A model checking approach for verifying COWS specifications

A Model Checking Approach for Verifying COWS Specifications*

Alessandro Fantechi[1], Stefania Gnesi[2], Alessandro Lapadula[1], Franco Mazzanti[2], Rosario Pugliese[1], and Francesco Tiezzi[1]

[1] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
[2] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", ISTI - CNR, Pisa

Abstract. We introduce a logical verification framework for checking functional properties of service-oriented applications formally specified using the service specification language COWS. The properties are described by means of SocL, a logic specifically designed to capture peculiar aspects of services. Service behaviours are abstracted in terms of Doubly Labelled Transition Systems, which are used as the interpretation domain for SocL formulae. We also illustrate the SocL model checker at work on a bank service scenario specified in COWS.

1 Introduction

Service-oriented computing (SOC) is an emerging paradigm for developing loosely coupled, interoperable, evolvable applications, which exploits the pervasiveness of the Internet and its related technologies. SOC systems deliver application functionality as services to either end-user applications or other services. Current software engineering technologies for SOC, however, remain at the descriptive level and do not support analytical tools for checking that services enjoy desirable properties and do not manifest unexpected behaviors. On the other end, logics have been since long proved able to reason about such complex software systems as SOC applications, because they only provide abstract specifications of these systems and can thus be used for describing system properties rather than system behaviours. Indeed, in the last twenty years, several modal, temporal and, more recently, spatial logics have been proposed as suitable means for specifying properties of concurrent and distributed systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc.

In this paper, we introduce a logical verification framework for checking functional properties of services by abstracting away from the computational contexts in which they are operating. In what follows, services are abstractly considered as entities capable of accepting requests, delivering corresponding responses and, on-demand, cancelling requests. Thus, we will say that a service is

1. *available:* if it is always capable to accept a request.
2. *reliable:* if, when a request is accepted, a final successful response is guaranteed.
3. *responsive:* if it always guarantees a response to each received request.
4. *broken:* if, after accepting a request, it does not provide the (expected) response.

* This work has been partially funded by the EU project SENSORIA (IST-2005-016004).

J. Fiadeiro and P. Inverardi (Eds.): FASE 2008, LNCS 4961, pp. 230–245, 2008.

5. *unavailable:* if it refuses all requests.
6. *fair:* if it is possible to cancel a request before the response.
7. *non-ambiguous:* if, after accepting a request, it provides no more than one response.
8. *sequential:* if, after accepting a request, no other requests may be accepted before giving a response.
9. *asynchronous:* if, after accepting a request, other requests may be accepted before giving a response.
10. *non-persistent:* if, after accepting a request, no other requests can be accepted.

Albeit not exhaustive, this list contains many common properties that express desirable attributes of services and SOC applications (see, e.g., the SENSORIA ontology [5] or [2]).

To formalize the properties above, we introduce SocL, a logic specifically designed to capture peculiar aspects of services. SocL is a variant of the logic UCTL [3], originally introduced to express properties of UML statecharts. UCTL and SocL have many commonalities: they share the same temporal logic operators, they are both state and event based branching-time logics, they are both interpreted on Doubly Labelled Transition Systems (L^2TSs, [9]) by exploiting the same on-the-fly model-checking engine. The two logics mainly differ for the syntax and semantics of state-predicates and action-formulae, and for the fact that SocL also permits to specify parametric formulae.

As specification language for the services and SOC applications of interest we use COWS (*Calculus for Orchestration of Web Services*, [14]), a recently proposed process calculus for specifying and combining services, while modelling their dynamic behaviour. The design of the calculus has been influenced by the principles underlying WS-BPEL [16], an OASIS standard language for orchestration of web services, and in fact COWS supports service instances with shared states, allows a same process to play more than one partner role and permits programming stateful sessions by correlating different service interactions. COWS has also taken advantage of previous work on process calculi. Indeed, it combines in an original way constructs and features borrowed from well-known process calculi, e.g. not-binding input activities, asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

To check if a COWS term enjoys some abstract properties expressed as SocL formulae, the following four steps must be performed. Firstly, the semantics of the COWS term is defined by using a Labelled Transition System (LTS). Secondly, this LTS is transformed into an L^2TS by labelling each state with the set of actions the COWS term is able to perform immediately from that state. Thirdly, by applying a set of application-dependent abstraction rules over the actions, the concrete L^2TS is abstracted into a simpler L^2TS. Finally, the SocL formulae are checked over this abstract L^2TS. To assist the verification process, we have developed CMC, an on-the-fly model checker for SocL formulae over L^2TS.

