \Delta \ominus \{a, b\} \vdash_1 (\nu\, c)P'$ follows by (T-RES) and $c \neq a, b$.

(3) By induction on the derivation of $P \xrightarrow{\overline{a}(b)} P'$, the proof proceeds by distinguishing the last transition rule applied. The interesting case is (OPEN), in the other cases the proof proceeds by induction as already seen for (2).
$(\nu\, b)P \xrightarrow{\overline{a}(b)} P'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and $a \neq b$. Consider the normal derivation of $\Gamma; \Delta \vdash_1 (\nu\, b)P$ and suppose rule (T-RES) is applied in the last step of the derivation (the case (T-RES-I) can be proved similarly and it cannot be the case that (T-RES-T) is applied, because $b \in \mathrm{on}(P)$.) Hence, $\Gamma, b; \Delta, b \vdash_1 P$ and by inductive hypothesis, $P \xrightarrow{\overline{a}\langle b\rangle} P'$ implies $\Gamma, b; (\Delta, b) \ominus \{a, b\} \vdash_1 P'$.

□

Theorem A.1.1 (Theorem 5.1). *Suppose* $\Gamma; \Delta \vdash_1 P$ *and* $P \xrightarrow{[a]} P'$. *Then* $\Gamma \ominus \{a\}; \Delta \ominus \{a\} \vdash_1 P'$.

PROOF: By induction on the derivation of $P \xrightarrow{[a]} P'$; we consider the last reduction rule applied:

(ALPHA): the proof proceeds by applying the inductive hypothesis by applying rule (T-STR);

(COM$_1$): $P|R \xrightarrow{\tau\langle a,b\rangle} P'|R'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and $R \xrightarrow{a(b)} R'$. We distinguish two cases, depending on the last rule applied in the normal derivation of $\Gamma; \Delta \vdash_1 P|R$:

(T-STR): the proof proceeds by applying the inductive hypothesis and relies on Lemma A.1.1 and (ALPHA);

(T-PAR): suppose, for simplicity, that P and R are prime. In the most general case where both are the parallel composition of prime sub-processes the proof proceeds similarly. $\Gamma; \Delta \vdash_1 P|R$ implies that $\Gamma = \Gamma_P \cup \Gamma_R$, $\Delta = \Delta_P \cup \Delta_R$, $\Gamma_P; \Delta_P \vdash_1 P$ and $\Gamma_R; \Delta_R \vdash_1 R$. Moreover $\Gamma_P^\rho \cap \Gamma_R^\rho = \Delta_P^\rho \cap \Delta_R^\rho = \emptyset$ and the dependency graph is acyclic. By Proposition A.1.2 (2,1), $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and $R \xrightarrow{a(b)} R'$ imply that $\Gamma_P; \Delta_P \ominus \{a, b\} \vdash_1 P'$ and $\Gamma_R \ominus \{a\}; \Delta'_R \vdash_1 R'$, with either $\Delta'_R = \Delta_R, b$, if $b \notin \Delta_R$, or $\Delta'_R = \Delta_R$ and b is either a ω-receptive or a inert name.

Suppose P' and R' are both prime. It is easy to see that the premises of (T-PAR) are still satisfied: the graph is acyclic because nested inputs are

not allowed hence no new arcs from b can be added to the graph; and if b is responsive $b \notin \Delta_R$ and $b \notin (\Delta_P \ominus \{a,b\})$. Hence $\Gamma \ominus \{a\}; \Delta \ominus \{a\} \vdash_1 P' \mid R'$. If P' and R' are not prime, rules (T-Str), (T-Res), (T-Res-i), (T-Res-$\mathbf{\bot}$) and (T-Par) can be applied for achieving the same result;

(par$_1$): $P|R \xrightarrow{[a]} P'|R$ implies $P \xrightarrow{[a]} P'$. We distinguish two cases, depending on the last typing rules applied in the normal derivation of $\Gamma; \Delta \vdash_1 P|R$:

(T-Str): the proof proceeds by applying the inductive hypothesis and relies on Lemma A.1.1 and (alpha);

(T-Par): the proof proceeds by applying the inductive hypothesis and either (T-Par), if P' and R are prime, or (T-Str), (T-Res), (T-Res-i), (T-Res-$\mathbf{\bot}$) and (T-Par), otherwise;

(close$_1$): $P|R \xrightarrow{\tau\langle a,b\rangle} (\nu\, b)(P'|R')$ implies $P \xrightarrow{\overline{a}(b)} P'$, $R \xrightarrow{a(b)} R'$ and $b \notin \mathrm{fn}(R)$. (T-Str) is the last rule applied in the normal derivation of $\Gamma; \Delta \vdash_1 P|R$. Process P is not prime (it contains a restriction on b), hence there exists S in normal-form such that: $P|R \equiv S$, $\Gamma; \Delta \vdash_1 S$ and $S = (\nu\, b)(\nu\, \widetilde{d}_P)(\nu\, \widetilde{d}_R)(P_1 \mid R_1)$, with P_1 and R_1 parallel compositions of prime processes, such that $(\nu\, b)(\nu\, \widetilde{d}_P)P_1 \equiv P$ and $(\nu\, \widetilde{d}_R)R_1 \equiv R$. The result follows by observing that $(\nu\, b)(\nu\, \widetilde{d}_P)P_1 \xrightarrow{\overline{a}(b)} (\nu\, \widetilde{d}_P)P_1'$ and $(\nu\, \widetilde{d}_P)P_1' \equiv P'$ (Lemma A.1.1), $(\nu\, \widetilde{d}_R)R_1 \xrightarrow{a(b)} (\nu\, \widetilde{d}_R)R_1'$ and $(\nu\, \widetilde{d}_R)R_1' \equiv R'$ (Lemma A.1.1), $S' = (\nu\, b)(\nu\, \widetilde{d}_P)(\nu\, \widetilde{d}_R)(R_1' \mid P_1')$ (close$_1$), $S' \equiv (\nu\, b)(P' \mid R')$ and by applying Proposition A.1.2 (1a,3) and rules (T-Par), (T-Str), (T-Res), (T-Res-i) and (T-Res-t);

(res): $(\nu\, a)P \xrightarrow{\tau\langle b,c\rangle} (\nu\, a)P'$ implies $P \xrightarrow{\tau\langle b,c\rangle} P'$ with either $a = b$ and a ω-receptive or $a \neq b$. In the normal derivation of $\Gamma; \Delta \vdash_1 (\nu\, a)P$ the premise of the last step is either $\Gamma; \Delta, a \vdash_1 P$, if $a : \mathsf{I}$ (by (T-Res-i)), or $\Gamma, a; \Delta, a \vdash_1 P$, if $a : \mathsf{T}^{\mathsf{U}}$ (by (T-Res)).

We distinguish the following cases:

$\mathsf{U} = [\omega, k]$ and $a = b$: by induction $P \xrightarrow{\tau\langle a,c\rangle} P'$ implies $(\Gamma, a) \ominus \{a\}; (\Delta, a) \ominus \{a\} \vdash_1 P'$, but a is ω-receptive, hence $(\Gamma, a) \ominus \{a\} = \Gamma$ and $(\Delta, a) \ominus \{a\} = \Delta$; by (T-Res), $\Gamma; \Delta \vdash_1 (\nu\, a)P'$;

otherwise: suppose $a \neq b$ and $a : \mathsf{T}^{\mathsf{U}}$ (the cases $a : \mathsf{I}$ and $a : \mathbf{\bot}$ are similar). By induction $(\Gamma, a) \ominus \{b\}; (\Delta, a) \ominus \{b\} \vdash_1 P'$; $b \neq a$, thus $(\Gamma, a) \ominus \{b\} = \Gamma \ominus \{b\}, a$ and $(\Delta, a) \ominus \{b\} = \Delta \ominus \{b\}, a$. Hence, by rule (T-Res), $\Gamma \ominus \{b\}; \Delta \ominus \{b\} \vdash_1 (\nu\, a)P'$;

(RES-ρ): $(\nu\, a)P \xrightarrow{\tau\langle a,b\rangle} (\nu\, c)P'[c/a]$, with $c : \mathbf{1}$, implies $P \xrightarrow{\tau\langle a,b\rangle} P'$ and $a : \mathsf{T}^{[\rho,k]}$. $\Gamma;\Delta \vdash_1 (\nu\, a)P$ implies, by rule (T-RES), $\Gamma, a;\Delta, a \vdash_1 P$; by induction $(\Gamma, a) \ominus \{a\};(\Delta, a) \ominus \{a\} \vdash_1 P'$ that is $\Gamma;\Delta \vdash_1 P'$ with $a \notin \mathrm{fn}(P')$. Moreover, c fresh implies $c \notin \mathrm{fn}(P'[c/a])$ and by (T-RES-$\mathbf{1}$) and $c : \mathbf{1}$ we have $\Gamma;\Delta \vdash_1 (\nu\, c)P'[c/a]$. □

A.2 Proof of Theorem 5.2

In this section we prove the intermediary results needed for proving Theorem 5.2 (responsiveness).

Lemma A.2.1 (Lemma 5.2). *Suppose* $\Gamma;\Delta \vdash_1 P$*, then there exists a normal derivation of* $\Gamma;\Delta \vdash_1 P$*.*

PROOF: The proof is straightforward by induction on the length of the derivation of $\Gamma;\Delta \vdash_1 P$. We distinguish the last typing rule applied.

The base cases are (T-OUT) and (T-NIL). Rules (T-INP) and (T-REP) relies on the inductive hypothesis for deriving a normal derivation of the well-typedness of the continuation processes. Similar comments for the typing rules for restriction and for (T-PAR).

The more involved case is when (T-STR) is the last rule applied. Suppose the premises are $P \equiv R$ and $\Gamma;\Delta \vdash_1 R$, and the conclusion is $\Gamma;\Delta \vdash_1 P$.

If P is not in normal-form and R is, then, by inductive hypothesis, well-typedness of R can be deduced by using a normal derivation and by further applying rule (T-STR) we obtain a normal derivation of $\Gamma;\Delta \vdash_1 P$.

Suppose R is not in normal-form and P is, then, rule (T-STR) has been applied for deducing $\Gamma;\Delta \vdash_1 R$; suppose the process in the premise of this step is R'. $R' \equiv R$ and $R \equiv P$ implies that $P \equiv R'$. Hence, if we consider the initial derivation of $\Gamma;\Delta \vdash_1 P$ and we substitute the last two steps (which are applications of (T-STR)) with a single application with premise $P \equiv R'$, we obtain a derivation of $\Gamma;\Delta \vdash_1 P$ smaller than the initial one, hence by inductive hypothesis there exists a normal derivation of $\Gamma;\Delta \vdash_1 P$.

Finally, suppose both P and R are in normal-form, with $P \neq R$. By inductive hypothesis a normal derivation of $\Gamma;\Delta \vdash_1 R$ exists. Suppose $P \equiv R$ is deduced by applying a single structural rule (hence without applying the transitivity of $\equiv$); we distinguish the following cases (it is easy to generalize to the case where $\equiv$ have been applied $n > 1$ times):

- if P is obtained from R by applying either commutativity/associativity of parallel composition or by commutating restrictions then it is easy to obtain a normal

derivation for $\Gamma; \Delta \vdash_1 P$ from the normal derivation of $\Gamma; \Delta \vdash_1 R$;

- scope extrusion cannot be applied because both P and R are in normal form;
- if alpha-renaming is applied, then we can obtain a normal derivation of $\Gamma; \Delta \vdash_1 P$ by considering the "alpha-renaming of the normal derivation" of $\Gamma; \Delta \vdash_1 R$;
- if $P = (\nu\, a)R$, with either $a : \mathsf{I}$ or $a : \mathbf{L}$, then a normal derivation for $\Gamma; \Delta \vdash_1 P$ can be obtained by the normal derivation of $\Gamma; \Delta \vdash_1 R$ followed by the application of either (T-Res-I) or (T-Res-$\mathbf{L}$) instead of (T-Str);
- if $R = (\nu\, a)P$, with either $a : \mathsf{I}$ or $a : \mathbf{L}$, then a normal derivation of $\Gamma; \Delta \vdash_1 P$ is obtained from the normal derivation of $\Gamma; \Delta \vdash_1 R$ by removing the last step (that is the step used for bounding a in P).

□

Proposition A.2.1 (Proposition 5.1). *Suppose that $\Gamma; \Delta \vdash_1 P$, with Γ, Δ and P satisfying the conditions in the premise of rule* (T-Par) *and $\Gamma^\rho = \Delta^\rho$. Then for some j in $1, \ldots, n$ we have $P_j = \overline{a}\langle b\rangle$ with either a or b responsive.*

Proof: $P = P_1 \mid \cdots \mid P_n$, for each i process P_i is prime, $\Gamma_i; \Delta_i \vdash_1 P_i$, $\Gamma_i^\rho \cap \Gamma_j^\rho = \emptyset$ and $\Delta_i^\rho \cap \Delta_j^\rho = \emptyset$ for $i \neq j$. Moreover, $\Gamma = \bigcup_{i=1\ldots n} \Gamma_i$, $\Delta = \bigcup_{i=1\ldots n} \Delta_i$; and $\mathrm{DG}(\Gamma_i^\rho, \Delta_i^\rho)_{i=1,\ldots,n}$ is acyclic.

The acyclicity of the graph implies that there is at least one node c with no outgoing arcs. By construction of the graph and $\Gamma^\rho = \Delta^\rho$ we have that $\exists j \in 1, \ldots, n$ s.t. $c \in \Delta_j^\rho$ and $\Gamma_j^\rho = \emptyset$. Consider the process P_j. By hypothesis P_j is prime and $\Gamma_j; \Delta_j \vdash_1 P_j$.

By contradiction, assume $P_j = !a(b).R$. $\Gamma_j; \Delta_j \vdash_1 !a(b).R$ and rule (T-Rep) imply ${\Delta_j}^\rho = \emptyset$, but this is in contradiction with the hypothesis $c \in \Delta_j^\rho$, thus $P_j \neq !a(b).R$.

Again by contradiction, assume $P_j = a(b).P$. $\Gamma_j; \Delta_j \vdash_1 a(b).P$ and rule (T-Inp) imply $\Gamma_j^\rho = \{a\}$, but this is in contradiction with the hypothesis $\Gamma_j^\rho = \emptyset$, thus $P_j \neq a(b).P$.

In conclusion, P_j prime implies that $P_j = \overline{a}\langle b\rangle$ with either $a = c$ or $b = c$, thus at least one of the two names is responsive. □

Substitutions preserve $\mathrm{wt}(\cdot)$:

Lemma A.2.2. *Suppose $\Gamma; \Delta, x \vdash_1 P$ and $x, b : \mathsf{T}$. Then $\mathrm{wt}(P) = \mathrm{wt}(P[b/x])$.*

Proof: The proof is straightforward by induction on the definition of $\mathrm{wt}(\cdot)$ (note that $x, b : \mathsf{T}$ implies that $\mathrm{lev}(x) = \mathrm{lev}(b)$). □

The following proposition ensures that the weight of a process (Table 5.7) is a good measure when considering responsive reductions, in fact it decreases after each communication involving responsive names. This thanks to the constraints on levels in the premises of rule (T-Rep) and to the linearity of responsive names. The lemma

below is useful for proving this result.

Lemma A.2.3. *Suppose* $\Gamma;\Delta \vdash_1 P$*, then:*

(1) *if* $a \in \Gamma$*,* $a : \mathsf{T}^{U}$ *and* $b : \mathsf{T}$ *then* $P \xrightarrow{a(b)} P'$ *and, if either* a *or* b *is responsive,* $\mathrm{wt}(P') \prec \mathrm{wt}(P) + 0_{\mathrm{lev}(a)}$*;*

(2) *if* $P \xrightarrow{\overline{a}\langle b\rangle} P'$ *(or* $P \xrightarrow{\overline{a}(b)} P'$*) then* $\mathrm{wt}(P') \preccurlyeq \mathrm{wt}(P) - 0_{\mathrm{lev}(a)}$.

Proof: In both cases the proof proceeds by induction on the derivation of $\Gamma;\Delta \vdash_1 P$.

(1) Consider the last typing rule applied in the derivation; the most interesting cases are rules (T-Inp) and (T-Rep). The other cases can be easily proved by applying the inductive hypothesis.

(T-Inp): Suppose $P = a(x).R$. By rule (in), $a(x).R \xrightarrow{a(b)} R[{}^{b}\!/\!_{x}]$ and by (T-Inp), $x : \mathsf{T}$. $\mathrm{wt}(a(x).R)+0_{\mathrm{lev}(a)} = \mathrm{wt}(R)+0_{\mathrm{lev}(a)} \succ \mathrm{wt}(R[{}^{b}\!/\!_{x}]) = \mathrm{wt}(R)$, by Lemma A.2.2.

(T-Rep): Suppose $P = !a(x).R$. $a;\Delta \vdash_1 !a(x).R$ implies that $\forall c \in \mathrm{os}(R)$: $\mathrm{lev}(c) < \mathrm{lev}(a)$. By rule (rep), $!a(x).R \xrightarrow{a(b)} !a(x).R|R[{}^{b}\!/\!_{x}]$. If b is ω-receptive, there is nothing to prove. Otherwise, from $\forall c \in \mathrm{os}(R)$: $\mathrm{lev}(c) < \mathrm{lev}(a)$, we have $\mathrm{wt}(!a(x).R) + 0_{\mathrm{lev}(a)} = 0_{\mathrm{lev}(a)} \succ \mathrm{wt}(R[{}^{b}\!/\!_{x}]) = \mathrm{wt}(!a(x).R|R[{}^{b}\!/\!_{x}])$;

(2) Consider the last typing rule applied in the derivation; the most interesting cases are rules (T-Out) and (T-Res). The other cases can be easily proved by applying the inductive hypothesis.

(T-Out): Suppose $P = \overline{a}\langle b\rangle$. By (out), $\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$ and $\mathrm{wt}(\mathbf{0}) = \mathrm{wt}(\overline{a}\langle b\rangle) - 0_{\mathrm{lev}(a)}$;

(T-Res): Suppose $P = (\nu\, d)R$ and $d : \mathsf{T}$ (the cases $d : \mathsf{I}$ and $d : \mathbf{1}$ are proved similarly.) By (T-Res), $\Gamma;\Delta \vdash_1 (\nu\, d)R$ implies $\Gamma, d;\Delta, d \vdash_1 R$. We distinguish two cases considering the transition rule applied:

(open): $(\nu\, d)R \xrightarrow{\overline{a}(d)} R'$ implies $R \xrightarrow{\overline{a}\langle d\rangle} R'$ and, by inductive hypothesis, $\mathrm{wt}(R') \preccurlyeq \mathrm{wt}(R) - 0_{\mathrm{lev}(a)} = \mathrm{wt}((\nu\, d)R) - 0_{\mathrm{lev}(a)}$;

(res): $(\nu\, d)R \xrightarrow{\overline{a}\langle b\rangle} (\nu\, d)R'$ implies $R \xrightarrow{\overline{a}\langle b\rangle} R'$, $a, b \neq d$ and, by inductive hypothesis, $\mathrm{wt}(R') \preccurlyeq \mathrm{wt}(R) - 0_{\mathrm{lev}(a)}$. By definition of $\mathrm{wt}(\cdot)$, $\mathrm{wt}((\nu\, d)R') = \mathrm{wt}(R') \preccurlyeq \mathrm{wt}(R) - 0_{\mathrm{lev}(a)} = \mathrm{wt}((\nu\, d)R) - 0_{\mathrm{lev}(a)}$.

□

Proposition A.2.2 (Proposition 5.2). *Suppose* $\Gamma;\Delta \vdash_1 P$ *and* $P \xrightarrow{\tau\langle a,b\rangle} P'$*, with either* a *or* b *responsive. Then* $\mathrm{wt}(P') \prec \mathrm{wt}(P)$.

PROOF: By induction on the derivation of $\Gamma;\Delta \vdash_1 P$, we distinguish the last typing rule applied. The interesting case is rule (T-PAR), the other cases can be easily proved by applying the inductive hypothesis.

$\Gamma;\Delta \vdash_1 P$ implies that $P = P_1|\cdots|P_n$, with P_i prime and $\Gamma_i;\Delta_i \vdash_1 P_i$, for $i = 1,\ldots,n$. $P \xrightarrow{\tau\langle a,b\rangle} P'$ implies (by rule (COM$_{1-2}$)) that we can divide the P_is into two groups called S and R respectively, such that S contains P_j such that $P_j \xrightarrow{\overline{a}\langle b\rangle}$ and R contains P_k such that $P_k \xrightarrow{a(b)}$. Thus, by (PAR$_1$) and (PAR$_2$), $S \xrightarrow{\overline{a}\langle b\rangle} S'$ and $R \xrightarrow{a(b)} R'$. In the same manner we group all contexts Γ_i and Δ_i into the contexts Γ_S, Δ_S, Γ_R and Δ_R such that $\Gamma_S;\Delta_S \vdash_1 S$ and $\Gamma_R;\Delta_R \vdash_1 R$. By Lemma A.2.3 (1,2) we have that $\mathrm{wt}(R') \prec \mathrm{wt}(R) + 0_{\mathrm{lev}(a)}$ and $\mathrm{wt}(S') \preccurlyeq \mathrm{wt}(S) - 0_{\mathrm{lev}(a)}$, that is $\mathrm{wt}(P') = \mathrm{wt}(R') + \mathrm{wt}(S') \prec \mathrm{wt}(S) - 0_{\mathrm{lev}(a)} + \mathrm{wt}(R) + 0_{\mathrm{lev}(a)} = \mathrm{wt}(P)$. □

A.3 Proof of Theorem 5.3

In what follows we introduce some notations and prove some preliminary results useful for giving a detailed proof of Theorem 5.3, which is only sketched in [64].

We denote by $\mathrm{O}(P)$ the set of all output actions of P that are active, that is, not underneath a replication; $\mathrm{O}(P)$ is formally defined as follows

$$\begin{array}{rrr} \mathrm{O}(\mathbf{0}) = \emptyset & \mathrm{O}(a(b).P) = \mathrm{O}(P) & \mathrm{O}(\overline{a}\langle b\rangle) = \{\overline{a}\langle b\rangle\} \\ \mathrm{O}(!a(b).P) = \emptyset & \mathrm{O}((\nu\, a)P) = \mathrm{O}(P) & \mathrm{O}(P|R) = \mathrm{O}(P) \cup \mathrm{O}(R)\ . \end{array}$$

We indicate with $\mathrm{O}^\rho(P)$ the set containing all output actions in $\mathrm{O}(P)$ involving a responsive name. We also write $|\,\mathrm{O}^\rho(P)\,|$ and $|\,\mathrm{O}(P)\,|$ for the size of $\mathrm{O}^\rho(P)$ and $\mathrm{O}(P)$, respectively. The *height* of P, written $\mathrm{h}(P)$, is defined as the greatest size of a replicated term in P. E.g. $\mathrm{h}(!a(x).P) = 1 + |\,P\,|$.

First of all, we prove that weight and height of a process and the number of outputs it contains are preserved by structural congruence. Moreover we prove that the number of outputs in a process P is upper bounded by $|\,P\,|$. After, in Proposition A.3.1, we show that the number of output actions in P and its size may grow only after a reduction where the subject is an ω-receptive name, but this grown is limited by $\mathrm{h}(P)$.

Lemma A.3.1. *If* $P \equiv R$ *then* $\mathrm{wt}(P) = \mathrm{wt}(R)$, $|\,\mathrm{O}^\rho(P)\,| = |\,\mathrm{O}^\rho(R)\,|$ *and* $\mathrm{h}(P) = \mathrm{h}(R)$.

PROOF: The proof is straightforward by induction on the derivation of $P \equiv R$.

□

Lemma A.3.2. *If* $\Gamma;\Delta \vdash_1 P$ *then* $|\,\mathrm{O}^\rho(P)\,| \leqslant |\,P\,|$.

PROOF: By definition of $|\,\mathrm{O}^\rho(P)\,|$. □

Proposition A.3.1. *If* $\Gamma;\Delta \vdash_1 P$ *then:*

(1) *if either* $P \xrightarrow{\overline{a}\langle b\rangle} P'$ *or* $P \xrightarrow{\overline{a}(b)} P'$ *and either* a *or* b *is responsive then* $|\,\mathrm{O}^\rho(P')\,| = |\,\mathrm{O}^\rho(P)\,| - 1$*;*

(2) *if* $P \xrightarrow{a(b)} P'$*, with* a *responsive name, then* $|\,\mathrm{O}^\rho(P')\,| = |\,\mathrm{O}^\rho(P)\,|$*;*

(3) *if* $P \xrightarrow{a(b)} P'$*, with* a *ω-receptive name, then* $|\,\mathrm{O}^\rho(P')\,| \leqslant |\,\mathrm{O}^\rho(P)\,| + \mathrm{h}(P)$*;*

(4) *if* $P \xrightarrow{[a]} P'$*, with* a *responsive name, then* $|\,\mathrm{O}^\rho(P')\,| \leqslant |\,\mathrm{O}^\rho(P)\,| - 1$*;*

(5) *if* $P \xrightarrow{[a]} P'$*, with* a *ω-receptive name carrying responsive names, then* $|\,\mathrm{O}^\rho(P')\,| \leqslant |\,\mathrm{O}^\rho(P)\,| + \mathrm{h}(P) - 1$.

PROOF: In all cases the proof proceeds by induction on the derivation of $P \xrightarrow{\mu} P'$; in each case we distinguish the last transition rule applied:

(1) the interesting case is (OUT); case (ALPHA) relies on Lemma A.3.1 and the other cases ((PAR$_1$), (OPEN) and (RES)) can be proved by applying the inductive hypothesis.
By (OUT), $\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$ with a or b responsive, and $|\,\mathrm{O}^\rho(\overline{a}\langle b\rangle)\,| - 1 = 0 = |\,\mathrm{O}^\rho(\mathbf{0})\,|$;

(2) the interesting case is (IN); case (ALPHA) relies on Lemma A.3.1 and the other cases ((PAR$_1$) and (RES)) can be proved by applying the inductive hypothesis.
By (IN), $a(x).P \xrightarrow{a(b)} P[b/x]$, $\mathrm{O}^\rho(a(x).P) = \mathrm{O}^\rho(P)$, thus $|\,\mathrm{O}^\rho(a(x).P)\,| = |\,\mathrm{O}^\rho(P)\,|$ and $|\,\mathrm{O}^\rho(P)\,| = |\,\mathrm{O}^\rho(P[b/x])\,|$;

(3) the interesting case is (REP); case (ALPHA) relies on Lemma A.3.1 and the other cases ((PAR$_1$) and (RES)) can be proved by applying the inductive hypothesis.
By (REP), $!a(x).P \xrightarrow{a(b)} !a(x).P|P[b/x]$, $\mathrm{O}^\rho(!a(x).P) = \emptyset$, $\mathrm{h}(!a(x).P) = 1 + |\,P\,|$ and $|\,\mathrm{O}^\rho(!a(x).P|P[b/x])\,| = |\,\mathrm{O}^\rho(P[b/x])\,|$. By Lemma A.3.2, $|\,\mathrm{O}^\rho(P)\,| \leqslant |\,P\,|$, hence $|\,\mathrm{O}^\rho(P[b/x])\,| \leqslant \mathrm{h}(!a(x).P)$ and $|\,\mathrm{O}^\rho(!a(x).P|P[b/x])\,| \leqslant |\,\mathrm{O}^\rho(!a(x).P)\,| + \mathrm{h}(!a(x).P)$;

(4) the interesting cases are rules (COM$_1$) and (CLOSE$_1$); case (ALPHA) relies on Lemma A.3.1 and the other cases ((PAR$_1$), (RES) and (RES-ρ)) can be proved by applying the inductive hypothesis.

(COM$_1$): $R|S \xrightarrow{\tau\langle a,b\rangle} R'|S'$, with a responsive name, implies $R \xrightarrow{\overline{a}\langle b\rangle} R'$ and $S \xrightarrow{a(b)} S'$. $\Gamma;\Delta \vdash_1 R|S$ implies, that there are suitable contexts $\Gamma_1, \Delta_1, \Gamma_2$ and Δ_2, such that $\Gamma_1;\Delta_1 \vdash_1 R$ and $\Gamma_2;\Delta_2 \vdash_1 S$.

By Proposition A.3.1 (1,2) we have $|\,\mathrm{O}^\rho(R')\,| = |\,\mathrm{O}^\rho(R)\,| - 1$ and $|\,\mathrm{O}^\rho(S')\,| = |\,\mathrm{O}^\rho(S)\,|$, thus $|\,\mathrm{O}^\rho(P')\,| = |\,\mathrm{O}^\rho(R'|S')\,| = |\,\mathrm{O}^\rho(R')\,| + |\,\mathrm{O}^\rho(S')\,| = |\,\mathrm{O}^\rho(R)\,| - 1 + |\,\mathrm{O}^\rho(S)\,| = |\,\mathrm{O}^\rho(P)\,| - 1$;

(CLOSE$_1$): this case is similar to the previous one;

(5) the interesting cases are rules (COM$_1$) and (CLOSE$_1$); case (ALPHA) relies on Lemma A.3.1 and the other cases ((PAR$_1$) and (RES)) can be proved by applying the inductive hypothesis.

(COM$_1$): $R|S \xrightarrow{\tau\langle a,b\rangle} R'|S'$, with a ω-receptive name, implies $R \xrightarrow{\overline{a}\langle b\rangle} R'$ and $S \xrightarrow{a(b)} S'$. $\Gamma;\Delta \vdash_1 R|S$ implies that there are suitable contexts $\Gamma_1, \Delta_1, \Gamma_2$ and Δ_2, such that $\Gamma_1;\Delta_1 \vdash_1 R$ and $\Gamma_2;\Delta_2 \vdash_1 S$.

By Proposition A.3.1 (1,3) we have $|\,\mathrm{O}^\rho(R')\,| = |\,\mathrm{O}^\rho(R)\,| - 1$ and $|\,\mathrm{O}^\rho(S')\,| \leqslant |\,\mathrm{O}^\rho(S)\,| + \mathrm{h}(S)$, thus $|\,\mathrm{O}^\rho(P')\,| = |\,\mathrm{O}^\rho(R'|S')\,| = |\,\mathrm{O}^\rho(R')\,| + |\,\mathrm{O}^\rho(S')\,| \leqslant |\,\mathrm{O}^\rho(P)\,| - 1 + |\,\mathrm{O}^\rho(S)\,| + \mathrm{h}(S) \leqslant |\,\mathrm{O}^\rho(P)\,| + \mathrm{h}(P) - 1$;

(CLOSE$_1$): this case is similar to the previous one.

□

The height of a process is preserved by transitions:

Proposition A.3.2. *If $P \xrightarrow{\mu} P'$ then $\mathrm{h}(P') = \mathrm{h}(P)$.*

PROOF: The proof is straightforward by induction on the derivation of $P \xrightarrow{\mu} P'$.

□

Each component of the weight vector of a process P gives us the number of active outputs in P of the corresponding level.

Proposition A.3.3. *If $\Gamma;\Delta \vdash_1 P$ and $\mathrm{wt}(P) = \langle w_k, \ldots, w_0\rangle$ then in $\mathrm{O}^\rho(P)$ there are at most w_i output of level i for i in $0, \ldots, k$.*

PROOF: By induction on the structure of P:

$P = \mathbf{0}$, $P = !a(x).P'$: $\mathrm{O}^\rho(P) = \emptyset$ and $\mathrm{wt}(P) = 0$;

$P = a(x).P'$: $\mathrm{wt}(a(x).P') = \mathrm{wt}(P')$ and $\mathrm{O}^\rho(a(x).P') = \mathrm{O}^\rho(P')$. $\Gamma;\Delta \vdash_1 a(x).P'$ implies $\emptyset;\Delta, x \vdash_1 P'$; the result follows by applying the inductive hypotesis;

$P = \overline{a}\langle b\rangle$: $\mathrm{wt}(\overline{a}\langle b\rangle) = 0_{\mathrm{lev}(a)}$ and $\mathrm{O}^\rho(\overline{a}\langle b\rangle) = \{\overline{a}\langle b\rangle\}$. In $\mathrm{O}^\rho(P)$ there is one output of level $\mathrm{lev}(a)$ and $w_{\mathrm{lev}(a)} = 1$;

$P = P_1|P_2$: $\mathrm{wt}(P_1|P_2) = \mathrm{wt}(P_1) + \mathrm{wt}(P_2)$; $\mathrm{O}^\rho(P_1|P_2) = \mathrm{O}^\rho(P_1) \cup \mathrm{O}^\rho(P_2)$. $\Gamma;\Delta \vdash_1 P_1|P_2$ can be derived by using a normal derivation (Lemma 5.2) and from this derivation it can be deduced that $\Gamma_j;\Delta_j \vdash_1 P_j$ for $j = 1, 2$ and suitables Γ_j and Δ_j. By inductive hypothesis, if $\mathrm{wt}(P_j) = \langle w_{k_j}, \ldots, w_{0_j}\rangle$, in $\mathrm{O}^\rho(P_j)$ there are at most w_{i_j} outputs of level i, thus in $\mathrm{O}^\rho(P_1|P_2)$ there are at most $w_{i_1} + w_{i_2} = w_i$ outputs of level i;

$P = (\nu\, c)P'$: $\mathrm{wt}((\nu\, c)P') = \mathrm{wt}(P') = \langle w_k, \ldots, w_0\rangle$. Suppose $c : \mathsf{T}^{\mathsf{U}}$, the other cases can be proved similarly. In the normal derivation of $\Gamma;\Delta \vdash_1 (\nu\, c)P'$ rule (T-RES)

is applied in the last step with premise $\Gamma, c; \Delta, c \vdash_1 P'$. Moreover, $\mathrm{O}^\rho((\nu\, c)P') = \mathrm{O}^\rho(P')$, by definition of $\mathrm{O}^\rho(\cdot)$. The result follows by applying the inductive hypotesis.

□

We introduce now the notion of *graph of the responsive scheduling*. Given a responsive scheduling S, this graph contains a node for every reduction step of S, and an arc from one node to another if the execution of the first reduction "activate" the second. More formally:

Definition A.3.1 (graph of the responsive scheduling). *Consider a responsive scheduling* $S = P_0 \xrightarrow{\tau\langle a_1,b_1\rangle} P_1 \xrightarrow{\tau\langle a_2,b_2\rangle} P_2 \cdots$. *We construct a graph* $\mathrm{G}(S)$ *where the nodes are the pairs* (a_i, b_i) *and there is an arc from* (a_i, b_i) *to* (a_j, b_j) *if* $j > i$, a_i *is an* ω*-receptive name and*

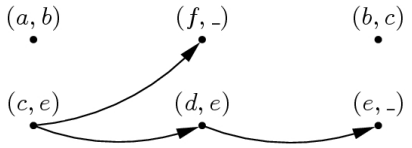
(a) $\overline{a}_j\langle b_j\rangle \notin \mathrm{O}^\rho(R)$ *for every process* R *such that* $P_{i-1} \xrightarrow{\tau\langle c_1,d_1\rangle} \cdots \xrightarrow{\tau\langle c_k,d_k\rangle} R$, *with* $k \geqslant 0$ *and each* c_i *responsive name;*

(b) $\overline{a}_j\langle b_j\rangle \in \mathrm{O}^\rho(R')$ *for a process* R' *such that* $P_i \xrightarrow{\tau\langle c_1,d_1\rangle} \cdots \xrightarrow{\tau\langle c_k,d_k\rangle} R'$, *with* $k \geqslant 0$ *and each* c_i *responsive name.*

As an example, consider the process (where c and d are ω-receptive and the other names responsive)

$$P = \overline{a}\langle b\rangle \mid a(x).\overline{x}\langle c\rangle \mid b(y).\overline{y}\langle e\rangle \mid !c(z).(\nu\, f)(f.\overline{d}\langle z\rangle \mid \overline{f}) \mid !d(w).\overline{w} \mid e$$

and the responsive scheduling $P \xrightarrow{\tau\langle a,b\rangle} \xrightarrow{\tau\langle b,c\rangle} \xrightarrow{\tau\langle c,e\rangle} \xrightarrow{\tau\langle f,\_\rangle} \xrightarrow{\tau\langle d,e\rangle} \xrightarrow{\tau\langle e,\_\rangle}$. The corresponding graph is depicted below.



In what follows, we define the level of a node as the level of the subject of the corresponding reduction and we say a node responsive (resp. ω-receptive) if the subject of the corresponding reduction is responsive (resp. ω-receptive).

Proposition A.3.4. *Suppose* $\Gamma; \Delta \vdash_1 P$, S *is a responsive scheduling from* P *and* $\mathrm{G}(S)$ *is the graph associated to* S. *For every vertex* v *of* $\mathrm{G}(S)$ *we have:*

(1) *the subgraph with root* v *contains only vertexes with level lower than* v;

(2) *if v is a responsive node then the subgraph with root v contains only the node itself;*

(3) *v has at most $\mathrm{h}(P)$ outgoing arcs.*

PROOF:

(1) By definition of graph of the responsive scheduling (Definition A.3.1) definition of active output ($\mathrm{O}(P)$) and rule (T-REP).

(2) By Definition A.3.1.

(3) By Proposition A.3.4 (2), if v is a responsive node it has not outgoing arcs. Suppose v is ω-receptive and let be $v = (a_i, b_i)$. By Proposition A.3.1 (5), if $P_{i-1} \xrightarrow{\tau\langle a_i, b_i\rangle} P_i$ then $|\,\mathrm{O}^\rho(P_i)\,| \leqslant |\,\mathrm{O}^\rho(P_{i-1})\,| + \mathrm{h}(P) - 1$. Moreover, by Proposition A.3.1 (4), each reduction with responsive subject decreases $|\,\mathrm{O}^\rho(\cdot)\,|$ by 1 and it does not add new pairs to $\mathrm{O}^\rho(\cdot)$, but possibly substitute names in already existing pairs. Hence $|\,\mathrm{O}^\rho(P_i)\,| > |\,\mathrm{O}^\rho(R_1)\,| > \cdots > |\,\mathrm{O}^\rho(R_n)\,|$ with $R_n = R'$ and each R_i reachable from P_i by a sequence of reductions with responsive subjects. Hence, by definition of active output, there can be at most $\mathrm{h}(P)$ nodes (a_j, b_j) in the scheduling such that $j > i$ and directly connected with v by an incoming arc.

□

Theorem A.3.1 (Theorem 5.3). *Let P be $(\Gamma;\Delta)$-balanced and $r \in \Delta^\rho$ and let k be the maximal level of names appearing in active output actions of P, $\mathrm{O}(P)$. Then there is at least one responsive scheduling that contains a reduction with r as subject. Moreover, in all such schedulings, the number n of reductions preceding the reduction on r is upper-bounded by $|P|^{k+1}$.*

PROOF: By Theorem 5.2 (responsiveness) $P \xrightarrow{[r]}$. For defining an upper bound for n, the length of the longest responsive scheduling that does not contains r as subject (which is finite because of Koenig's Lemma) can be estimated.

Suppose that $S = P_0 \xrightarrow{\tau\langle a_1, b_1\rangle} P_1 \xrightarrow{\tau\langle a_2, b_2\rangle} P_2 \cdots$ is this scheduling (note that every process in S is well typed by Theorem 5.1 (subject reduction)). The number of reductions in S is bounded above by the size of $\mathrm{G}(S)$. For each i, the function $\mathrm{f}(i)$ is defined as the greatest size of a subgraph of $\mathrm{G}(S)$ that has exactly one root of level i. From Proposition A.3.4, every node in the subgraph has level $< i$. By Proposition A.3.4 (3), every node in $\mathrm{G}(S)$ has at most $\mathrm{h}(P)$ outgoing arcs; considering that $\mathrm{h}(P) \leqslant |\,P\,|$:

$$\mathrm{f}(i) \leqslant 1 + |\,P\,| * \mathrm{f}(i-1) \leqslant \sum_{j=0}^{i} |\,P\,|^j = \frac{|\,P\,|^{i+1} - 1}{|\,P\,| - 1}.$$

Note that $\mathrm{f}(i)$ is monotone on i by definition.
By Proposition A.3.3, there are at most w_i roots of level i in $\mathrm{G}(S)$, thus:

$$n \leqslant \sum_{i=0}^{k} w_i * \mathrm{f}(i) \;\text{ hence }\; \mathrm{f}(i) \leqslant \mathrm{f}(k) * (w_0 + \cdots + w_k)$$

the last inequation because of the monotonicity of $\mathrm{f}(\cdot)$. $\mathrm{f}(k) = \frac{|P|^{k+1}-1}{|P|-1}$ and $w_k + \cdots + w_0 \leqslant |P|$, thus

$$n \leqslant \mathrm{f}(k) * (w_k + \cdots + w_0) \leqslant |P|^{k+1}$$

in other words n is $O(|P|^{k+1})$. □

A.4 Proofs of Section 5.4

In this section we prove that the subject reduction theorem is satisfied by type system $\vdash_2$ and the intermediary results needed for proving Theorem 5.5 (responsiveness). Firstly we introduce some preliminary results.