The rest of the paper is organized as follows. Section 2 introduces SocL, while Section 3 presents syntax and main features of COWS; this is done in a step-by-step fashion while modelling a bank service scenario, used for illustration purposes in the rest of the paper. Section 4 demonstrates how to transform the original LTS of a COWS term into an abstract L^2TS by using suitable abstraction rules. Section 5 presents CMC and illustrates the results of the verification of the bank service scenario. Section 6 touches upon related work and directions for future works.

2 The Logic SocL

In this section, we introduce the *action* and *state-based* branching time temporal logic SocL that is interpreted over L^2TSs [9]. SocL combines the action paradigm, classically used to describe systems via LTS, with predicates that are true over states, as usually exploited when using Kripke structures as semantic model. The advantage of *action and state-based* logics lies in the ease of expressiveness of properties that in pure action-based or pure state-based logics can be quite cumbersome to write down. Indeed in recent years, several logics that allow one to express both action-based and state-based properties have been introduced, for many different purposes (see for example [3,7,8,6,13]).

Before presenting the syntax of SocL, we report some basic definitions and notations used in the sequel.

Definition 1 (Doubly Labelled Transition System, L^2TS). *An* L^2TS *is a tuple* $\langle Q, q_0, Act, R, AP, L\rangle$*, where:*

- Q *is a set of states;*
- $q_0 \in Q$ *is the initial state;*
- Act *is a finite set of observable events (actions) with* α *ranging over* 2^{Act} *and* ϵ *denoting the empty set;*
- $R \subseteq Q \times 2^{Act} \times Q$ *is the transition relation*[1]*; instead of* $(q, \alpha, q') \in R$ *we may also write* $q \xrightarrow{\alpha} q'$.
- AP *is a set of atomic propositions with* π *ranging over* AP*;*
- $L : Q \longrightarrow 2^{AP}$ *is a labelling function that maps each state in* Q *to a subset of* AP.

Basically, an L^2TS is an LTS (defined as the quadruple $\langle Q, q_0, Act, R\rangle$), extended with a labelling function from states to sets of atomic propositions. By means of an L^2TS, a system can be characterized by states and state changes and by the events (actions) that are performed when moving from one state to another.

In the interpretation domain of SocL, Act and AP are defined as follows.

- Act is a finite set of observable actions, ranged over by a, such as: $request(i, c)$, $response(i, c)$, $cancel(i, c)$ and $fail(i, c)$, where the name i indicates the interaction to which the operation performed by a service belongs[2], and c denotes a tuple of correlation values that identifies a particular invocation of the operation. The meaning of actions is as follows: $request(i, c)$ indicates that the performed operation corresponds to the initial request of the interaction i and its invocation is identified by the correlation tuple c; similarly, $response(i, c)$, $cancel(i, c)$ and $fail(i, c)$ characterise operations that correspond to a response, a cancellation and a failure notification, respectively, of the interaction i.

[1] Notice that this definition differs from the classical one [9] for the labelling of the transitions: we label transitions by sets of events rather than by single (un)observable events. This extension allows to model the occurrence of more than one action at the same time. Unobservable actions are rendered by the empty set.

[2] See Section 5 for an explanation of the mapping between service operations and interactions.

– *AP* is a finite set of atomic propositions, parameterized by interactions and correlation tuples, like $accepting\_request(i)$ and $accepting\_cancel(i,c)$, that can be true over a state of an L^2TS.