Proposition A.4.1 (substitution). *Suppose* $\Gamma; \Delta, x^t \vdash_2 P$, *with* $t \neq \mathsf{p}, \mathsf{n}$ *and* $x, b : \mathsf{T}$, *then*

(1) $b \notin \Delta$ *and* $b^{\mathsf{m}} \notin \Gamma$ *imply* $\Gamma; \Delta, b^t \vdash_2 P[b/x]$;
(2) $b^t \in \Delta$, $b^{\mathsf{m}} \notin \Gamma$ *and* $\mathsf{T} \neq \mathsf{S}^{[\rho,k]}$, *imply* $\Gamma; \Delta \vdash_2 P[b/x]$.

Proof: In both cases the proof is straightforward by induction on the derivation of $\Gamma; \Delta, x^t \vdash_2 P$. The additional constraints on b and t ensure that, in case P is a parallel composition, the premises of rule (T_+-Par) are still satisfied after substitution. □

Lemma A.4.1. $P >_{\alpha,\rho} R$ *and* $\Gamma; \Delta \vdash_2 P$ *imply* $\Gamma; \Delta \vdash_2 R$.

Proof: The result follows recalling that alpha-renaming is sort-respecting. □

Proposition A.4.2. $\Gamma; \Delta \vdash_2 P$ *implies:*

(1) *if* $P \xrightarrow{a(c)} P'$, *with* $a : \mathsf{T}^{\mathsf{U}}$ *and* $c : \mathsf{T}$ *then*
   (a) *if* $c \notin \Delta$ *and* $c^{\mathsf{m}} \notin \Gamma$ *then* $\Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P')); \Delta, c^t \vdash_2 P'$ *with* $t \neq \mathsf{n}, \mathsf{p}$;
   (b) *if* $c^t \in \Delta$, $c^{\mathsf{m}} \notin \Gamma$, *with* $t \neq \mathsf{n}, \mathsf{p}$ *and* $\mathsf{T} \neq \mathsf{S}^{[\rho,k]}$, *then* $\Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P')); \Delta \vdash_2 P'$;
(2) *if* $P \xrightarrow{\overline{a}\langle b \rangle} P'$ *then* $\Gamma; \Delta \ominus^+ (\{a, b\} \setminus \mathrm{on}(P')) \vdash_2 P'$;
(3) *if* $P \xrightarrow{\overline{a}(b)} P'$ *then* $\Gamma; \Delta \ominus^+ (\{a\} \setminus \mathrm{on}(P')), b \vdash_2 P'$ *if* $b : \mathsf{I}$ *and* $\Gamma, b; (\Delta, b) \ominus^+ (\{a, b\} \setminus \mathrm{on}(P')) \vdash_2 P'$ *otherwise.*

PROOF: The proof proceeds by induction on the derivation of $P \xrightarrow{\mu} P'$. In each case we distinguish the last transition rule applied. Omitted cases can be easily proved by applying the inductive hypothesis; case (ALPHA) relies on Lemma A.4.1.

(1) (IN): $a(b).P \xrightarrow{a(c)} P[c/b]$. By ($T_+$-INP), $\Gamma, a^{t''}; \Delta \vdash_2 a(b).P$ implies $a : \mathsf{T}^{[\mathsf{u},k]}$ with $\mathsf{u} \neq \omega$, $b : \mathsf{T}$ and $\Gamma; \Delta, b^t \vdash_2 P$ with $t \neq \mathsf{n}, \mathsf{p}$.

Suppose $c^{\mathsf{m}} \notin \Gamma$ and $c \notin \Delta$. By $c : \mathsf{T}$ and by Proposition A.4.1 (1) (substitution), $\Gamma; \Delta, c^t \vdash_2 P[c/b]$ with $t \neq \mathsf{n}, \mathsf{p}$ (note that $\Gamma = (\Gamma, a) \ominus^+ (\{a\} \setminus \mathrm{in}(P[c/b]))$ because $a \notin \mathrm{in}(P[c/b])$ by (T_+-INP)).

Suppose $c^{\mathsf{m}} \notin \Gamma$, $c^{t'} \in \Delta$, with $t' \neq \mathsf{n}, \mathsf{p}$, and $\mathsf{T} \neq \mathsf{S}^{[\rho,k]}$. Thus, the capabilities t and t' are univocally determined by T, hence, by $b, c : \mathsf{T}$, $t = t'$ and, by Proposition A.4.1 (2) (substitution), $\Gamma; \Delta \vdash_2 P[c/b]$.

(REP): $!a(b).P \xrightarrow{a(c)} !a(b).P \mid P[c/b]$. Suppose a +-responsive, if a is an ω-receptive name the proof proceeds similarly.

By (T_+-REP$^{\mathsf{p}}$), $\Gamma, a^{\mathsf{p}}; \Delta \vdash_2 !a(b).P$ implies $a : \mathsf{T}^{[\rho^+,k]}$, $b : \mathsf{T}$, $\Delta^\rho = \Delta^{\mathsf{p}} = \Gamma^\rho = \Gamma^{\mathsf{s}} = \Gamma^\omega = \Gamma^{\mathsf{p}} = \emptyset$ and $\Gamma; \Delta, b^t \vdash_2 P$ with $t \neq \mathsf{n}, \mathsf{p}$.

Suppose $c^{\mathsf{m}} \notin \Gamma$ and $c \notin \Delta$. By $c : \mathsf{T}$ and by Proposition A.4.1 (1) (substitution), $\Gamma; \Delta, c^t \vdash_2 P[c/b]$ with $t \neq \mathsf{n}, \mathsf{p}$. Rule ($T_+$-PAR) can be applied for deducing $\Gamma, a^{\mathsf{p}}; \Delta, c^t \vdash_2 !a(b).P \mid P[c/b]$ (note that $\Gamma, a^{\mathsf{p}} = (\Gamma, a^{\mathsf{p}}) \ominus^+ (\{a\} \setminus \mathrm{in}(!a(b).P \mid P[c/b]))$).

Suppose $c^{\mathsf{m}} \notin \Gamma$ and $c^{t'} \in \Delta$ with $t' \neq \mathsf{n}, \mathsf{p}$, depending on T, and $\mathsf{T} \neq \mathsf{S}^{[\rho,k]}$. By $c : \mathsf{T}$, $t = t'$ and, by Proposition A.4.1 (2) (substitution), $\Gamma; \Delta \vdash_2 P[c/b]$. Rule ($T_+$-PAR) can be applied for deducing $\Gamma, a^{\mathsf{p}}; \Delta \vdash_2 !a(b).P \mid P[c/b]$.

Note that in case (PAR_1) the premises of the rule are guaranteed by the additional constraints $t \neq \mathsf{n}, \mathsf{p}$ and $c^{\mathsf{m}} \notin \Gamma$.

(2) (OUT): $\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$; by ($T_+$-OUT) $\emptyset; \Delta, a^t, b^{t'} \vdash_2 \overline{a}\langle b\rangle$ implies $\Delta^\rho = \Delta^{\rho^+} = \emptyset$; hence, by ($T_+$-NIL), $\emptyset; (\Delta, a^t, b^{t'}) \ominus^+ \{a, b\} \vdash_2 \mathbf{0}$;

(OUT$^{\mathsf{p}}$): $!\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} !\overline{a}\langle b\rangle$; and $\emptyset; \Delta, a^t, b^{t'} \vdash_2 !\overline{a}\langle b\rangle$ $((\Delta, a^t, b^{t'}) \ominus^+ (\{a, b\} \setminus \mathrm{on}(!\overline{a}\langle b\rangle)) = (\Delta, a^t, b^{t'}))$.

(3) (OPEN): $(\nu\, b)P \xrightarrow{\overline{a}(b)} P'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$. Suppose $b : \mathsf{T}^{\mathsf{U}}$, if $b : \mathsf{I}$ the proof proceeds in a similar way and it cannot be $b : \mathbb{L}$ because $b \in \mathrm{on}(P)$. By (T_+-RES), $\Gamma, b^t; \Delta, b^{t'} \vdash_2 P$ and by Proposition A.4.2 (2) $\Gamma, b^t; (\Delta, b^{t'}) \ominus^+ (\{a, b\} \setminus \mathrm{on}(P')) \vdash_2 P'$.

□

Lemma A.4.2. *Suppose* $\Gamma; \Delta \vdash_2 P$. $P \xrightarrow{\overline{a}\langle b\rangle} P'$ *and* b *responsive name imply that* $b \notin \mathrm{on}(P')$.

PROOF: The proof is straightforward by induction on the derivation of $\Gamma;\Delta \vdash_2 P$. □

Theorem A.4.1 (Theorem 5.4). *$\Gamma;\Delta \vdash_2 P$ and $P \xrightarrow{[a]} P'$ imply $\Gamma';\Delta' \vdash_2 P'$, with $\Gamma' = \Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P'))$ and $\Delta' = \Delta \ominus^+ (\{a\} \setminus \mathrm{on}(P'))$.*

PROOF: The proof proceeds by induction on the derivation of $P \xrightarrow{[a]} P'$; we distinguish the last transition rule applied:

(COM$_1$): $P|R \xrightarrow{[a]} P'|R'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and $R \xrightarrow{a(b)} R'$. $\Gamma;\Delta \vdash_2 P|R$ implies $\Gamma = \Gamma_P \cup \Gamma_R$, $\Delta = \Delta_P \cup \Delta_R$, $\Gamma_P;\Delta_P \vdash_2 P$, $\Gamma_R;\Delta_R \vdash_2 R$, $\Gamma_P^\ell \cap \Gamma_R^\ell = \emptyset$ for $\ell = \rho, \mathsf{s}, \mathsf{p}$ and $\Delta_P^{\ell'} \cap \Delta_R^{\ell'} = \emptyset$ for $\ell' = \rho, \mathsf{p}$. Moreover, $\Gamma^{\mathsf{m}} \cap \Delta^{\mathsf{m}} = \emptyset$ and $\Gamma^{\mathsf{p}} \cap \Delta^{\mathsf{p}} = \emptyset$.
By $P \xrightarrow{\overline{a}\langle b\rangle} P'$, $\Gamma_P;\Delta_P \vdash_2 P$ and Proposition A.4.2 (2), $\Gamma_P;\Delta_P \ominus^+ (\{a,b\} \setminus \mathrm{on}(P')) \vdash_2 P'$. $b^t \in \Delta_P$ with $t \neq \mathsf{n},\mathsf{p}$ (because b is used as object of an output and because of (T$_+$-OUT), (T$_+$-OUT$^{\mathsf{p}}$)), thus either $t = -$ or $t = \mathsf{m}$ and by $\Gamma^{\mathsf{m}} \cap \Delta^{\mathsf{m}} = \emptyset$ we have $b^{\mathsf{m}} \notin \Gamma$.
Suppose $b \notin \Delta_R$. By $R \xrightarrow{a(b)} R'$, $\Gamma_R;\Delta_R \vdash_2 R$, $b^{\mathsf{m}} \notin \Gamma_R \subseteq \Gamma$ and Proposition A.4.2 (1), $\Gamma_R \ominus^+ (\{a\} \setminus \mathrm{in}(R'));\Delta_R, b^{t'} \vdash_2 R'$ with $t' \neq \mathsf{n},\mathsf{p}$.
Let be $\Gamma'_P = \Gamma_P$, $\Gamma'_R = \Gamma_R \ominus^+ (\{a\} \setminus \mathrm{in}(R'))$, $\Delta'_P = \Delta_P \ominus^+ (\{a,b\} \setminus \mathrm{on}(P'))$ (by Lemma A.4.2 if b is a responsive name then $b \notin \mathrm{on}(P')$) and $\Delta'_R = \Delta_R, b^{t'}$. Note that $t = t'$ because both are different from n and p, hence univocally determined by the type of b. By (T$_+$-PAR), $\Gamma';\Delta' \vdash_2 P'|R'$ with $\Gamma' = \Gamma'_P \cup \Gamma'_R = \Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P'|R'))$ and $\Delta' = \Delta \ominus^+ (\{a\} \setminus \mathrm{on}(P'|R'))$.
Similar proof if $b^t \in \Delta_R$. Note that it cannot be $b^{t'} \in \Delta_R$ with $t \neq t'$ because otherwise $\Delta_R \cup \Delta_P$ would not be defined;

(CLOSE$_1$): the proof proceeds similarly. Note that in this case $b \notin \Delta_R$ because b is bound in P;

(RES), (RES-ρ), (ALPHA), (PAR$_1$): the proof is straightforward by induction hypothesis.

□

We now prove the intermediary results needed for proving the responsiveness theorem. Firstly, we show that each name carrying (+-)responsive objects has level greather than the carried object's.

Lemma A.4.3. *Suppose P is $(\Gamma;\Delta)$-strongly balanced and $\overline{a}\langle b\rangle \in \mathrm{O}(P)$, with b (+-)responsive name, then $\mathrm{lev}(a) > \mathrm{lev}(b)$.*

PROOF: P $(\Gamma;\Delta)$-strongly balanced imply that $\Gamma^\rho = \Delta^\rho$, $\Delta^\omega \subseteq \Gamma^\omega$, $\Gamma^{\rho^+} \subseteq \Delta^{\rho^+}$ and $(\Delta^{\rho^+})^\dagger \subseteq (\Gamma^{\rho^+})^\dagger$ (similar comments for bound +-responsive names). Hence, a is used as input subject in P.

Suppose the (perhaps guarded) subprocess that use a as subject of an input in P is $(!)a(x).R$. By well-typedness of P, $\mathrm{lev}(b) = \mathrm{lev}(x)$. From ($\mathrm{T}_+$-Nil), ($\mathrm{T}_+$-Out), ($\mathrm{T}_+$-Out$^{\mathsf{p}}$) and ($\mathrm{T}_+$-Par), x is used in R in output either as subject or object. Moreover, this output cannot be guarded by an ω-receptive input, (T_+-Rep).

First of all, we prove that P $(\Gamma;\Delta)$-strongly balanced implies that each name in P carrying (+-)responsive names cannot have level equals to 0.

Consider a and, by contradiction, suppose $\mathrm{lev}(a) = 0$. By well typedness of P there are suitable Γ' and Δ' such that $\Gamma';\Delta', x^t \vdash_2 R$, with $t \neq \mathsf{n}, \mathsf{p}$. Moreover, by the typing rules for input, $\forall c \in \mathrm{os}(R) \cup \mathrm{is}(R)$ it holds that $\mathrm{lev}(c) < \mathrm{lev}(a)$. Suppose $R = \mathbf{0}$, by rule (T_+-Nil), R wouldn't be well typed because x is (+-)responsive: contradiction. If either $R = (!)d(y).R'$ or $R = \overline{d}\langle e\rangle$, it would be $\mathrm{lev}(d) < \mathrm{lev}(a) = 0$, and this is not possible because levels are positive integers: contradiction. This reasoning can be extended to the cases $R = R_1 \mid R_2$ and $R = (\nu\, t)(R')$. Hence $\mathrm{lev}(a) > 0$.

We continue by proving that $\mathrm{lev}(b) < \mathrm{lev}(a)$; the proof proceeds by induction on $\mathrm{lev}(a)$.

$\mathrm{lev}(a) = 1$**:** by (T_+-Inp), (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$), for each $c \in (\mathrm{os}(R) \cup \mathrm{is}(R))$ it holds that $\mathrm{lev}(c) < \mathrm{lev}(a)$, hence $\mathrm{lev}(c) = 0$. The output action involving x cannot be guarded by an input (because otherwise the subject of the output would have a negative level, by typing rules for input). Moreover, x is the subject of such an action, because as discussed before the level of a name carrying (+-)responsive names cannot be 0. In conclusion, $\mathrm{lev}(x) = 0 < \mathrm{lev}(a)$.

$\mathrm{lev}(a) = n$**:** Suppose x is used as subject and the output is not guarded by a replicated input ($x \in \mathrm{os}(R)$). By (T_+-Inp), (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$), for each $c \in (\mathrm{os}(R) \cup \mathrm{is}(R))$ it holds that $\mathrm{lev}(c) < \mathrm{lev}(a)$, that is $\mathrm{lev}(b) = \mathrm{lev}(x) < \mathrm{lev}(a)$. Suppose the output is guarded by a replicated input, let's say on d (which is +-responsive because x is free in R). $d \in \mathrm{is}(R)$ and $\mathrm{lev}(d) < \mathrm{lev}(a)$. By ($\mathrm{T}_+$-Rep$^{\mathsf{p}}$), $\mathrm{lev}(b) = \mathrm{lev}(x) < \mathrm{lev}(d) < \mathrm{lev}(a)$.

Suppose x is used as object of an output action, let's say $\overline{e}\langle x\rangle$. As previously seen, we have $\mathrm{lev}(e) < \mathrm{lev}(a)$, and by applying the inductive hypothesis $\mathrm{lev}(b) = \mathrm{lev}(x) < \mathrm{lev}(e) < \mathrm{lev}(a)$.

□

The following proposition ensures that each process strongly balanced – under nontrivial contexts – always has an enabled reduction involving a (+-)responsive name.

Proposition A.4.3. *Suppose P is $(\Gamma;\Delta)$-strongly balanced with $\Delta^{\rho} \cup \Gamma^{\rho^+} \neq \emptyset$. Then $P \xrightarrow{\tau\langle a,b\rangle}$ with either a or b (+-)responsive name.*

PROOF: Suppose c is the (either free or bound) (+-)responsive name with highest level appearing as input subject in P (non-guarded by a replicated input on a ω-receptive name). P is $(\Gamma;\Delta)$-strongly balanced, hence c is used in output in P and all names carrying (+-)responsive names are used both in input and output in P.

By contradiction, suppose that P cannot reduce using c as subject or object of the communication and consider the normal-form (Lemma 5.1) of $P \equiv (\nu\widetilde{d})(P_1|\cdots|P_n)$.

If P cannot reduce using c as subject or object of the communication then at least every output action $(!)\overline{a}_i\langle b_i\rangle$ (with $i = 1,\ldots,n$) involving c, or every corresponding input $(!)a_i(x).R_i$, are guarded.

Suppose that all outputs $(!)\overline{a}_i\langle b_i\rangle$, with a_i or b_i equal to c, are guarded. By rule (T_+-REP), they cannot be guarded by a replicated input on an ω-receptive name. By rules (T_+-INP) and (T_+-REP$^{\mathsf{p}}$), $(!)\overline{a}_i\langle b_i\rangle$ can be guarded by an input on a (+-)responsive name, say d, but only if $\mathrm{lev}(d) > \mathrm{lev}(a_i) \geqslant \mathrm{lev}(c)$, Lemma A.4.3. But this is a contradiction, because c has the highest level among the (+-)responsive (free or bound) names used in input in P. Hence each output $(!)\overline{a}_i\langle b_i\rangle$ involving c cannot be guarded.

Let's look at the inputs.

Suppose $a_i \neq c$ and the input is not available because guarded. a_i cannot be an ω-receptive name because otherwise the input is immediately available, (T_+-INP), (T_+-REP) and (T_+-REP$^{\mathsf{p}}$). Moreover, a_i cannot be (+-)responsive because, by Lemma A.4.3, $\mathrm{lev}(a_i) > \mathrm{lev}(c)$ and c has the highest level among the (+-)responsive free or bound names used in input in P.

Suppose $a_i = c$ and the input on c is guarded. As previously seen, by rule (T_+-REP), it cannot be guarded by a replicated input on an ω-receptive name and by rules (T_+-INP) and (T_+-REP$^{\mathsf{p}}$), $c(x)$ can be guarded by an input on a (+-)responsive name, say d. But from the well-typedness of P, it would be $\mathrm{lev}(d) > \mathrm{lev}(c)$, but c has the highest level among the (+-)responsive free or bound names used in input in P.

In both cases we have a contradiction. In conclusion, there are P_i and P_j such that $P_i \xrightarrow{\overline{a}_i\langle b_i\rangle}$ and $P_j \xrightarrow{a_i(b_i)}$, with either a_i or b_i equals to c; hence $P \xrightarrow{\tau\langle a_i,b_i\rangle}$ with either a_i or b_i equals to c. □

Lemma A.4.4. *Suppose* $\Gamma;\Delta \vdash_2 P$*, then:*

(1) $P \xrightarrow{a(b)} P'$*, with either* a *or* b *(+-)responsive name,* $a : \mathsf{T}^{\mathsf{U}}$ *and* $b : \mathsf{T}$*, implies*

   (a) $\mathrm{wt}^+(P') \prec \mathrm{wt}^+(P)$ *if the input on* a *is not replicated;*

   (b) $\mathrm{wt}^+(P') \prec \mathrm{wt}^+(P) + 0_{\mathrm{lev}(a)}$ *if the input on* a *is replicated;*

(2) $P \xrightarrow{\overline{a}\langle b\rangle} P'$ *(*$P \xrightarrow{\overline{a}(b)} P'$*) implies*

(a) $\mathrm{wt}^+(P') \preccurlyeq \mathrm{wt}^+(P) - 0_{\mathrm{lev}(a)}$ *if the output on* a *is not replicated;*

(b) $\mathrm{wt}^+(P') = \mathrm{wt}^+(P)$ *if the output on* a *is replicated.*

PROOF: In each case the proof proceeds by induction on the derivation of $P \xrightarrow{\mu} P'$; we consider the last transition rule applied.

(1) *(a)*,(IN): $a(x).P \xrightarrow{a(b)} P[b/x]$; $\mathrm{wt}^+(a(x).P) = \mathrm{wt}^+(P) + 0_{\mathrm{lev}(a)}$ and $\mathrm{wt}^+(P) = \mathrm{wt}^+(P[b/x])$ (because $x, b : \mathsf{T}$). Thus, $\mathrm{wt}^+(P[b/x]) \prec \mathrm{wt}^+(P) + 0_{\mathrm{lev}(a)} = \mathrm{wt}^+(a(x).P)$;

(b),(REP): $!a(x).P \xrightarrow{a(b)} !a(x).P \mid P[b/x]$; $\mathrm{wt}^+(!a(x).P) = 0$ and $\mathrm{wt}^+(!a(x).P|P[b/x]) \prec 0_{\mathrm{lev}(a)} = \mathrm{wt}^+(!a(x).P) + 0_{\mathrm{lev}(a)}$ because of the definition of $\mathrm{wt}^+(\cdot)$ and rule (T_+-REP$^{\mathsf{p}}$) or (T_+-REP) ($\forall c \in (\mathrm{os}(P) \cup \mathrm{is}(P)) : \mathrm{lev}(c) < \mathrm{lev}(a)$).

(2) *(a)*:

(OUT): $\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$, $\mathrm{wt}^+(\overline{a}\langle b\rangle) = 0_{\mathrm{lev}(a)}$ and $\mathrm{wt}^+(\mathbf{0}) = 0 = \mathrm{wt}^+(\overline{a}\langle b\rangle) - 0_{\mathrm{lev}(a)}$;

(OPEN): $(\nu b)P \xrightarrow{\overline{a}(b)} P'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$; by induction $\mathrm{wt}^+(P') \preccurlyeq \mathrm{wt}^+(P) - 0_{\mathrm{lev}(a)} = \mathrm{wt}^+((\nu b)P) - 0_{\mathrm{lev}(a)}$;

(b):

(OUT$^{\mathsf{p}}$): $!\overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} !\overline{a}\langle b\rangle$;

(OPEN): $(\nu b)P \xrightarrow{\overline{a}(b)} P'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and by induction $\mathrm{wt}^+(P') = \mathrm{wt}^+(P) = \mathrm{wt}^+((\nu b)P)$.

Omitted cases can be easily proved by applying the inductive hypothesis. □

The following proposition is the analog of Proposition 5.2 adapted to system $\vdash_2$ and show that $\mathrm{wt}^+(\cdot)$ is a good measure because decreases after each (+-)responsive reduction.

Proposition A.4.4 (Proposition 5.5). $\Gamma; \Delta \vdash_2 P$ *and* $P \xrightarrow{\tau\langle a,b\rangle} P'$ *with either* a *or* b *(+-)responsive, implies* $\mathrm{wt}^+(P') \prec \mathrm{wt}^+(P)$.

PROOF: By induction on the derivation of $P \xrightarrow{\tau\langle a,b\rangle} P'$, the proof proceeds by distinguishing the last transition rule applied:

(COM$_1$): $P|R \xrightarrow{\tau\langle a,b\rangle} P'|R'$ implies $P \xrightarrow{\overline{a}\langle b\rangle} P'$ and $R \xrightarrow{a(b)} R'$. $\Gamma; \Delta \vdash_2 P \mid R$ implies, (T_+-PAR), $\Gamma_1; \Delta_1 \vdash_2 P$ and $\Gamma_2; \Delta_2 \vdash_2 R$ for suitable $\Gamma_1, \Gamma_2, \Delta_1$ and Δ_2. We consider the following cases:

both input and output are non-replicated: by Lemma A.4.4 (1a,2a), $\mathrm{wt}^+(R') \prec \mathrm{wt}^+(R)$ and $\mathrm{wt}^+(P') \preccurlyeq \mathrm{wt}^+(P) - 0_{\mathrm{lev}(a)}$; that is $\mathrm{wt}^+(P'|R') = \mathrm{wt}^+(P') + \mathrm{wt}^+(R') \prec \mathrm{wt}^+(P) + \mathrm{wt}^+(R) = \mathrm{wt}^+(P|R)$;

the input is replicated: by Lemma A.4.4 (1b,2a), $\mathrm{wt}^+(R') \prec \mathrm{wt}^+(R)+0_{\mathrm{lev}(a)}$ and $\mathrm{wt}^+(P') \preccurlyeq \mathrm{wt}^+(P)-0_{\mathrm{lev}(a)}$; that is $\mathrm{wt}^+(P'|R') = \mathrm{wt}^+(P')+\mathrm{wt}^+(R') \prec \mathrm{wt}^+(P) + \mathrm{wt}^+(R) = \mathrm{wt}^+(P|R)$;

the output is replicated: by Lemma A.4.4 (1a,2b), $\mathrm{wt}^+(R') \prec \mathrm{wt}^+(R)$ and $\mathrm{wt}^+(P') = \mathrm{wt}^+(P)$; hence, $\mathrm{wt}^+(P'|R') = \mathrm{wt}^+(P') + \mathrm{wt}^+(R') \prec \mathrm{wt}^+(P) + \mathrm{wt}^+(R) = \mathrm{wt}^+(P|R)$;

($\textsc{close}_1$): in this case the proof proceeds in a similar way.

Omitted cases can be easily proved by applying the inductive hypothesis. □

The following lemma states that strong balancing is always preserved by responsive and ω-receptive reductions, while can be violated by $+$-responsive reductions, but only if the input is replicated. Moreover, strong balancing can be re-established by erasing the subprocess guarded by this input, without affecting well-typedness.

Lemma A.4.5 (Lemma 5.3). *Suppose P is $(\Gamma;\Delta)$-strongly balanced and $P \xrightarrow{\tau\langle a,b\rangle} P'$ with P' non strongly balanced. Let be $\Gamma';\Delta' \vdash_2 P'$. Then*

(1) $a \in (\Gamma'^{\rho^+} \setminus \Delta'^{\rho^+}) \cup (\mathrm{r}_{\mathrm{i}}^+(P') \setminus \mathrm{r}_{\mathrm{o}}^+(P'))$;

(2) $P \equiv (\nu\, \widetilde{d})(!a(x).R \mid R')$ *and* $a \notin \mathrm{in}(R,R')$;

(3) $P' \equiv (\nu\, \widetilde{d})(!a(x).R \mid R[b/x] \mid R'')$ *and* $a \notin \mathrm{in}(R,R[b/x],R'')$;

(4) $P'' = (\nu\, \widetilde{d})(R[b/x] \mid R'')$ *is strongly balanced.*

Proof: Recall that by Theorem 5.4 (subject reduction) we have $\Gamma';\Delta' \vdash_2 P'$, with $\Gamma' = \Gamma \ominus^+ (\{a\} \setminus \mathrm{in}(P'))$ and $\Delta' = \Delta \ominus^+ (\{a\} \setminus \mathrm{on}(P'))$.

(1) P' non strongly balanced means that Definition 5.10 is not satisfied, hence at least one of its three points does not hold.
It cannot be $\Gamma'^{\rho} \neq \Delta'^{\rho}$ because of the linearity of responsive names (rules (T_+-Par), (T_+-Inp), (T_+-Rep) and (T_+-Rep$^{\mathsf{p}}$)) and $\Gamma^{\rho} = \Delta^{\rho}$.
It cannot be $\Delta'^{\omega} \not\subseteq \Gamma'^{\omega}$ because ω-receptive names are used as subject of replicated inputs (rules (T_+-Rep), (T_+-Inp) and (T_+-Rep$^{\mathsf{p}}$)), which cannot disappear, and $\Delta^{\omega} \subseteq \Gamma^{\omega}$.
Similarly, it cannot be neither $(\Delta^{\rho^+})^{\dagger} \not\subseteq (\Gamma^{\rho^+})^{\dagger}$ nor $(\mathrm{r}_{\mathrm{o}}^+(P))^{\dagger} \not\subseteq (\mathrm{r}_{\mathrm{i}}^+(P))^{\dagger}$, because $+$-responsive names carrying $(+)$-responsive objects are used as subject of replicated inputs, (T_+-Inp), which cannot disappear.
In conclusion, $a \in (\Gamma'^{\rho^+} \setminus \Delta'^{\rho^+}) \cup (\mathrm{r}_{\mathrm{i}}^+(P') \setminus \mathrm{r}_{\mathrm{o}}^+(P'))$.

(2) We firstly prove that a is used as subject of a replicated input. By contradiction, assume a used as subject of non-replicated inputs. The are at least two of such inputs in P, because otherwise it cannot be $a \in (\Gamma'^{\rho^+} \setminus \Delta'^{\rho^+}) \cup (\mathrm{r}_{\mathrm{i}}^+(P') \setminus \mathrm{r}_{\mathrm{o}}^+(P'))$.
Hence, by rule (T_+-Par) a has to be used as subject of a replicated output

(which cannot disappear), hence $a \in {\Delta'}^{\rho^+} \cup \mathrm{r}_{\mathrm{o}}^+(P')$ and this is not the case. Thus, a is used as subject of a replicated input in P, hence in P'. By (T_+-Par) and (T_+-Rep$^{\mathsf{p}}$), a is used once in input subject position. Moreover, $P \xrightarrow{[a]}$ implies that such input cannot be guarded. In conclusion, $P \equiv (\nu\, \widetilde{d})(!a(x).R \mid R')$ and by (T_+-Par) and (T_+-Rep$^{\mathsf{p}}$) $a \notin \mathrm{in}(R, R')$.

(3) By point (2) and the reduction $P \xrightarrow{\tau\langle a,b\rangle} P'$.

(4) By points (1,2,3), $\Gamma; \Delta \ominus^+ \{a\} \vdash_2 P' \equiv (\nu\, \widetilde{d})(!a(x).R \mid R[b/x] \mid R'')$.
Suppose $a \in \mathrm{fn}(P)$ (hence $a \in \mathrm{fn}(P')$). By the typing rules for restriction (suppose $\widetilde{d}$ does not contain inert names) $\Gamma, \widetilde{d^c}; \Delta \ominus^+ \{a\}, \widetilde{d^{c'}} \vdash_2 !a(x).R \mid R[b/x] \mid R''$. By ($\mathrm{T}_+$-Par), $\Gamma, \widetilde{d^c} = \Gamma_1 \cup \{a\} \cup \Gamma_2$ and $\Delta \ominus^+ \{a\}, \widetilde{d^{c'}} = \Delta_1 \cup \Delta_2$ with $\Gamma_1, a; \Delta_1 \vdash_2 !a(x).R$ and $\Gamma_2; \Delta_2 \vdash_2 R[b/x] \mid R''$. Moreover, $\Gamma_1 \subseteq \Gamma_2$ and $\Delta_1 \subseteq \Delta_2$, hence $\Gamma_2 = \Gamma \ominus^+ \{a\}, \widetilde{d^c}$ and $\Delta_2 = \Delta \ominus^+ \{a\}, \widetilde{d^c}$. Again by the typing rules for restriction, $\Gamma \ominus^+ \{a\}; \Delta \ominus^+ \{a\} \vdash_2 (\nu\, \widetilde{d})(R[b/x] \mid R'') = P''$ and P'' is strongly balanced.
The proof proceeds similarly in case $a \in \widetilde{d}$. Note that in this case $\Gamma; \Delta \vdash_2 (\nu\, \widetilde{d})(R[b/x] \mid R'')$ follows by applying (T_+-Weak-Γ) and (T_+-Weak-Δ).

□

A.5 Proof of Proposition 5.6

In this section, the encoding of ORC introduced in Table 5.12 is shown to be correct. In what follows, given an ORC term f, we write $\mathrm{fv}(f)$ for the set of free variables in f and $\xrightarrow{\hat{\mu}}$ stands for $\xrightarrow{\mu}$ if $\mu \neq \tau$ or a possible τ reduction if $\mu = \tau$. We borrow from [14] the definition of *expansion preorder*, $\gtrsim$, and from [111] the definition of *strong bisimulation relation*, $\sim$. For the sake of completeness, we recall both definitions.

Definition A.5.1 (expansion preorder). *A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is an expansion preorder if $S\mathcal{R}P$ implies:*

(1) *whenever $S \xrightarrow{\mu} S'$, there exists P' s.t. $P \xrightarrow{\hat{\mu}} P'$ and $S'\mathcal{R}P'$;*

(2) *whenever $P \xrightarrow{\mu} P'$, there exists S' s.t. $S \stackrel{\mu}{\Rightarrow} S'$ and $S'\mathcal{R}P'$.*

We say that S expands P, written $S \gtrsim P$, if $S\mathcal{R}P$ for some expansion $\mathcal{R}$.

Definition A.5.2 (strong bisimulation). *A symmetric relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a strong bisimulation if $S\mathcal{R}P$ implies that whenever $S \xrightarrow{\mu} S'$, there exists P' such that $P \xrightarrow{\mu} P'$ and $S'\mathcal{R}P'$. We say that S is strongly bisimilar to P, written $S \sim P$, if $S\mathcal{R}P$ for some strong bisimulation $\mathcal{R}$.*

The following lemmata introduce some properties of $\sim$ and $\gtrsim$ that are useful for

proving the correctness of the encoding. The (omitted) proofs rely on asynchrony and input locality of the calculus.

Lemma A.5.1.

(1) $(\nu x)(!\overline{x}\langle c\rangle \mid P_1 \mid P_2) \sim (\nu x)(!\overline{x}\langle c\rangle \mid P_1) \mid (\nu x)(!\overline{x}\langle c\rangle \mid P_2)$ *if* $x \notin \mathrm{O}(P_1, P_2)$;
(2) $(\nu a)(!a(y).P \mid P_1 \mid P_2) \sim (\nu a)(!a(y).P \mid P_1) \mid (\nu a)(!a(y).P \mid P_2)$ *if* $a \notin \mathrm{in}(P_1, P_2)$;
(3) $(\nu x)(!\overline{x}\langle c\rangle \mid !a(y).P) \sim !a(y).(\nu x)(!\overline{x}\langle c\rangle \mid P)$ *if* $a, y \neq x$;
(4) $(\nu x)(!x(z).P' \mid !a(y).P) \sim !a(y).(\nu x)(!x(z).P' \mid P)$ *if* $a, y \neq x$ *and* $a, y \notin \mathrm{fn}(P')$.

Lemma A.5.2. $P' \gtrsim P$ *implies:*

(1) $P' \mid R \gtrsim P \mid R$;
(2) $(\nu\widetilde{d})P' \gtrsim (\nu\widetilde{d})P$;
(3) $\alpha.P' \gtrsim \alpha.P$ *with either* $\alpha = !a(y)$ *or* $\alpha = a(y)$.

In the following proofs, recall that given an ORC term f, in $[\![f]\!]_s$ all site and expression names are used only in output subject position and all variables only in input subject position. Moreover, if f is a closed term $[\![f]\!]_s$ can interact with the environment only by calling sites or expressions or by publishing (outputting) on s.

Proposition A.5.1. $(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![g]\!]_s)) \gtrsim (\nu\widetilde{d})(D \mid [\![g[c/x]]\!]_s)$.

PROOF: The proof proceeds by induction on the structure of g:

$g = M(p)$, $g = E(p)$ **or** $g = let(p)$**:** with $p \neq x$. $M(p)[c/x] = M(p)$ and $(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![M(p)]\!]_s)) \sim (\nu\widetilde{d})(D \mid [\![M(p)]\!]_s)$ because $x \notin \mathrm{fn}([\![M(p)]\!]_s)$ (similarly for $g = E(p)$ and $g = let(p)$);

$g = let(x)$**:**

$$\begin{aligned}
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![let(x)]\!]_s)) &= && \text{(by definition of } [\![f]\!]_s\text{)}\\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid x(y).\overline{s}\langle y\rangle)) &\xrightarrow{\tau} &&\\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid \overline{s}\langle c\rangle)) &\sim && (x \neq s, c)\\
(\nu\widetilde{d})(D \mid \overline{s}\langle c\rangle) &= && \text{(by definition of } [\![f]\!]_s\text{)}\\
(\nu\widetilde{d})(D \mid [\![let(x)[c/x]]\!]_s)\ . &&&
\end{aligned}$$

$g = E(x)$**:**

$$\begin{aligned}
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![E(x)]\!]_s)) &= && \text{(by definition of } [\![f]\!]_s\text{)}\\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid x(y).\overline{E}\langle y, s\rangle)) &\xrightarrow{\tau} &&\\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid \overline{E}\langle c, s\rangle)) &\sim && (x \neq E, c, s)\\
(\nu\widetilde{d})(D \mid \overline{E}\langle c, s\rangle) &= && \text{(by definition of } [\![f]\!]_s\text{)}\\
(\nu\widetilde{d})(D \mid [\![E(x)[c/x]]\!]_s)\ . &&&
\end{aligned}$$

$g = M(x)$:

$$
\begin{aligned}
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![M(x)]\!]_s)) \;&=\; && \text{(by definition of } [\![f]\!]_s) \\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid x(y).[\![M(y)]\!]_s)) \;&\xrightarrow{\tau}\; \\
(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![M(c)]\!]_s)) \;&\sim && (x \notin \mathrm{fv}([\![M(c)]\!]_s)) \\
(\nu\widetilde{d})(D \mid [\![M(x)[c/x]]\!]_s)\ .
\end{aligned}
$$

$g = f \mid f'$:

$$
\begin{aligned}
&(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f \mid f']\!]_s)) \\
&\quad = && \text{(by definition of } [\![f]\!]_s) \\
&(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_s \mid [\![f']\!]_s)) \\
&\quad \sim && \text{(by Lemma A.5.1 (1,2))} \\
&(\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_s)) \mid (\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f']\!]_s)) \\
&\quad \gtrsim && \text{(by induction and Lemma A.5.2 (1)} \\
&(\nu\widetilde{d})(D \mid [\![f[c/x]]\!]_s) \mid (\nu\widetilde{d})(D \mid [\![f'[c/x]]\!]_s) \\
&\quad \sim && \text{(by def. of } [\![f]\!]_s \text{ and Lemma A.5.1 (1,2))} \\
&(\nu\widetilde{d})(D \mid [\![(f \mid f')[c/x]]\!]_s)\ .
\end{aligned}
$$