To define the auxiliary logic of observable actions $\mathcal{AF}(Act\$)$, we extend *Act* to include the possibility that the correlation tuples refer variables. Let *var* be a correlation variable name, we use $\$var$ to indicate the binder of the occurrences $\%var$. For example, $request(i,\$var)$ denotes a request action for the interaction i that is uniquely identified through the correlation variable $\$var$. This way, subsequent actions, corresponding e.g. to response to that specific request, can unambiguously refer it through $\%var$. We denote the extended set by $Act\$$ and let $a\$$ to range over it. We will use $a\%$ to range over actions of $Act\$$ whose correlation tuple does not contain variables of the form $\$var$. Note that $Act \subset Act\$$.

Definition 2 (Action formulae). *Given a set of observable actions Act\$, the language $\mathcal{AF}(Act\$)$ of the action formulae on Act\$ is defined as follows:*

$$\gamma ::= a\$ \mid \chi \qquad\qquad \chi ::= tt \mid a\% \mid \tau \mid \neg\chi \mid \chi \wedge \chi$$

As usual, *ff* abbreviates $\neg tt$ and $\chi \vee \chi'$ abbreviates $\neg(\neg\chi \wedge \neg\chi')$.

The introduction of variables to express correlation requires the notion of *substitution*, that in its turn requires that of pattern-matching function.

Definition 3 (Substitutions and the pattern-matching function)

– Substitutions*, ranged over by ρ, are functions mapping correlation variables to values and are written as collections of pairs of the form var/val.*
– *The empty substitution is denoted by $\emptyset$.*
– *Application of substitution ρ to a formula ϕ, written $\phi \cdot \rho$, has the effect of replacing every occurrence $\%var$ in ϕ with val, for each $var/val \in \rho$.*
– *The partial function $\mathsf{m}(\_,\_)$ from pairs of actions to substitutions, that permits performing* pattern-matching*, is defined by the following rules:*

$$\mathsf{m}(request(i,c), request(i,c')) = \mathsf{m}(c,c') \qquad \mathsf{m}(\$var, val) = \{var/val\}$$
$$\mathsf{m}(response(i,c), response(i,c')) = \mathsf{m}(c,c') \qquad \mathsf{m}(val, val) = \emptyset$$
$$\mathsf{m}(cancel(i,c), cancel(i,c')) = \mathsf{m}(c,c') \qquad \mathsf{m}(fail(i,c), fail(i,c')) = \mathsf{m}(c,c')$$
$$\mathsf{m}((e_1 \cdot c_1), (e_2 \cdot c_2)) = \mathsf{m}(e_1,e_2) \cup \mathsf{m}(c_1,c_2)$$

where notation $e \cdot c$ stands for a tuple with first element e.

Definition 4 (Action formulae semantics). *The satisfaction relation $\models$ for action formulae is defined over sets of observable actions in Act\$ and over a substitution.*

– *$\alpha \models a\$ \triangleright \rho$ iff $\exists!\, b \in \alpha$ such that $\mathsf{m}(a\$, b) = \rho$;*
– *$\alpha \models \chi \triangleright \emptyset$ iff $\alpha \models \chi$, where the relation $\alpha \models \chi$ is defined as follows:*
  • *$\alpha \models tt$ holds always;*
  • *$\alpha \models a\%$ iff $\exists!\, b \in \alpha$ such that $\mathsf{m}(a\%, b) = \emptyset$;*

- $\alpha \models \tau$ *iff* $\alpha = \epsilon$;
- $\alpha \models \neg\chi$ *iff not* $\alpha \models \chi$;
- $\alpha \models \chi \wedge \chi'$ *iff* $\alpha \models \chi$ *and* $\alpha \models \chi'$.

The notation $\alpha \models \gamma \triangleright \rho$ means: the formula γ is satisfied over the set of observable actions α (only) under substitution ρ. Notably, in the above definition we require that an observable action $a\$$ or $a\%$ matches only and only one action in α. This is a consequence of the assumption that inside a single evolution step two or more actions with the same type and interaction do not occur. Thus, e.g., the transition label $\{request(i, \langle 1\rangle), request(i, \langle 2\rangle)\}$ never appears in SocL interpretation models. Notice also that actions containing correlation variable occurrences like $\%var$ (that have not yet been replaced by values) cannot be assigned a semantics; indeed, the case $\alpha \models a\%$ requires that $\mathsf{m}(a\%, b) = \emptyset$ that, according to the rules defining the pattern-matching function, means that $a\% \in Act$, i.e. $a\%$ does not contain variables.