$g = f > y > f'$:

$$
\begin{aligned}
&(\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f > y > f']\!]_s)\big) \\
&\quad = && \text{(by definition of } [\![f]\!]_s) \\
&(\nu\widetilde{d})\Big(D \mid (\nu x)\big(!\overline{x}\langle c\rangle \mid (\nu w)([\![f]\!]_w \mid !w(z).(\nu y)(!\overline{y}\langle z\rangle \mid [\![f']\!]_s))\big)\Big) \\
&\quad \sim && \text{(by Lemma A.5.1 (1,2))} \\
&(\nu w)\Big((\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_w)\big) \mid (\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid !w(z).(\nu y)(!\overline{y}\langle z\rangle \mid [\![f']\!]_s))\big)\Big) \\
&\quad \sim && \text{(by Lemma A.5.1 (3,4))} \\
&(\nu w)\Big((\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_w)\big) \mid !w(z).(\nu\widetilde{d}, y)\big(!\overline{y}\langle z\rangle \mid D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f']\!]_s)\big)\Big) \\
&\quad \gtrsim && \text{(by induction and Lemma A.5.2)} \\
&(\nu w)\Big((\nu\widetilde{d})(D \mid [\![f[c/x]]\!]_w) \mid !w(z).(\nu\widetilde{d}, y)(!\overline{y}\langle z\rangle \mid D \mid [\![f'[c/x]]\!]_s)\Big) \\
&\quad \sim && \text{(by Lemma A.5.1 (2))} \\
&(\nu\widetilde{d})(D \mid (\nu w)([\![f[c/x]]\!]_w \mid !w(z).(\nu y)(!\overline{y}\langle z\rangle \mid [\![f'[c/x]]\!]_s))) \\
&\quad = && \text{(by definition of } [\![f]\!]_s) \\
&(\nu\widetilde{d})(D \mid [\![(f > y > f')[c/x]]\!]_s\ .
\end{aligned}
$$

$g = f \ \mathit{where}\ y :\in f'$**:**

$$
\begin{aligned}
&(\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f \ \mathit{where}\ y :\in f']\!]_s)\big)\\
&\quad = && \text{(by definition of } [\![f]\!]_s)\\
&(\nu\widetilde{d})\Big(D \mid (\nu x)\big(!\overline{x}\langle c\rangle \mid (\nu w)([\![f']\!]_w \mid (\nu y)(w(z).!\overline{y}\langle z\rangle \mid [\![f]\!]_s))\big)\Big)\\
&\quad \sim && \text{(by Lemma A.5.1 (1,2))}\\
&(\nu w)\Big((\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f']\!]_w)\big) \mid (\nu\widetilde{d})\big(D \mid (\nu x, y)(!\overline{x}\langle c\rangle \mid w(z).!\overline{y}\langle z\rangle \mid [\![f]\!]_s)\big)\Big)\\
&\quad \sim && (\mathrm{fn}(w(z).!\overline{y}\langle z\rangle) \cap \{\widetilde{d}, x\} = \emptyset)\\
&(\nu w)\Big((\nu\widetilde{d})\big(D \mid (\nu x)(!\overline{x}\langle c\rangle | [\![f']\!]_w)\big) \mid (\nu y)\big(w(z).!\overline{y}\langle z\rangle \mid (\nu\widetilde{d})(D \mid (\nu x)(!\overline{x}\langle c\rangle | [\![f]\!]_s))\big)\Big)\\
&\quad \gtrsim && \text{(by induction hp. and Lemma A.5.2 (1,2)}\\
&(\nu w)\Big((\nu\widetilde{d})\big(D \mid [\![f'[c/x]]\!]_w \mid (\nu y)(w(z).!\overline{y}\langle z\rangle \mid (\nu\widetilde{d})(D \mid [\![f[c/x]]\!]_s))\big)\Big)\\
&\quad \sim && \text{(by Lemma A.5.1 (2))}\\
&(\nu\widetilde{d})\Big(D \mid (\nu w)([\![f'[c/x]]\!]_w \mid (\nu y)(w(z).!\overline{y}\langle z\rangle \mid [\![f[c/x]]\!]_s))\Big)\\
&\quad = && \text{(by definition of } [\![f]\!]_s)\\
&(\nu\widetilde{d})(D \mid [\![(f \ \mathit{where}\ y :\in f')[c/x]]\!]_s)\ .
\end{aligned}
$$

□

Proposition A.5.2. *Suppose D is a set of function and site definitions. $(\nu\widetilde{d}, y)(D|[\![f]\!]_y|P) \gtrsim (\nu\widetilde{d})(D|P)$ if $y \notin \mathrm{fn}(P)$, $\widetilde{d} = \mathrm{in}(D)$, $\widetilde{d} \cap \mathrm{in}(P) = \emptyset$ and f is closed.*

Proof: By Lemma A.5.1 (2) $(\nu\widetilde{d}, y)(D|[\![f]\!]_y|P) \sim (\nu\widetilde{d}, y)(D|[\![f]\!]_y) \mid (\nu\widetilde{d})(D|P)$. Moreover, $(\nu\widetilde{d}, y)(D|[\![f]\!]_y) \gtrsim \mathbf{0}$ because $\mathrm{fn}([\![f]\!]_y) \subseteq \{\widetilde{d}, y\}$ and, by definition of $[\![\cdot]\!]_y$, name y cannot be extruded. Hence, by Lemma A.5.1 (2), $(\nu\widetilde{d}, y)(D|[\![f]\!]_y|P) \gtrsim (\nu\widetilde{d})(D|P)$. □

The following proposition is a first step towards proving the correctness of the encoding.

In what follows λ represents a generic ORC's label and can be either $!c$ or τ. We define $[\![\lambda]\!]_s$ and $[\![\mu]\!]^{-1}$ as follows: $[\![!c]\!]_s = \overline{s}\langle c\rangle$, $[\![\tau]\!]_s = \tau$, $[\![\overline{s}\langle c\rangle]\!]^{-1} = !c$ and $[\![\tau]\!]^{-1} = \tau$.

Proposition A.5.3. *Let f be a closed* ORC *term.*

1. $f \overset{\lambda}{\hookrightarrow} g$ *implies* $(\nu\widetilde{d})(D \mid [\![f]\!]_s) \xrightarrow{[\![\lambda]\!]_s} \gtrsim (\nu\widetilde{d})(D \mid [\![g]\!]_s)$;
2. $(\nu\widetilde{d})(D \mid [\![f]\!]_s) \overset{\mu}{\longrightarrow} (\nu\widetilde{d})(D \mid P)$ *implies* $f \overset{[\![\mu]\!]^{-1}}{\hookrightarrow} g$, *with* $(\nu\widetilde{d})(D \mid P) \gtrsim (\nu\widetilde{d})(D \mid [\![g]\!]_s)$;
3. $f \overset{!c}{\hookrightarrow}$ *implies* $[\![f]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$;
4. $[\![f]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ *implies* $f \overset{!c}{\hookrightarrow}$.

PROOF:

(1) This case is straightforward by induction on the derivation of $f \overset{\lambda}{\hookrightarrow} g$. The base cases are (PUB), (SITE) and (DEF). In the other cases the result is obtained by applying the inductive hypothesis and Lemma A.5.2. Moreover, in cases (SEQ2) and (WH2) also Proposition A.5.1 and A.5.2 are applied.

(2) The proof proceeds by induction on the derivation of $\overset{\mu}{\longrightarrow}$, by considering only closed ORC terms. The most interesting cases are sequential composition and asymmetric parallel composition. In the other cases the proof proceeds by applying the inductive hypothesis and Lemma A.5.2.

$[\![f > x > g]\!]_s$**:** $(\nu\widetilde{d})\big(D \mid (\nu y)([\![f]\!]_y \mid !y(z).(\nu x)(!\overline{x}\langle z\rangle \mid [\![g]\!]_s))\big) \overset{\mu}{\longrightarrow} (\nu\widetilde{d})(D \mid P)$ implies $(\nu\widetilde{d})(D \mid [\![f]\!]_y) \overset{\mu'}{\longrightarrow} (\nu\widetilde{d})(D \mid P')$. By induction, $f \overset{[\![\mu']\!]^{-1}}{\hookrightarrow} f'$ and $(\nu\widetilde{d})(D \mid P') \gtrsim (\nu\widetilde{d})(D \mid [\![f']\!]_y)$. We distinguish two cases depending on μ':

$\mu' \neq \overline{y}\langle c\rangle$**:** in this case $f \overset{[\![\mu']\!]^{-1}}{\hookrightarrow} f'$ implies, by (SEQ1), $f > x > g \overset{[\![\mu']\!]^{-1}}{\hookrightarrow} f' > x > g$; moreover, $(\nu\widetilde{d})(D \mid P) = (\nu\widetilde{d})\big(D \mid (\nu y)\big(P' \mid !y(z).(\nu x)(!\overline{x}\langle z\rangle \mid [\![g]\!]_s)\big)\big) \gtrsim (\nu\widetilde{d})\big(D \mid (\nu y)\big([\![f']\!]_y \mid !y(z).(\nu x)(!\overline{x}\langle z\rangle \mid [\![g]\!]_s)\big)\big) = (\nu\widetilde{d})(D \mid [\![f' > x > g]\!]_s)$ (Lemma A.5.2);

$\mu' = \overline{y}\langle c\rangle$**:** in this case, by induction, $f \overset{!c}{\hookrightarrow} f'$ and, by (SEQ2), $f > x > g \overset{\tau}{\hookrightarrow} (f' > x > g) \mid g[c/x]$. By $(\nu\widetilde{d})(D \mid P') \gtrsim (\nu\widetilde{d})(D \mid [\![f']\!]_y)$, Lemma A.5.2 and Proposition A.5.1:

$$
\begin{aligned}
& (\nu\widetilde{d})(D \mid P) \\
= \;& (\nu\widetilde{d})\big(D \mid (\nu y)\big(P' \mid !y(z).(\nu x)(!\overline{x}\langle z\rangle \mid [\![g]\!]_s) \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![g]\!]_s)\big)\big) \\
\gtrsim \;& (\nu\widetilde{d})\big(D \mid (\nu y)\big([\![f']\!]_y \mid !y(z).(\nu x)(!\overline{x}\langle z\rangle \mid [\![g]\!]_s) \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![g]\!]_s)\big)\big) \\
= \;& (\nu\widetilde{d})(D \mid [\![f' > x > g]\!]_s \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![g]\!]_s)) \\
\gtrsim \;& (\nu\widetilde{d})(D \mid [\![f' > x > g]\!]_s \mid [\![g[c/x]]\!]_s) \;;
\end{aligned}
$$

$[\![f \textit{ where } x :\in g]\!]_s$**:** $(\nu\widetilde{d})\big(D \mid (\nu y)\big([\![g]\!]_y \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![f]\!]_s)\big)\big) \overset{\mu}{\longrightarrow} (\nu\widetilde{d})(D \mid P)$; we distinguish the following cases:

$(\nu\widetilde{d})(D \mid [\![g]\!]_y) \overset{\mu}{\longrightarrow} (\nu\widetilde{d})(D \mid P')$ **with** $\mu \neq \overline{y}\langle c\rangle$**:** by applying the inductive hypothesis, $g \overset{[\![\mu]\!]^{-1}}{\hookrightarrow} g'$ and $(\nu\widetilde{d})(D \mid P') \gtrsim (\nu\widetilde{d})(D \mid [\![g']\!]_y)$.

Moreover, by Lemma A.5.2:

$$\begin{aligned}
& (\nu\widetilde{d})(D \mid P) \\
= \;& (\nu\widetilde{d})\big(D \mid (\nu y)\big(P' \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![f]\!]_s)\big)\big) \\
\gtrsim \;& (\nu\widetilde{d})\big(D \mid (\nu y)\big([\![g']\!]_y \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![f]\!]_s)\big)\big) \\
= \;& (\nu\widetilde{d})(D \mid [\![f \ \mathit{where}\ x :\in g']\!]_s)
\end{aligned}$$

and $g \overset{[\![\mu]\!]^{-1}}{\hookrightarrow} g'$ implies, by (WH1), $f \ \mathit{where}\ x :\in g \overset{[\![\mu]\!]^{-1}}{\hookrightarrow} f \ \mathit{where}\ x :\in g'$;

$(\nu\widetilde{d})(D \mid [\![f]\!]_s) \xrightarrow{\mu} (\nu\widetilde{d})(D \mid P')$**:** in this case the proof proceeds in a similar way;

$(\nu\widetilde{d})(D \mid [\![g]\!]_y) \xrightarrow{\overline{y}\langle c\rangle} (\nu\widetilde{d})(D \mid P')$**:** by induction, $g \overset{!c}{\hookrightarrow} g'$, $(\nu\widetilde{d})(D \mid P') \gtrsim (\nu\widetilde{d})(D \mid [\![g']\!]_y)$ and $f \ \mathit{where}\ x :\in g \overset{\tau}{\hookrightarrow} f[c/x]$, by (WH2).
Moreover, $(\nu\widetilde{d})(D \mid P) = (\nu\widetilde{d})\big(D \mid (\nu y)\big(P' \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_s)\big)\big) \gtrsim (\nu\widetilde{d})\big(D \mid (\nu y)\big([\![g']\!]_y \mid (\nu x)(!\overline{x}\langle c\rangle \mid [\![f]\!]_s)\big)\big) \gtrsim (\nu\widetilde{d})(D \mid [\![f[c/x]]\!]_s)$ by Proposition A.5.1 and Proposition A.5.2 (recall that f is a closed term and $y \notin \mathrm{fn}([\![f]\!]_s)$.)

(3) By induction on transitions, we distinguish the following cases:

$let(c) \overset{!c}{\hookrightarrow}$**:** $[\![let(c)]\!]_s = \overline{s}\langle c\rangle \xrightarrow{\overline{s}\langle c\rangle}$;

$f \mid g \overset{!c}{\hookrightarrow}$**:** implies, by either (PAR1) or (PAR2), either $f \overset{!c}{\hookrightarrow}$ or $g \overset{!c}{\hookrightarrow}$; by induction either $[\![f]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ or $[\![g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ and $[\![f \mid g]\!]_s = [\![f]\!]_s \mid [\![g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$;

$g \ \mathit{where}\ x :\in f \overset{!c}{\hookrightarrow}$**:** implies, (WH3), $g \overset{!c}{\hookrightarrow}$, and by induction $[\![g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$. $[\![g \ \mathit{where}\ x :\in f]\!]_s = (\nu y)\big([\![f]\!]_y \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![g]\!]_s)\big)$ and $(\nu y)\big([\![f]\!]_y \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![g]\!]_s)\big) \xrightarrow{\overline{s}\langle c\rangle}$.

(4) We distinguish the following cases:

$[\![let(c)]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$**:** $[\![let(c)]\!]_s = \overline{s}\langle c\rangle \xrightarrow{\overline{s}\langle c\rangle}$ and $let(c) \overset{!c}{\hookrightarrow}$, (PUB);

$[\![f \mid g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$**:** $[\![f \mid g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ implies either $[\![f]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ or $[\![g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$. By induction, either $f \overset{!c}{\hookrightarrow}$ or $g \overset{!c}{\hookrightarrow}$, hence $f \mid g \overset{!c}{\hookrightarrow}$, by either (PAR1) or (PAR2);

$[\![g \ \mathit{where}\ x :\in f]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$**:** $(\nu y)\big([\![f]\!]_y \mid (\nu x)(y(z).!\overline{x}\langle z\rangle \mid [\![g]\!]_s)\big) \xrightarrow{\overline{s}\langle c\rangle}$ implies $[\![g]\!]_s \xrightarrow{\overline{s}\langle c\rangle}$ and by induction $g \overset{!c}{\hookrightarrow}$, that is $g \ \mathit{where}\ x :\in f \overset{!c}{\hookrightarrow}$, (WH3).

□

Proposition A.5.4. *Consider an* ORC *term f and suppose D_f well typed. If* $\mathrm{fv}(f) = \widetilde{x}$*, then* $F = (\nu \widetilde{d}, \widetilde{x})([\![f]\!]_s \mid \prod_{x\in\widetilde{x}} !\overline{x}\langle c\rangle \mid D_f \mid !s(x).\mathbf{0})$*, with* $\mathrm{fn}(F) = \{s\}$*, c inert and $\widetilde{d}$, $\widetilde{x}$ and s +-responsive names, is strongly balanced.*

PROOF: Well typedness of $[\![f]\!]_s$ is easy to prove by induction on the structure of f. In particular $\Gamma; \Delta \vdash_2 [\![f]\!]_s$, for suitable Γ and Δ such that $\Gamma^\rho = \emptyset$, $\mathrm{dom}(\Gamma) = \mathrm{fv}(f)$ (annotated with capability m) and $\mathrm{dom}(\Delta)$ contains only s and some expression and site names, annotated with m. Hence well-typedness of F is ensured. Balancing of F may be proved by induction on the structure of f.

As an example, suppose $f = g_2 \ \mathit{where}\ y :\in g_1$. In this case

$$F = (\nu\, \widetilde{x}, \widetilde{d})\Big((\nu\, r)\big([\![g_1]\!]_r | (\nu\, y)(r(z).!\overline{y}\langle z\rangle | [\![g_2]\!]_s)\big) | \prod_{x\in\widetilde{x}} !\overline{x}\langle c\rangle | D_f | !s(x)\Big)$$

where $\widetilde{x} = \widetilde{x}_1 \cup \widetilde{x}_2$, with $\widetilde{x}_1 = \mathrm{fv}(g_1)$ and $\widetilde{x}_2 = \mathrm{fv}(g_2) \setminus \{y\}$, and $\widetilde{d} = \widetilde{d}_1 \cup \widetilde{d}_2$, with $\widetilde{d}_1$ and $\widetilde{d}_2$ containing all names corresponding to sites and expressions called respectively by g_1 and g_2.

By induction, $G_1 = (\nu \widetilde{d}_1, \widetilde{x}_1)([\![g_1]\!]_r \mid \prod_{x\in\widetilde{x}_1} !\overline{x}\langle c\rangle \mid D_1 \mid !r(z).\mathbf{0})$ and $G_2 = (\nu \widetilde{d}_2, \widetilde{x}_2, y)([\![g_2]\!]_s \mid \prod_{x\in\widetilde{x}_2} !\overline{x}\langle c\rangle \mid !\overline{y}\langle c\rangle \mid D_2 \mid !s(x).\mathbf{0})$ are strongly balanced.

Note that channel r is +-responsive and does not carry (+-)responsive names, hence if we substitute $!r(z).\mathbf{0}$ with $r(z).\mathbf{0}$ then G_1 is still strongly-balanced. Thus, given that g_1 and g_2 can share only site, expression names and variables (which are used only in output – resp. input – in $[\![g_1]\!]_r$ and $[\![g_2]\!]_s$ and replicated in input in D – resp. replicated output in $\prod !\overline{x}\langle c\rangle$):

$$(\nu\, \widetilde{d}, \widetilde{x}, y)\big([\![g_1]\!]_r \mid \prod_{x\in\widetilde{x}_1} !\overline{x}\langle c\rangle \mid D_f \mid r(z).\mathbf{0} \mid [\![g_2]\!]_s \mid \prod_{x\in\widetilde{x}_2} !\overline{x}\langle c\rangle \mid !\overline{y}\langle c\rangle \mid !s(x).\mathbf{0}\big)$$

can be rewritten as

$$(\nu\, \widetilde{d}, \widetilde{x})\Big((\nu\, r, y)\big([\![g_1]\!]_r \mid r(z).\mathbf{0} \mid !\overline{y}\langle c\rangle \mid [\![g_2]\!]_s\big) \mid \prod_{x\in\widetilde{x}} !\overline{x}\langle c\rangle \mid D_f \mid !s(x).\mathbf{0}\Big)\ .$$

Given that G_1 and G_2 are strongly balanced, the process below is strongly balanced too

$$(\nu\, \widetilde{d}, \widetilde{x})\Big((\nu\, r)\big([\![g_1]\!]_r \mid (\nu\, y)(r(z).!\overline{y}\langle z\rangle \mid [\![g_2]\!]_s)\big) \mid \prod_{x\in\widetilde{x}} !\overline{x}\langle c\rangle \mid D_f \mid !s(x).\mathbf{0}\Big) = F\ .$$

□

Proposition A.5.5 (Proposition 5.6). *Let f be a closed* ORC *term and suppose D_f is well typed.*

(1) $[\![f]\!]_s$ *is well-typed and* $F \stackrel{\triangle}{=} (\nu\widetilde{d})([\![f]\!]_s \mid D_f \mid !s(x).\mathbf{0})$*, with* s *and* $\widetilde{d}$ *+-responsive, is strongly balanced;*

(2) $f \stackrel{!c}{\Longrightarrow}$ *if and only if* $F \xrightarrow{\tau\langle s,c\rangle}$.