Definition 5 (SocL syntax). *The syntax of SocL formulae is defined as follows:*

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

We comment on salient points of the grammar above. $\pi \in AP$ are atomic propositions, A and E are *path quantifiers*, and X, U and W are indexed *next*, *until* and *weak until* operators drawn on from those firstly introduced in [9]. The next operator says that in the next state of the path, reached by an action satisfying γ, the formula ϕ holds; the meaning of the until operators is that ϕ' holds at the current or at a future state (reached by an action satisfying γ or without any specific behaviour), and ϕ has to hold until that state is reached and the actions executed satisfy χ or are unobservable; finally, the weak until operators hold either if the corresponding strong until operators hold or if for all states of the path the formula ϕ holds (by executing actions satisfying χ or unobservable). A peculiarity of SocL is that the satisfaction relation of the next and until operators may define a substitution which is propagated to subformulae. Notably, in the left side of until operators we use χ instead of γ, to avoid formulae like the following $\phi\ {}_{request(i,\langle \$v\rangle)} U_\gamma\ \phi'$, where the satisfaction relation for $request(i, \langle \$v\rangle)$ could produce a different substitution for each state that comes before the one where ϕ' holds.

To define the semantics of SocL, we first formalise the notion of *path* in an L^2TS.

Definition 6 (Path). *Let* $\langle Q, q_0, Act, R, AP, L\rangle$ *be an* L^2TS *and let* $q \in Q$.

- σ *is a* path *from* q *if* $\sigma = q$ *(the* empty path *from* q*) or* σ *is a (possibly infinite) sequence* $(q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2)\cdots$ *with* $q_0 = q$ *and* $(q_{i-1}, \alpha_i, q_i) \in R$ *for all* $i > 0$.
- *The concatenation of paths* σ_1 *and* σ_2*, denoted by* $\sigma_1\sigma_2$*, is a partial operation, defined only if* σ_1 *is finite and its final state coincides with the first state of* σ_2.
- *If* $\sigma = (q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2)\cdots$ *then the* i^{th} *state in* σ*, i.e.* q_i*, is denoted by* $\sigma(i)$.
- *We write* $path(q)$ *for the set of all paths from* q.

Definition 7 (SocL semantics). *The satisfaction relation of closed SocL formulae, i.e. formulae without unbound variables, over an* L^2TS *is defined as follows:*

- $q \models$ *true holds always;*
- $q \models \pi$ *iff* $\pi \in L(q)$;

- $q \models \neg\phi$ *iff not* $q \models \phi$;
- $q \models \phi \wedge \phi'$ *iff* $q \models \phi$ *and* $q \models \phi'$;
- $q \models E\Psi$ *iff* $\exists\, \sigma \in path(q)$ *such that* $\sigma \models \Psi$;
- $q \models A\Psi$ *iff* $\forall\, \sigma \in path(q) \quad \sigma \models \Psi$;
- $\sigma \models X_\gamma \phi$ *iff* $\sigma = (q, \alpha, q')\sigma'$, $\alpha \models \gamma \triangleright \rho$, *and* $q' \models \phi \cdot \rho$;
- $\sigma \models \phi\ {}_\chi U \phi'$ *iff there exists* $j \geq 0$ *such that* $\sigma(j) \models \phi'$ *and for all* $0 \leq i < j$:
  $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i+1))\sigma''$ *implies* $\sigma(i) \models \phi$ *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$;
- $\sigma \models \phi\ {}_\chi U_\gamma \phi'$ *iff there exists* $j \geq 1$ *such that* $\sigma = \sigma'(\sigma(j-1), \alpha_j, \sigma(j))\sigma''$
  and $\alpha_j \models \gamma \triangleright \rho$ *and* $\sigma(j) \models \phi' \cdot \rho$ *and* $\sigma(j-1) \models \phi$, *and for all* $0 < i < j$:
  $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$, *and* $\alpha_i = \epsilon$ *or* $\alpha_i \models \chi$;
- $\sigma \models \phi\ {}_\chi W \phi'$ *iff either*
  there exists $j \geq 0$ *such that* $\sigma(j) \models \phi'$ *and for all* $0 \leq i < j$:
  $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i+1))\sigma''$ *implies* $\sigma(i) \models \phi$ *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$
  or for all $0 \leq i$:
  $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i+1))\sigma''$ *implies* $\sigma(i) \models \phi$, *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$;
- $\sigma \models \phi\ {}_\chi W_\gamma \phi'$ *iff either*
  there exists $j \geq 1$ *such that* $\sigma = \sigma'(\sigma(j-1), \alpha_j, \sigma(j))\sigma''$ *and*
  $\alpha_j \models \gamma \triangleright \rho$ *and* $\sigma(j) \models \phi' \cdot \rho$ *and* $\sigma(j-1) \models \phi$, *and for all* $0 < i < j$:
  $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$, *and* $\alpha_i = \epsilon$ *or* $\alpha_i \models \chi$
  or for all $0 \leq i$:
  $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$, *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$.