Proof:

(1) Well-typedness of $[\![f]\!]_s$ and balancing of F follow by Proposition A.5.4.

(2) **(⇒):** $f \stackrel{!c}{\Longrightarrow}$ means that $f \stackrel{\tau}{\hookrightarrow}{}^* g \stackrel{!c}{\hookrightarrow}$; by Proposition A.5.3 (1), $f \stackrel{\tau}{\hookrightarrow} f'$ implies $(\nu\widetilde{d})(D_f \mid [\![f]\!]_s) \stackrel{\tau}{\rightarrow} (\nu\widetilde{d})(D_f \mid P') \gtrsim (\nu\widetilde{d})(D_f \mid [\![f']\!]_s)$, $f' \stackrel{\tau}{\hookrightarrow} f''$ implies $(\nu\widetilde{d})(D_f \mid [\![f']\!]_s) \stackrel{\tau}{\rightarrow} (\nu\widetilde{d})(D_f \mid P'') \gtrsim (\nu\widetilde{d})(D_f \mid [\![f'']\!]_s)$, and so on. Thus, $f \stackrel{\tau}{\hookrightarrow}{}^* g$ implies $(\nu\widetilde{d})(D_f \mid [\![f]\!]_s) \stackrel{\tau}{\rightarrow}{}^* (\nu\widetilde{d})(D_f \mid P) \gtrsim (\nu\widetilde{d})(D_f \mid [\![g]\!]_s)$ and $g \stackrel{!c}{\hookrightarrow}$ implies, by Proposition A.5.3 (3), $(\nu\widetilde{d})(D_f \mid [\![g]\!]_s) \xrightarrow{\overline{s}\langle c\rangle}$; thus by "$\gtrsim$", $(\nu\widetilde{d})(D_f \mid P) \xrightarrow{\overline{s}\langle c\rangle}$ and $(\nu\widetilde{d})(D_f \mid [\![f]\!]_s \mid !s(x).\mathbf{0}) \xrightarrow{\tau\langle s,c\rangle}$;

(⇐): in this case we can proceed similarly, the result follows by applying Proposition A.5.3 (2,4).

□

Appendix

B

Proofs of Chapter 6

B.1 Proofs of Section 6.3

Before proving the validity of Theorem 6.1 and Theorem 6.2, it is necessary to introduce some preliminary results.

The following proposition reminds an important property of asynchronous calculi: no behavior causally depends on the execution of output actions. Relation $\sim$ stands for the usual strong bisimulation relation (see e.g. Definition A.5.2).

Proposition B.1.1. *$P \xrightarrow{\overline{a}} P'$ implies $P \sim P' \,|\, \overline{a}$.*

PROOF: By observing that outputs are non-blocking actions, a suitable strong bisimulation can be defined. □

As direct consequences of the previous proposition, we get the results enunciated in the following lemma: (1) output actions can always be delayed and (2) a diamond property involving outputs.

Lemma B.1.1. *Let μ be a generic action ($\mu ::= \overline{b} \,|\, \theta$):*

(1) *$P \xrightarrow{\overline{a}}\xrightarrow{\mu} P'$ implies $P \xrightarrow{\mu}\xrightarrow{\overline{a}} P'$; similarly $P \xrightarrow{\overline{a}}\stackrel{\mu}{\Longrightarrow} P'$ implies $P \stackrel{\mu}{\Longrightarrow}\xrightarrow{\overline{a}} P'$;*

(2) *$P \xrightarrow{\overline{a}} P'$ and $P \xrightarrow{\mu} P''$ imply that there is a P''' such that $P' \xrightarrow{\mu} P'''$ and $P'' \xrightarrow{\overline{a}} P'''$; similarly $P \xrightarrow{\overline{a}} P'$ and $P \stackrel{\mu}{\Longrightarrow} P''$ imply that there is a P''' such that $P' \stackrel{\mu}{\Longrightarrow} P'''$ and $P'' \xrightarrow{\overline{a}} P'''$.*

PROOF: In both cases, the result follows by applying Proposition B.1.1. □

The following propositions enunciate two relevant properties of the hiding operator.

Proposition B.1.2. *$(P \,|\, \overline{a}) \setminus^n b \sim (P \setminus^n b \,|\, \overline{a})$ if $a \neq b$.*

PROOF: By Proposition 6.1 (HID), and definition of $\xrightarrow{\mu}$. □

Proposition B.1.3. $(P \,|\, \overline{a}) \setminus^n a \approx_{\mathsf{a}} P \setminus^{n+1} a$.

PROOF: It suffices to note that $(P \,|\, \overline{a}) \setminus^n a \xrightarrow{\tau} P \setminus^{n+1} a$, Proposition 6.1 (HIDAT). □

In the following propositions we prove that $\approx_{\mathsf{a}}$ and $\backsimeq$ are closed under contexts; as a consequence we obtain that both are congruences.

Proposition B.1.4. *$P \approx_{\mathsf{a}} R$ implies $\forall a :\ a.P \approx_{\mathsf{a}} a.R$.*

PROOF: It is enough to show that the relation $\mathcal{R} = \approx_{\mathsf{a}} \cup \{(a.P, a.R)\}$ is a weak asynchronous bisimulation. □

Proposition B.1.5. *$P \approx_{\mathsf{a}} R$ implies $\forall a : !a.P \approx_{\mathsf{a}} !a.R$.*

PROOF: It is enough to show that the relation

$$\mathcal{R} \;=\; \{((\prod_i P_i^{n_i} \,|\, !a.P), (\prod_i R_i^{n_i} \,|\, !a.R)) \;\big|\; n_i \geqslant 0,\ (P_i, R_i) \in \approx_{\mathsf{a}}\}$$

where P^n is a shorthand for the parallel composition of n copies of P and $\prod_i P_i$ stands for $P_1 | \cdots | P_n | \cdots$, is a weak asynchronous bisimulation up to $\sim$.

The proof proceeds as usual, by showing that every transition of the left term can be matched by a transition of the right one (and vice-versa), and the pair composed by the arrival processes is in $\mathcal{R}$. The proof is straightforward by a simple case analysis of transitions, as defined in Proposition 6.1. The most involved case is when a communication occurs between two subprocesses, let's say P_j and P_k. Suppose $P_j \xrightarrow{\{\!|a|\!\}} P_j'$ and $P_k \xrightarrow{\overline{a}} P_k'$. This means that, by Proposition 6.1 (COM):

$$(\prod_i P_i^{n_i} \,|\, !a.P) \xrightarrow{\tau} (\prod_{i \neq j,k} P_i^{n_i} \,|\, P_j^{n_j - 1} \,|\, P_k^{n_k - 1} \,|\, P_j' \,|\, P_k' \,|\, !a.P) = S_1 \ .$$

By $P_k \approx_{\mathsf{a}} R_k$ we know that $R_k \stackrel{\overline{a}}{\Rightarrow} R_k'$ with $P_k' \approx_{\mathsf{a}} R_k'$. We distinguish the following cases for R_j:

$R_j \xRightarrow{\{\!|a|\!\}} R_j'$**:** in this case $R_j' \approx_{\mathsf{a}} P_j'$ and, by Proposition 6.1 (COM):

$$(\prod_i R_i^{n_i} \,|\, !a.R) \stackrel{\tau}{\Rightarrow} (\prod_{i \neq j,k} R_i^{n_i} \,|\, R_j^{n_j - 1} \,|\, R_k^{n_k - 1} \,|\, R_j' \,|\, R_k' \,|\, !a.R) = S_2$$

and $(S_1, S_2) \in \mathcal{R}$ by definition of $\mathcal{R}$.

$R_j \stackrel{\theta}{\Rightarrow} R_j'$**:** this means that, by Proposition 6.1 (PAR):

$$(\prod_i R_i^{n_i} \,|\, !a.R) \stackrel{\theta}{\Rightarrow} (\prod_{i \neq j} R_i^{n_i} \,|\, R_j^{n_j - 1} \,|\, R_j' \,|\, !a.R) = S_2$$

and we have to show that $S_1 \,|\, \prod_{b \in \theta} \overline{b} \approx_{\mathsf{a}} S_2$. We distinguish two cases:

$a \in \theta$**:** from $P_j \approx_{\mathsf{a}} R_j$ we obtain that $P'_j \mid \prod_{b\in\theta\setminus a} \overline{b} \approx_{\mathsf{a}} R'_j$. Moreover, from $P'_k \approx_{\mathsf{a}} R'_k$, we have (by definition of $\mathcal{R}$):

$$(\textstyle\prod_{i\neq j,k} P_i^{n_i} \mid P_j^{n_j-1} \mid P_k^{n_k-1} \mid P'_j \mid \prod_{b\in\theta\setminus a} \overline{b} \mid P'_k \mid !a.P)$$
$$\mathcal{R}$$
$$(\textstyle\prod_{i\neq j,k} R_i^{n_i} \mid R_j^{n_j-1} \mid R_k^{n_k-1} \mid R'_j \mid R'_k \mid !a.R)$$

but $\overline{a} \approx_{\mathsf{a}} \overline{a}$, thus we also have (again by definition of $\mathcal{R}$)

$$(\textstyle\prod_{i\neq j,k} P_i^{n_i} \mid P_j^{n_j-1} \mid P_k^{n_k-1} \mid P'_j \mid \prod_{b\in\theta\setminus a} \overline{b} \mid P'_k \mid \overline{a} \mid !a.P)$$
$$\mathcal{R}$$
$$(\textstyle\prod_{i\neq j,k} R_i^{n_i} \mid R_j^{n_j-1} \mid R_k^{n_k-1} \mid R'_j \mid R'_k \mid \overline{a} \mid !a.R)$$

by Proposition B.1.1, $\overline{a} \mid R'_k \sim R_k$, thus

$$(\prod_{i\neq j,k} P_i^{n_i} | P_j^{n_j-1} | P_k^{n_k-1} | P'_j \mid \prod_{b\in\theta} \overline{b} \mid P'_k \mid !a.P)\mathcal{R} \sim (\prod_{i\neq j} R_i^{n_i} | R_j^{n_j-1} | R'_j \mid !a.R)$$

that is $(S_1 \mid \prod_{b\in\theta} \overline{b})\ \mathcal{R} \sim S_2$.

$a \notin \theta$**:** from $P_j \approx_{\mathsf{a}} R_j$ we obtain that $P'_j \mid \prod_{b\in\theta} \overline{b} \approx_{\mathsf{a}} R'_j \mid \overline{a}$. Moreover, from $P'_k \approx_{\mathsf{a}} R'_k$, we have (by definition of $\mathcal{R}$):

$$(\textstyle\prod_{i\neq j,k} P_i^{n_i} \mid P_j^{n_j-1} \mid P_k^{n_k-1} \mid P'_j \mid \prod_{b\in\theta} \overline{b} \mid P'_k \mid !a.P)$$
$$\mathcal{R}$$
$$(\textstyle\prod_{i\neq j,k} R_i^{n_i} \mid R_j^{n_j-1} \mid R_k^{n_k-1} \mid R'_j \mid \overline{a} \mid R'_k \mid !a.R)$$

by Proposition B.1.1, $\overline{a} \mid R'_k \sim R_k$, thus

$$(\prod_{i\neq j,k} P_i^{n_i} | P_j^{n_j-1} | P_k^{n_k-1} | P'_j | \prod_{b\in\theta} \overline{b} | P'_k | !a.P)\ \mathcal{R} \sim\ (\prod_{i\neq j} R_i^{n_i} | R_j^{n_j-1} | R'_j | !a.R)$$

that is $(S_1 \mid \prod_{b\in\theta} \overline{b})\ \mathcal{R} \sim S_2$.

□

Proposition B.1.6. *$P \approx_{\mathsf{a}} S$ implies $\forall R:\ P \mid R \approx_{\mathsf{a}} S \mid R$.*

Proof: The proof proceeds by showing that the relation

$$\mathcal{R} = \{(P \mid R, S \mid R) \mid (P, S) \in \approx_{\mathsf{a}}\}$$

is a weak asynchronous bisimulation up to $\sim$.

Suppose $P \mid R \xrightarrow{\mu} P''$; by applying Proposition 6.1, we can distinguish the following cases obtained by applying Proposition 6.1 (par) or (com):

$R \xrightarrow{\mu} R'$**:** $P'' = P \,|\, R'$; by Proposition 6.1 (PAR), $S \,|\, R \xrightarrow{\mu} S \,|\, R'$ and $(P \,|\, R')\mathcal{R}(S \,|\, R')$ by definition of $\mathcal{R}$;

$P \xrightarrow{\overline{a}} P'$**:** $\mu = \overline{a}$ and $P'' = P' \,|\, R$. By $P \approx_{\mathsf{a}} S$ we have $S \stackrel{\overline{a}}{\Rightarrow} S'$ with $P' \approx_{\mathsf{a}} S'$. By Proposition 6.1 (PAR), $S \,|\, R \stackrel{\overline{a}}{\Rightarrow} S' \,|\, R$ and $(P' \,|\, R)\mathcal{R}(S' \,|\, R)$ by definition of $\mathcal{R}$;

$P \xrightarrow{\theta} P'$**:** $\mu = \theta$ and $P'' = P' \,|\, R$. By $P \approx_{\mathsf{a}} S$ we have $S \stackrel{\theta'}{\Rightarrow} S'$ and $\big(P' \,|\, \prod_{a \in \theta' \setminus \theta} \overline{a}\big) \approx_{\mathsf{a}} \big(S' \,|\, \prod_{a \in \theta \setminus \theta'} \overline{a}\big)$.

By Proposition 6.1 (PAR), $S \,|\, R \stackrel{\theta'}{\Rightarrow} S' \,|\, R$ and $\big(P' \,|\, \prod_{a \in \theta' \setminus \theta} \overline{a} \,|\, R\big)\mathcal{R}\big(S' \,|\, \prod_{a \in \theta \setminus \theta'} \overline{a} \,|\, R\big)$ follows from $\big(P' \,|\, \prod_{a \in \theta' \setminus \theta} \overline{a}\big) \approx_{\mathsf{a}} \big(S' \,|\, \prod_{a \in \theta \setminus \theta'} \overline{a}\big)$ and definition of $\mathcal{R}$;

$P \xrightarrow{\overline{a}} P'$ **and** $R \xrightarrow{\{\!|a|\!\}} R'$**:** $\mu = \tau$ and $P'' = P' \,|\, R'$. $P \approx_{\mathsf{a}} S$ implies $S \stackrel{\overline{a}}{\Rightarrow} S'$ and $P' \approx_{\mathsf{a}} S'$. By Proposition 6.1 (COM), $S \,|\, R \Rightarrow S' \,|\, R'$ and, by definition of $\mathcal{R}$, $(P' \,|\, R')\mathcal{R}(S' \,|\, R')$;

$P \xrightarrow{\{\!|a|\!\}} P'$ **and** $R \xrightarrow{\overline{a}} R'$**:** $\mu = \tau$ and $P'' = P' \,|\, R'$. $P \approx_{\mathsf{a}} S$ implies that $S \stackrel{\theta}{\Rightarrow} S'$. We consider the following cases by distinguishing the possible values of θ:

$\theta = \{\!|a|\!\}$**:** in this case $P' \approx_{\mathsf{a}} S'$. By Proposition 6.1 (COM), $S \,|\, R \Rightarrow S' \,|\, R'$ and, by definition of $\mathcal{R}$, $(P' \,|\, R')\mathcal{R}(S' \,|\, R')$;

otherwise: $S \,|\, R \stackrel{\theta}{\Rightarrow} S' \,|\, R$ by Proposition 6.1 (PAR); we have to prove that $P' \,|\, R' \,|\, \prod_{b \in \theta} \overline{b} \mathcal{R} S' \,|\, R$. We distinguish the following cases:

$a \in \theta$**:** from $P \approx_{\mathsf{a}} S$ we obtain $P' \,|\, \prod_{b \in \theta \setminus a} \overline{b} \approx_{\mathsf{a}} S'$ and by definition of $\mathcal{R}$:

$$(P' \,|\, \prod_{b \in \theta \setminus a} \overline{b} \,|\, R) \;\mathcal{R}\; (S' \,|\, R)$$

and by Proposition B.1.1, $R \sim R' \,|\, \overline{a}$, thus

$$(P' \,|\, R' \,|\, \prod_{b \in \theta} \overline{b}) \;\sim \mathcal{R}\; (S' \,|\, R)\;;$$

$a \notin \theta$**:** from $P \approx_{\mathsf{a}} S$ we obtain $P' \,|\, \prod_{b \in \theta} \overline{b} \approx_{\mathsf{a}} S' \,|\, \overline{a}$, by definition of $\mathcal{R}$:

$$(P' \,|\, \prod_{b \in \theta} \overline{b} \,|\, R') \;\mathcal{R}\; (S' \,|\, \overline{a} \,|\, R')$$

and by Proposition B.1.1, $R \sim R' \,|\, \overline{a}$, thus

$$(P' \,|\, R' \,|\, \prod_{b \in \theta} \overline{b}) \;\mathcal{R} \sim\; (S' \,|\, R)\;.$$

□

In what follows, we denote $\xrightarrow{a^n}$ the relation $\xrightarrow{a} \cdots \xrightarrow{a}$, that is the composition of n copies of $\xrightarrow{a}$.

Proposition B.1.7. *$P \approx_{\mathsf{a}} R$ implies $\forall a, n \geqslant 0 :\ P \setminus^n a \approx_{\mathsf{a}} R \setminus^n a$.*

PROOF: The proof proceeds by showing that the relation:

$$\mathcal{R} = \{(P_i \setminus^{n+i} a, R_j \setminus^{n+j} a) \,|\, n \geqslant 0,\ (P,R) \in \approx_{\mathsf{a}},\ P \xrightarrow{\overline{a}^i} P_i,\ R \xrightarrow{\overline{a}^j} R_j\}$$

is a weak asynchronous bisimulation up to $\sim$. We distinguish the following cases:

(HID): $P_i \setminus^{n+i} a \xrightarrow{\mu} P_i' \setminus^{n+i} a$ is derived by $P_i \xrightarrow{\mu} P_i'$, if a does not appear in μ. By Lemma B.1.1 (1), $P \xrightarrow{\overline{a}^i} P_i \xrightarrow{\mu} P_i'$ implies $P \xrightarrow{\mu} P' \xrightarrow{\overline{a}^i} P_i'$. From $P \approx_{\mathsf{a}} R$ we obtain $R \stackrel{\mu}{\Rightarrow} R'$ with $P' \approx_{\mathsf{a}} R'$ and by $R \xrightarrow{\overline{a}^j} R_j$ and Lemma B.1.1 (2), $R' \xrightarrow{\overline{a}^j} R_j'$ and $R_j \stackrel{\mu}{\Rightarrow} R_j'$; by Proposition 6.1 (HID), $R_j \setminus^{n+j} a \stackrel{\mu}{\Rightarrow} R_j' \setminus^{n+j} a$. Finally, $(P_i' \setminus^{n+i} a)\mathcal{R}(R_j' \setminus^{n+j} a)$ because $P' \approx_{\mathsf{a}} R'$, $P' \xrightarrow{\overline{a}^i} P_i'$, $R' \xrightarrow{\overline{a}^j} R_j'$ and by definition of $\mathcal{R}$;

(HIDAT): $P_i \setminus^{n+i} a \xrightarrow{\theta} P_i' \setminus^{n'} a$ is derived by $P_i \xrightarrow{\theta'} P_i'$ with $\theta' = \theta \uplus a^m$ and $n' = n + i - m$. By Lemma B.1.1 (1), $P \xrightarrow{\overline{a}^i} P_i \xrightarrow{\theta'} P_i'$ implies $P \xrightarrow{\theta'} P' \xrightarrow{\overline{a}^i} P_i'$. By $P \approx_{\mathsf{a}} R$, $R \stackrel{\gamma'}{\Rightarrow} R'$ with $(P' \mid \prod_{b\in\gamma'\setminus\theta'} \overline{b}) \approx_{\mathsf{a}} (R' \mid \prod_{b\in\theta'\setminus\gamma'} \overline{b})$. Suppose $\gamma' = \gamma \uplus a^{m'}$ and, without loss of generality, that $m' > m$. We can rewrite $P' \mid \prod_{b\in\gamma'\setminus\theta'} \overline{b}$ as $P' \mid \overline{a}^{m'-m} \mid \prod_{b\in\gamma\setminus\theta} \overline{b}$ and $R' \mid \prod_{b\in\theta'\setminus\gamma'} \overline{b}$ as $R' \mid \prod_{b\in\theta\setminus\gamma} \overline{b}$, thus

$$(P' \mid \overline{a}^{m'-m} \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \approx_{\mathsf{a}} (R' \mid \prod_{b\in\theta\setminus\gamma} \overline{b})\ .$$

Moreover, by Lemma B.1.1 (2), $R \stackrel{\gamma'}{\Rightarrow} R'$ and $R \xrightarrow{\overline{a}^j} R_j$ imply $R_j \stackrel{\gamma'}{\Rightarrow} R_j'$ and $R' \xrightarrow{\overline{a}^j} R_j'$; by Proposition 6.1 (HIDAT), $R_j \setminus^{n+j} a \stackrel{\gamma}{\Rightarrow} R_j' \setminus^{n+j-m'} a$.
We have to relate the processes $P_i' \setminus^{n+i-m} a \mid \prod_{b\in\gamma\setminus\theta} \overline{b}$ and $R_j' \setminus^{n+j-m'} a \mid \prod_{b\in\theta\setminus\gamma} \overline{b}$. By Proposition 6.1 (HIDOUT), $(P' \mid \overline{a}^{m'-m} \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \setminus^{n-m'} a \xrightarrow{\tau} (P_i' \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \setminus^{n+i-m} a$ and $(R' \mid \prod_{b\in\theta\setminus\gamma} \overline{b}) \setminus^{n-m'} a \xrightarrow{\tau} (R_j' \mid \prod_{b\in\theta\setminus\gamma} \overline{b}) \setminus^{n+j-m'} a$; thus from $(P' \mid \overline{a}^{m'-m} \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \approx_{\mathsf{a}} (R' \mid \prod_{b\in\theta\setminus\gamma} \overline{b})$ we obtain

$$\big((P_i' \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \setminus^{n+i-m} a\big)\ \mathcal{R}\ \big((R_j' \mid \prod_{b\in\theta\setminus\gamma} \overline{b}) \setminus^{n+j-m'} a\big)$$

that is, by Proposition B.1.2, $(P_i' \setminus^{n+i-m} a \mid \prod_{b\in\gamma\setminus\theta} \overline{b}) \sim \mathcal{R} \sim (R_j' \setminus^{n+j-m'} a \mid \prod_{b\in\theta\setminus\gamma} \overline{b})$.

(HIDOUT): $P_i \setminus^{n+i} a \xrightarrow{\tau} P_i' \setminus^{n+i+1} a$ is derived by $P_i \xrightarrow{\overline{a}} P_i'$; $P_i' = P_{i+1}$ and by definition of $\mathcal{R}$ we have $(P_{i+1} \setminus^{n+i+1} a)\,\mathcal{R}\,(R_j \setminus^{n+j} a)$.

□

Proposition B.1.8. *Suppose $\alpha = rd(a)$ or $\alpha = wt(a)$. If $M \backsimeq N$ then $\alpha.M \backsimeq \alpha.N$.*

PROOF: Let be $\alpha = rd(a)$. For each σ such that $a \notin \sigma$ we have $(rd(a).M)_{\sigma;\epsilon} \rightarrow (retry)_{\sigma;\epsilon}$ and $(rd(a).N)_{\sigma;\epsilon} \rightarrow (retry)_{\sigma;\epsilon}$.

Moreover, for every σ such that $a \in \sigma$, by Definition 6.4 ($\backsimeq$), we have

- $(M)_{\sigma\setminus\{a\};\epsilon} \Rightarrow (retry)_{\sigma\setminus\{a\};\delta_M}$ and $(N)_{\sigma\setminus\{a\};\epsilon} \Rightarrow (retry)_{\sigma\setminus\{a\};\delta_N}$ imply $(rd(a).M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;rd(a).\delta_M}$ and $(rd(a).N)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;rd(a).\delta_N}$
- $(M)_{\sigma\setminus\{a\};\epsilon} \Rightarrow (end)_{\sigma\setminus\{a\};\delta_M}$, $(N)_{\sigma\setminus\{a\};\epsilon} \Rightarrow (end)_{\sigma\setminus\{a\};\delta_N}$ and $\delta_M =_{\sigma\setminus\{a\}} \delta_N$ imply $(rd(a).M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(a).\delta_M}$, $(rd(a).N)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(a).\delta_N}$ and $rd(a).\delta_M =_\sigma rd(a).\delta_N$.

Hence, by Definition 6.4 ($\backsimeq$), $\alpha.M \backsimeq \alpha.N$ □

Proposition B.1.9. *If $M_1 \backsimeq N_1$ and $M_2 \backsimeq N_2$ then M_1 orElse $M_2 \backsimeq N_1$ orElse N_2.*

PROOF: By Definition 6.4 ($\backsimeq$), $M_1 \backsimeq N_1$ implies

- for each σ such that $(M_1)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_M}$ then $(N_1)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_N}$ and $\delta_M =_\sigma \delta_N$ (and vice versa). Hence, by (AOE), $(M_1\ orElse\ M_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_M}$, $(N_1\ orElse\ N_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_N}$ and $\delta_M =_\sigma \delta_N$;
- for each σ such that $(M_1)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta}$ it holds that $(N_1)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta'}$; we distinguish two cases (recall that $M_2 \backsimeq N_2$):
  - if $(M_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_M}$ then $(N_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_N}$, $\delta_M =_\sigma \delta_N$ and, by (AOF): $(M_1\ orElse\ M_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_M}$ and $(N_1\ orElse\ N_2)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta_N}$ with $\delta_M =_\sigma \delta_N$;
  - if $(M_2)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta_M}$ then $(N_2)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta_N}$ and, again by (AOF): $(M_1\ orElse\ M_2)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta_M}$ with $(N_1\ orElse\ N_2)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta_N}$.

Hence, by Definition 6.4 ($\backsimeq$), $M_1\ orElse\ M_2 \backsimeq N_1\ orElse\ N_2$. □

We can now prove the main results of Section 6.3.

Theorem B.1.1 (Theorem 6.1). *Weak asynchronous bisimulation $\approx_{\mathsf{a}}$ is a congruence.*

PROOF: The result follows by Propositions B.1.4–B.1.7. □

Theorem B.1.2 (Theorem 6.2). *Weak atomic equivalence $\backsimeq$ is a congruence.*

PROOF: The result follows by Propositions B.1.8 and B.1.9. □

Proposition B.1.10 (Proposition 6.2). *$M \backsimeq N$ implies $atom(M) \approx_{\mathsf{a}} atom(N)$.*

PROOF: The proof proceeds by contradiction. Suppose that $M \backsimeq N$ and $atom(M) \not\approx_{\mathsf{a}} atom(N)$. This means that there is a δ such that $atom(M) \stackrel{\mathrm{RD}(\delta)}{\Longrightarrow} P$, with $P = \prod_{b \in \mathrm{WT}(\delta)} \overline{b}$, and for every δ' such that $atom(N) \stackrel{\mathrm{RD}(\delta')}{\Longrightarrow} R$, with $R = \prod_{b \in \mathrm{WT}(\delta')} \overline{b}$,

we have $(P \mid \prod_{b\in(\mathrm{RD}(\delta')\setminus\mathrm{RD}(\delta))} \overline{b}) \not\approx_{\mathsf{a}} (R \mid \prod_{b\in(\mathrm{RD}(\delta)\setminus\mathrm{RD}(\delta'))} \overline{b})$. This means that there is an a such that $(P \mid \prod_{b\in(\mathrm{RD}(\delta')\setminus\mathrm{RD}(\delta))} \overline{b}) \xrightarrow{\overline{a}}$ and $(R \mid \prod_{b\in(\mathrm{RD}(\delta)\setminus\mathrm{RD}(\delta'))} \overline{b}) \not\xrightarrow{\overline{a}}$ (or vice versa).

By rules (ATPASS) and (ATOK) and definition of $\xrightarrow{\mu}$, $atom(M) \xRightarrow{\mathrm{RD}(\delta)} P$ implies that there is a σ such that $(M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta} \xrightarrow{\mathrm{RD}(\delta)} P$. By definition of $\backsimeq$ there is a δ'' such that $(N)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta''}$, with $\delta =_\sigma \delta''$, that is $\sigma \setminus \mathrm{RD}(\delta) \uplus \mathrm{WT}(\delta) = \sigma \setminus \mathrm{RD}(\delta'') \uplus \mathrm{WT}(\delta'')$. Thus by rules (ATPASS) and (ATOK) and Proposition 6.1 $atom(N) \xRightarrow{\mathrm{RD}(\delta'')} R$ with $R = \prod_{b\in\mathrm{WT}(\delta'')} \overline{b}$.

Suppose $P = \prod_{b\in\mathrm{WT}(\delta)} \overline{b} \xrightarrow{\overline{a}}$; this means that $a \in \mathrm{WT}(\delta)$. From $\sigma \setminus \mathrm{RD}(\delta) \uplus \mathrm{WT}(\delta) = \sigma \setminus \mathrm{RD}(\delta'') \uplus \mathrm{WT}(\delta'')$ we obtain $\mathrm{WT}(\delta) = (\mathrm{WT}(\delta'') \uplus \mathrm{RD}(\delta)) \setminus \mathrm{RD}(\delta'')$, hence $a \in (\mathrm{WT}(\delta'') \uplus \mathrm{RD}(\delta)) \setminus \mathrm{RD}(\delta'')$ and either $R = \prod_{b\in\mathrm{WT}(\delta'')} \overline{b} \xrightarrow{\overline{a}}$ or $\prod_{b\in(\mathrm{RD}(\delta)\setminus\mathrm{RD}(\delta''))} \overline{b} \xrightarrow{\overline{a}}$.

Suppose $a \in (\mathrm{RD}(\delta'') \setminus \mathrm{RD}(\delta))$, then $\mathrm{WT}(\delta'') = (\mathrm{WT}(\delta) \uplus \mathrm{RD}(\delta'')) \setminus \mathrm{RD}(\delta)$ implies that $a \in \mathrm{WT}(\delta'')$, that is $R \xrightarrow{\overline{a}}$.

In both cases we have a contradiction because we have assumed that $(R \mid \prod_{b\in(\mathrm{RD}(\delta)\setminus\mathrm{RD}(\delta''))} \overline{b}) \not\xrightarrow{\overline{a}}$. Hence $atom(M) \approx_{\mathsf{a}} atom(N)$. □

B.2 Proofs of laws in Table 6.5

In this section we prove the correctness of laws in Table 6.5. In what follows $a \notin \sigma$ means that the name a does not appear in σ and $a^n \in \sigma$ means that σ contains n copies of a.

(COMM) $\alpha.\alpha'.M \backsimeq \alpha'.\alpha.M$: Suppose $\alpha = rd(a)$ and $\alpha' = rd(b)$ (the other cases are similar.) For each state σ we distinguish the following cases:

- $a, b \notin \sigma$**:** $(rd(a).\,rd(b).M)_{\sigma;\epsilon} \to (retry)_{\sigma;\epsilon}$ and $(rd(b).\,rd(a).M)_{\sigma;\epsilon} \to (retry)_{\sigma;\epsilon}$;
- $a^n, b^m \in \sigma$ **and** $(M)_{\sigma\setminus\{a,b\};\epsilon} \Rightarrow (end)_{\sigma\setminus\{a,b\};\delta}$**:** $(rd(a).\,rd(b).M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(a).\,rd(b).\delta}$, $(rd(b).\,rd(a).M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(b).\,rd(a).\delta}$ and $rd(a).\,rd(b).\delta =_\sigma rd(b).\,rd(a).\delta$;
- $a^n, b^m \in \sigma$ **and** $(M)_{\sigma\setminus\{a,b\};\epsilon} \Rightarrow (retry)_{\sigma\setminus\{a,b\};\delta}$**:** $(rd(a).\,rd(b).M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;rd(a).\,rd(b).\delta}$ and $(rd(b).\,rd(a).M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;rd(b).\,rd(a).\delta}$;
- $a \notin \sigma$ **and** $b^m \in \sigma$ **(or vice versa):** $(rd(a).\,rd(b).M)_{\sigma;\epsilon} \to (retry)_{\sigma;\epsilon}$ and $(rd(b).\,rd(a).M)_{\sigma;\epsilon} \to (rd(a).M)_{\sigma;rd(b)} \to (retry)_{\sigma;rd(b)}$.

Hence $\alpha.\alpha'.M \backsimeq \alpha'.\alpha.M$ by Definition 6.4 ($\backsimeq$).

(DIST) $\alpha.(M \;orElse\; N) \backsimeq (\alpha.M) \;orElse\; (\alpha.N)$: Suppose $M' = rd(a).(M \;orElse\; N)$ and $N' = (rd(a).M) \;orElse\; (rd(a).N)$. For each state σ we distinguish the following cases:

$a \notin \sigma$**:** $(rd(a).(M\ orElse\ N))_{\sigma;\epsilon} \rightarrow (retry)_{\sigma;\epsilon}$ and $((rd(a).M)\ orElse\ (rd(a).N))_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\epsilon}$, (ARDF) and (AOF);

$a^n \in \sigma$ **and** $(M)_{\sigma\setminus\{a\};\epsilon} \rightarrow (end)_{\sigma\setminus\{a\};\delta}$**:** $(rd(a).(M\ orElse\ N))_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(a).\delta}$ and $((rd(a).M)\ orElse\ (rd(a).N))_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;rd(a).\delta}$, in both cases by (ARDOK) and (AOE);

$a^n \in \sigma$ **and** $(M)_{\sigma\setminus\{a\};\epsilon} \rightarrow (retry)_{\sigma\setminus\{a\};\delta}$**:** $(rd(a).(M\ orElse\ N))_{\sigma;\epsilon} \Rightarrow (N')_{\sigma;rd(a).\delta}$ and $((rd(a).M)\ orElse\ (rd(a).N))_{\sigma;\epsilon} \Rightarrow (N')_{\sigma;rd(a).\delta}$, in both cases by (AOF), where $(N)_{\sigma\setminus\{a\};\epsilon} \Rightarrow (N')_{\sigma\setminus\{a\};\delta}$ with either $N' = end$ or $N' = retry$.

Hence $\alpha.(M\ orElse\ N) \backsimeq (\alpha.M)\ orElse\ (\alpha.N)$ by Definition 6.4 ($\backsimeq$).

(ASS) $M_1\ orElse\ (M_2\ orElse\ M_3) \backsimeq (M_1\ orElse\ M_2)\ orElse\ M_3$ follows by recalling that *orElse* is a left preemptive operator ((AOF) and (AOE)).

(IDEM) $M\ orElse\ M \backsimeq M$ can be shown by observing that $(M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta}$ implies $(M\ orElse\ M)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta}$, by (AOE), and $(M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta}$ implies $(M\ orElse\ M)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta}$, by (AOF).

(ABSRT1) $\alpha.retry \backsimeq retry$. Suppose $\alpha = rd(a)$. The result follows by observing that either if $a \in \sigma$ or $a \notin \sigma$ $(rd(a).retry)_{\sigma;\epsilon} \Rightarrow (retry)_{\sigma;\delta}$ with either $\delta = rd(a)$ or $\delta = \epsilon$.

(ABSRT2) $retry\ orElse\ M \backsimeq M \backsimeq M\ orElse\ retry$ follows by (AOF) and left preemption of *orElse*.

(ABSEND) $end\ orElse\ M \backsimeq end$ follows by (AOE).

(ASY) $a.\overline{a} \approx_{\mathsf{a}} \mathbf{0}$:

$$\mathcal{R} = \left\{(a.\overline{a}, \mathbf{0}),\ (\overline{a}, \overline{a}),\ (\mathbf{0}, \mathbf{0})\right\}\ .$$

(A-ASY) $atom(rd(a).wt(a).end) \approx_{\mathsf{a}} \mathbf{0}$:

$$\begin{aligned}\mathcal{R} = &\left\{(atom(rd(a).wt(a).end), \mathbf{0}),\ (\{\!|(rd(a).wt(a).end)_{\sigma;\epsilon}|\!\}_{rd(a).wt(a).end}, \mathbf{0})\right\}\\ &\cup \left\{(\{\!|(wt(a).end)_{\sigma;rd(a)}|\!\}_{rd(a).wt(a).end}, \mathbf{0}),\right.\\ &\qquad \left.(\{\!|(end)_{\sigma;rd(a).wt(a)}|\!\}_{rd(a).wt(a).end}, \mathbf{0}) \middle| a^n \in \sigma, n > 0\right\}\\ &\cup \left\{(\{\!|(retry)_{\sigma;\epsilon}|\!\}_{rd(a).wt(a).end}, \mathbf{0}) \middle| a \notin \sigma\right\} \cup \left\{(\overline{a}, \overline{a}),\ (\mathbf{0}, \mathbf{0})\right\}\ .\end{aligned}$$

(A-1) $atom(rd(a).end) \approx_{\mathsf{a}} a$:

$$\begin{aligned}\mathcal{R} \ = \ &\left\{(atom(rd(a).end), a),\ (\{\!|(rd(a).end)_{\sigma;\epsilon}|\!\}_{rd(a).end}, a)\right\}\\ \cup \ &\left\{(\{\!|(end)_{\sigma;rd(a)}|\!\}_{rd(a).end}, a),\ (\mathbf{0}, \mathbf{0}) \middle| a^n \in \sigma, n > 0\right\}\\ \cup \ &\left\{(\{\!|(retry)_{\sigma;\epsilon}|\!\}_{rd(a).end}, a) \middle| a \notin \sigma\right\}\ .\end{aligned}$$

B.3 Proof of Proposition 6.3

In this section we show that laws in Table 6.5 can be used for eliminating redundant branches from an atomic expression and obtaining an equivalent expression in normal form (see proof of Proposition 6.3.) Some preliminary results are needed.

The next proposition states that if K''s reads include K's then K' is bigger than K in our weak atomic preorder.

Proposition B.3.1. *Suppose* $K = \alpha_1.\cdots.\alpha_n$ *and* $K' = \beta_1.\cdots.\beta_m$. *If* $\text{RD}(K) \subseteq \text{RD}(K')$ *then* $K \sqsupseteq K'$.

PROOF: It is enough to observe that if $(K')_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta}$ then $\text{RD}(K') \subseteq \sigma$ (rules (ARDOK) and (ARDF)); thus $\text{RD}(K) \subseteq \sigma$, and by (ARDOK) we get $(K)_{\sigma;\epsilon} \Rightarrow (end)_{\sigma;\delta'}$. □

As a consequence of the previous proposition, we obtain that, in an *orElse* expression, a redundant branch, that is a branch which includes the reads of at least one of its preceding branches, can be eliminated.

Proposition B.3.2. *Consider the expressions* $K_1, \ldots, K_n$ *where, for* $i = 1, \ldots, n$, K_i *is of the form* $\alpha_{i_1}.\cdots.\alpha_{i_{n_i}}$. *If* $\text{RD}(K_j) \subseteq \text{RD}(K_n)$, *for a* j *such that* $0 < j < n$, *then*

$$K_1 \ orElse \ \cdots \ orElse \ K_{n-1} \ orElse \ K_n \backsimeq K_1 \ orElse \ \cdots \ orElse \ K_{n-1} \ .$$

PROOF: The proof proceeds by applying Proposition B.3.1, the fact that $M \sqcup M' \backsimeq M$ if and only if $M \sqsupseteq M'$ (see pag. 145) and *orElse* 's rules in Table 6.4. □

As previously said, the proof of the following theorem show how to apply rules in Table 6.5 for rearranging an atomic expression into an equivalent one in normal form.

Proposition B.3.3 (Proposition 6.3). *For every expression* M *there is an expression* M' *in normal form such that* $M \backsimeq M'$.

PROOF: The proof proceeds by induction on the structure of M:

$M = end$**:** $M' = M = end$;

$M = retry$**:** $M' = M = retry$;

$M = \alpha.N$**:** by induction hypothesis, there is an N' in normal form such that $N \backsimeq N'$. By Proposition B.1.8, $\alpha.N \backsimeq \alpha.N'$. Rule (DIST) can be applied for distributing α among the *orElse*s in N' in such a manner to obtain an M' in normal-form such that $M \backsimeq M'$;

$M = N \ orElse \ N'$**:** by induction hypothesis, there are N_0 and N_0', in normal form, such that $N \backsimeq N_0$ and $N' \backsimeq N_0'$. By Proposition B.1.9, $M = N \ orElse \ N' \backsimeq N_0 \ orElse \ N_0'$. We choose M' by considering the following cases:

- if $N_0 = \mathit{retry}$ we choose $M' = N'_0$, because, by (ABSRT), $\mathit{retry}\ \mathit{orElse}\ N'_0 \backsimeq N'_0$;
- if $N_0 = N_{0_1}\ \mathit{orElse}\ \ldots\ \mathit{orElse}\ N_{0_n}$ and $N'_0 = N'_{0_1}\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_m}$, consider $I = \{j \mid k \in \{1,\ldots,n\} : \mathrm{RD}(N_{0_k}) \subseteq \mathrm{RD}(N'_{0_j})\}$. If $I = \emptyset$ this means that $M' = N_0\ \mathit{orElse}\ N'_0$ is in normal form.

  Otherwise, suppose $I = \{j_1,\ldots,j_l\}$ with $j_i < j_w$ for $i < w$; by applying Proposition B.3.2, B.1.9 and (ASS) at every step, we have

$$
\begin{array}{ll}
& N_0\ \mathit{orElse}\ N'_0 \\
\backsimeq & \text{(by removing } N'_{0_{j_1}}\text{)} \\
& N_0\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_{j_1-1}}\ \mathit{orElse}\ N'_{0_{j_1+1}}\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_m} \\
\backsimeq & \text{(by removing } N'_{0_{j_2}}\text{)} \\
& N_0\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_{j_1-1}}\ \mathit{orElse}\ N'_{0_{j_1+1}}\ \mathit{orElse}\ \cdots \\
& \qquad \mathit{orElse}\ N'_{0_{j_2-1}}\ \mathit{orElse}\ N'_{0_{j_2+1}}\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_m} \\
\backsimeq & \text{(by removing } N'_{0_{j_3}}\text{)} \\
& \vdots \\
\backsimeq & \text{(by removing } N'_{0_{j_l}}\text{)} \\
& N_0\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_{j_1-1}}\ \mathit{orElse}\ N'_{0_{j_1+1}}\ \mathit{orElse}\ \cdots \\
& \qquad \mathit{orElse}\ N'_{0_{j_2-1}}\ \mathit{orElse}\ N'_{0_{j_2+1}}\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_{j_l-1}} \\
& \qquad \mathit{orElse}\ N'_{0_{j_l+1}}\ \mathit{orElse}\ \cdots\ \mathit{orElse}\ N'_{0_m} \\
= & M' \text{ (that is in normal form.)}
\end{array}
$$

In each case, $N_0\ \mathit{orElse}\ N'_0 \backsimeq M'$, thus $M \backsimeq M'$.

□

B.4 Proofs of Section 6.4

Lemma B.4.1 (Lemma 6.1). *Assume that $s' \preccurlyeq s$ and $P \stackrel{\overline{s}}{\Rightarrow} P'$, then there is a process P'' such that $P \stackrel{\overline{s'}}{\Rightarrow} P''$.*

PROOF: $s' \preccurlyeq s$ means $s' \preccurlyeq_0^n s$, for some $n \geqslant 0$. The proof proceeds by induction on n. For $n = 0$ we have $s = s'$. Suppose $n > 0$ and $s' \preccurlyeq_0^{n-1} s'' \preccurlyeq_0 s$. The result follows by induction hypothesis if we show that $P \stackrel{\overline{s''}}{\Longrightarrow}$. We proceed by distinguishing the possible cases for $s'' \preccurlyeq_0 s$ according to laws (TO1)-(TO4).

(TO1): $s'' = rr'$ and $s = r\{\!|a|\!\}r'$, thus $\overline{s}'' = \overline{r}\overline{r}'$ and $\overline{s} = \overline{r}\overline{a}\overline{r}'$. $P \stackrel{\overline{s}}{\Rightarrow} P'$ implies $P \stackrel{\overline{r}}{\Rightarrow} P_1 \stackrel{\overline{a}}{\Rightarrow} P_2 \stackrel{\overline{r}'}{\Rightarrow} P'$, and by Proposition B.1.1, $P_1 \sim P_2 \,|\, \overline{a}$, that is $P \stackrel{\overline{r}}{\Rightarrow} P_2 \,|\, \overline{a} \stackrel{\overline{r}'}{\Rightarrow} P' \,|\, \overline{a}$, hence $P \stackrel{\overline{s}''}{\Longrightarrow} P' \,|\, \overline{a}$;

(TO2): $s'' = rl\{\!|a|\!\}r'$ and $s = r\{\!|a|\!\}lr'$, thus $\overline{s}'' = \overline{r}\overline{l}\overline{a}\overline{r}'$ and $\overline{s} = \overline{r}\overline{a}\overline{l}\overline{r}'$. $P \stackrel{\overline{s}}{\Rightarrow} P'$ implies $P \stackrel{\overline{r}}{\Rightarrow} P_1 \stackrel{\overline{a}}{\Rightarrow} P_2 \stackrel{\overline{l}}{\Rightarrow} P_3 \stackrel{\overline{r}'}{\Rightarrow} P'$, and by Proposition B.1.1, $P_1 \sim P_2 \,|\, \overline{a}$, that is $P \stackrel{\overline{r}}{\Rightarrow} P_2 \,|\, \overline{a} \stackrel{\overline{l}}{\Rightarrow} P_3 \,|\, \overline{a} \stackrel{\overline{a}}{\Rightarrow} P_3 \stackrel{\overline{r}'}{\Rightarrow} P'$, hence $P \stackrel{\overline{s}''}{\Longrightarrow} P'$;

(TO3): $s'' = rr'$ and $s = r\{\!|a|\!\}\overline{a}r'$, thus $\overline{s}'' = \overline{r}\overline{r}'$ and $\overline{s} = \overline{r}\overline{a}\{\!|a|\!\}\overline{r}'$. $P \stackrel{\overline{s}}{\Rightarrow} P'$ implies $P \stackrel{\overline{r}}{\Rightarrow} P_1 \stackrel{\overline{a}}{\Rightarrow} P_2 \stackrel{a}{\Rightarrow} P_3 \stackrel{\overline{r}'}{\Rightarrow} P'$, hence, by Proposition B.1.1, $P_1 \sim P_2 \,|\, \overline{a}$, that is P_2 can synchronize with $\overline{a}$ (Proposition 6.1 (COM)) and $P \stackrel{\overline{r}}{\Rightarrow} P_2 \,|\, \overline{a} \Rightarrow P_3 \stackrel{\overline{r}'}{\Rightarrow} P'$, that is $P \stackrel{\overline{s}''}{\Longrightarrow} P'$;

(TO4): $s'' = \{\!|a_1|\!\} \cdots \{\!|a_n|\!\}$ and $s = \{\!|a_1, \ldots, a_n|\!\}$, or vice versa; in this case $\overline{s} = \overline{s}''$ by definition of $\bar{\cdot}$.

□

Lemma B.4.2 (Lemma 6.2). *Consider two traces s and r. If there is a process R such that $\mathcal{O}(s) \stackrel{\overline{r}}{\Rightarrow} \stackrel{\overline{w}}{\Longrightarrow} R$ then $r \preccurlyeq s$.*

PROOF: The proof proceeds by induction on s.

$s = \overline{a}s'$**:** $\mathcal{O}(s) = a.\mathcal{O}(s')$ and $\mathcal{O}(s) \stackrel{\overline{r}\overline{w}}{\Longrightarrow}$ implies $\overline{r} = \{\!|a|\!\}\overline{r}'$ such that $\mathcal{O}(s) \xrightarrow{\{\!|a|\!\}} \mathcal{O}(s') \stackrel{\overline{r}'}{\Longrightarrow}$. By induction hypothesis, $r' \preccurlyeq s'$, hence by prefixing, $r = \overline{a}r' \preccurlyeq \overline{a}s' = s$;

$s = \{\!|a_1, \ldots, a_n|\!\}s'$**:** $\mathcal{O}(s) = \left(\prod_{a \in \{a_1, \ldots, a_n\}} \overline{a}\right) \,|\, \mathcal{O}(s')$. We have $\mathcal{O}(s) \stackrel{\overline{r}\overline{w}}{\Longrightarrow}$, we can distinguish the following cases depending on $\overline{r}$:

$\overline{a}_i \notin \overline{r}$**:** by induction hypothesis, $\mathcal{O}(s') \stackrel{\overline{r}\overline{w}}{\Longrightarrow}$ implies $r \preccurlyeq s'$ and by (TO1) and (TO4), $r \preccurlyeq s' \preccurlyeq \{\!|a_1|\!\} \cdots \{\!|a_n|\!\}s' \; {}_0\!\succcurlyeq\!\preccurlyeq_0 \; \{\!|a_1, \ldots, a_n|\!\}s' = s$;

$\overline{a}_{i_1}, \ldots \overline{a}_{i_k} \in \overline{r}$ **for** $\{a_{i_1}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}$**:** in this case $\overline{r} = \overline{r}_1\overline{a}_{i_1} \cdots \overline{r}_k\overline{a}_{i_k}\overline{r}_{k+1}$ and $\mathcal{O}(s') \stackrel{\overline{r}_1 \cdots \overline{r}_{k+1}\overline{w}}{\Longrightarrow}$. By induction hypothesis, $r_1 \cdots r_{k+1} \preccurlyeq s'$:

$$
\begin{array}{rcll}
r & = & r_1\{\!|a_{i_1}|\!\} \cdots r_k\{\!|a_{i_k}|\!\}r_{k+1} & \\
 & \preccurlyeq & \{\!|a_{i_1}|\!\} \cdots \{\!|a_{i_k}|\!\}r_1 \cdots r_{k+1} & \text{(by (TO2))} \\
 & \preccurlyeq & \{\!|a_{i_1}|\!\} \cdots \{\!|a_{i_k}|\!\}s' & \text{(by induction and prefixing)} \\
 & \preccurlyeq & \{\!|a_1|\!\} \cdots \{\!|a_n|\!\}s' & \text{(by (TO1) and (TO2))} \\
 & {}_0\!\succcurlyeq\!\preccurlyeq_0 & \{\!|a_1, \cdots, a_n|\!\}s' & \text{(by (TO4))} \\
 & = & s; &
\end{array}
$$

$\overline{r} = \overline{r}_1 \cdots \overline{r}_{k+1}$ **and** $\mathcal{O}(s') \xrightarrow{\overline{r}_1\{\!|a_{i_1}|\!\}\cdots\overline{r}_k\{\!|a_{i_k}|\!\}r_{k+1}} \Rightarrow$ **for** $\{a_{i_1}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}$**:**
by induction hypothesis, $r_1\overline{a}_{i_1} \cdots r_k\overline{a}_{i_k}r_{k+1} \preccurlyeq s'$ and:

$$
\begin{array}{rcll}
r & = & r_1 \cdots r_k & \\
 & \preccurlyeq & r_1\{\!|a_{i_1}|\!\}\overline{a}_{i_1} \cdots r_k\{\!|a_{i_k}|\!\}\overline{a}_{i_k}r_{k+1} & \text{(by (TO3))} \\
 & \preccurlyeq & \{\!|a_{i_1}|\!\} \cdots \{\!|a_{i_k}|\!\}r_1\overline{a}_{i_1} \cdots r_k\overline{a}_{i_k}r_{k+1} & \text{(by (TO2))} \\
 & \preccurlyeq & \{\!|a_{i_1}|\!\} \cdots \{\!|a_{i_k}|\!\}s' & \text{(by induction)} \\
 & \preccurlyeq & \{\!|a_1|\!\} \cdots \{\!|a_n|\!\}s' & \text{(by (TO1) and (TO2))} \\
 & {}_0\succcurlyeq\preccurlyeq_0 & \{\!|a_1, \ldots, a_n|\!\}s' & \text{(by (TO4))} \\
 & = & s. &
\end{array}
$$

□

The proof of the full-abstraction theorem is standard (see e.g. [30]).

Theorem B.4.1 (Theorem 6.3). *For all processes P and R, $P \sqsubseteq_{may} R$ if and only if $P \ll_{may} R$.*

PROOF:

(⇒): Suppose $P \ll_{may} R$ and P *may* O for any observer O we have to show that R *may* O. P *may* O means that $P \,|\, O \stackrel{\overline{w}}{\Rightarrow}$, that is there exists a trace s such that $P \stackrel{s}{\Rightarrow}$ and $O \stackrel{\overline{sw}}{\Longrightarrow}$. $P \ll_{may} R$ implies that there exists $s' \preccurlyeq s$ such that $R \stackrel{s'}{\Rightarrow}$. $s' \preccurlyeq s$ implies $s'w \preccurlyeq sw$. By Lemma B.4.1 and $O \stackrel{\overline{sw}}{\Longrightarrow}$ we get that $O \stackrel{\overline{s'w}}{\Longrightarrow}$. Hence, from $R \stackrel{s'}{\Rightarrow}$ we obtain $R \,|\, O \stackrel{\overline{w}}{\Rightarrow}$, that is R *may* O ($P \sqsubseteq_{may} R$).

(⇐): Suppose $P \sqsubseteq_{may} R$ and $P \stackrel{s}{\Rightarrow}$, we have to show that there exists $s' \preccurlyeq s$ such that $R \stackrel{s'}{\Rightarrow}$. From $P \stackrel{s}{\Rightarrow}$ and $\mathcal{O}(s) \stackrel{\overline{sw}}{\Longrightarrow}$ we have $P \,|\, \mathcal{O}(s) \stackrel{\overline{w}}{\Rightarrow}$, that is P *may* $\mathcal{O}(s)$. It follows that R *may* $\mathcal{O}(s)$, that is $R \,|\, \mathcal{O}(s) \stackrel{\overline{w}}{\Rightarrow}$. Thus, there exists s' such that $R \stackrel{s'}{\Rightarrow}$ and $\mathcal{O}(s) \stackrel{\overline{s'w}}{\Longrightarrow}$, and, by Lemma B.4.2 and $\mathcal{O}(s) \stackrel{\overline{s'w}}{\Longrightarrow}$ we have $s' \preccurlyeq s$, that is $P \ll_{may} R$.

□

Lemma B.4.3 (Lemma 6.3). *Assume $M = \bigsqcup_{i=1,\ldots,n} K_i$ is an expression in normal form. For every index i in $\{1, \ldots, n\}$ we have $atom(M)\,;\sigma_i \rightarrow^* \{\!|(end)_{\sigma_i;\delta}|\!\}_M\,;\sigma_i$ where $\sigma_i = \text{RD}(K_i) = \text{RD}(\delta)$ and $\text{WT}(\delta) = \text{WT}(K_i)$.*

PROOF: By definition of normal form. □

Corollary B.4.1. *Assume $M = \bigsqcup_{i=1,\ldots,n} K_i$ is an expression in normal form. The possible behavior of $atom(M)$ can be described as $atom(M) \xrightarrow{\text{RD}(K_i)} \prod_{b \in \text{WT}(K_i)} \overline{b}$ for every $i \in 1, \ldots, n$.*

PROOF: By Lemma B.4.3, rule (ATOK) and definition of $\xrightarrow{\mu}$. □

We can prove now the main result of Section 6.4, that is that may-testing semantics is not able to distinguish the behaviour of an atomic expression from the behaviour of the corresponding CCS process.

Theorem B.4.2 (Theorem 6.4). *For every expression M in normal form we have $atom(M) \simeq_{may} [\![M]\!]$.*

PROOF: The proof proceeds by using the alternative preorder instead of the may preorder; in what follows it is shown that:

(1) $atom(M) \ll_{may} [\![M]\!]$;
(2) $[\![M]\!] \ll_{may} atom(M)$.

Recall that M is in normal-form, thus $M = \bigsqcup_{i=1,\ldots,n} K_i$ and $[\![M]\!] = \sum_{i=1,\ldots,n} [\![K_i]\!]$.

(1) For proving that $atom(M) \ll_{may} [\![M]\!]$, we have to show that $\forall s$ such that $atom(M) \stackrel{s}{\Rightarrow}$ there exists $s' \preccurlyeq s$ such that $[\![M]\!] \stackrel{s'}{\Rightarrow}$. We distinguish the following cases for s:

$s = \epsilon$**:** in this case we can choose $s' = \epsilon$;

$s = \theta\overline{a}_{i_1} \cdots \overline{a}_{i_l}$ **with** $l \geqslant 0$**:** by Corollary B.4.1, there is a $j \in \{1, \ldots, n\}$ such that $\theta = \text{RD}(K_j)$,

$$atom(M) \xrightarrow{\text{RD}(K_j)} \overline{a}_1 \mid \cdots \mid \overline{a}_m \xrightarrow{\overline{a}_{i_1} \cdots \overline{a}_{i_l}}$$

with $\{a_{i_1}, \ldots, a_{i_l}\} \subseteq \{a_1, \ldots, a_m\} = \text{WT}(K_j)$.
Suppose $\text{RD}(K_j) = \{b_1, \ldots, b_k\}$. By definition, $[\![K_j]\!] = b_1.\cdots.b_k.(\overline{a}_1 \mid \cdots \mid \overline{a}_m)$. That is, if we choose the j-th addend of $[\![M]\!]$, we have $[\![M]\!] \stackrel{s'}{\Rightarrow}$ with $s' = \{\!|b_1|\!\} \cdots \{\!|b_k|\!\}\overline{a}_{i_1} \cdots \overline{a}_{i_l}$, and by (TO4) ${s'}_0 \succcurlyeq \preccurlyeq_0 s$;

(2) For proving that $[\![M]\!] \ll_{may} atom(M)$, we have to show that $\forall s$ such that $[\![M]\!] \stackrel{s}{\Rightarrow}$ there exists $s' \preccurlyeq s$ such that $atom(M) \stackrel{s'}{\Rightarrow}$. We distinguish the following cases for s:

$s = \{\!|b_1|\!\} \cdots \{\!|b_k|\!\}$**:** s contains only input actions, thus we can choose $s' = \epsilon \preccurlyeq s$, (TO1), and $atom(M) \stackrel{s'}{\Rightarrow}$;

$s = \{\!|b_1|\!\} \cdots \{\!|b_k|\!\}\overline{a}_1 \cdots \overline{a}_m$ **with** $m > 0$**:** in this case there is a $j \in \{1, \ldots, n\}$ such that $[\![K_j]\!] \stackrel{s}{\Rightarrow}$, $\{b_1, \ldots, b_k\} = \text{RD}(K_j)$ and $\{a_1, \ldots, a_m\} \subseteq \text{WT}(K_j)$ (by definition of $[\![\cdot]\!]$). Suppose $\sigma = \text{RD}(K_j)$, by Lemma B.4.3, $atom(M); \sigma \Rightarrow \{\!|(end)_{\sigma;\delta}|\!\}_M$ with $\text{RD}(\delta) = \text{RD}(K_j)$ and $\text{WT}(\delta) = \text{WT}(K_j)$. This means

that $atom(M) \xrightarrow{\text{RD}(K_j)} \prod_{a \in \text{WT}(K_j)} \overline{a}$, that is (by (TO4)) there is an $s' = \text{RD}(K_j)\overline{a}_1 \cdots \overline{a}_m \; 0 \succcurlyeq \preccurlyeq 0 \; \{\!|b_1|\!\} \cdots \{\!|b_k|\!\} \overline{a}_1 \cdots \overline{a}_m = s$ such that $atom(M) \stackrel{s'}{\Longrightarrow}$.

□