Other useful operators can be derived as usual. In particular, the ones that we use in the sequel are: *false* stands for $\neg\, true$; $< \gamma > \phi$ stands for $EX_\gamma\, \phi$; $[\gamma]\, \phi$ stands for $\neg < \gamma > \neg\, \phi$; $EF\phi$ stands for $E(true\ {}_{tt} U \phi)$; $EF_\gamma\, true$ stands for $E(true\ {}_{tt} U_\gamma true)$; $AF_\gamma\, true$ stands for $A(true\ {}_{tt} U_\gamma true)$; $AG\ \phi$ stands for $\neg\, EF\, \neg\, \phi$.

We end this section by showing how the abstract properties presented in the Introduction can be expressed as generic patterns in SocL. For the sake of readability, here we consider correlation tuples composed of only one element and use notations $\$v$ and $\%v$ instead of the more cumbersome notations $\langle \$v \rangle$ and $\langle \%v \rangle$, respectively.

1. *Available* service: $AG(accepting\_request(i))$.
   This formula means that in every state the service may accept a request; a weaker interpretation of service availability, meaning that the server accepts a request infinitely often, is given by the formula $AGAF(accepting\_request(i))$.
2. *Reliable* service: $AG[request(i, \$v)]AF_{response(i,\%v)}\, true$.
   Notably, the response belongs to the same interaction i of the accepted request and they are correlated by the variable v.
3. *Responsive* service: $AG[request(i, \$v)]\, AF_{response(i,\%v) \vee fail(i,\%v)}\, true$.
4. *Broken* service: $\neg AG[request(i, \$v)]\, AF_{response(i,\%v) \vee fail(i,\%v)}\, true$.
   This formula means that the service is *temporarily broken*; instead, the formula $AG[request(i, \$v)]\, \neg EF_{response(i,\%v) \vee fail(i,\%v)}\, true$ means that the service is *permanently broken*.
5. *Unavailable* service: $AG[request(i, \$v)]\, AF_{fail(i,\%v)}\, true$.
6. *Fair* service:
   $AG[request(i, \$v)]\, A(accepting\_cancel(i, \%v)\ {}_{tt} W_{response(i,\%v) \vee fail(i,\%v)} true)$.

Table 1. COWS syntax

$s ::=$	$\mathbf{kill}(k)$	$\mid$	$u \bullet u'!\bar{e}$	$\mid$	$\sum_{i=0}^{l} p_i \bullet o_i?\bar{w}_i.s_i$		(kill, invoke, receive-guarded sum)
	$\mid s \mid s$	$\mid$	$\{\!\mid s \mid\!\}$	$\mid$	$[d]\, s$	$\mid \ * s$	(parallel, protection, delimitation, replication)

This formula means that the server is ready to accept a cancellation required by the client (fairness towards the client); instead the formula $AG[response(i, \$v)]\, \neg EF < cancel(i, \%v) > true$ means that the server cannot accept a cancellation after responding to a request (fairness towards the server).

7. *Non-ambiguous* service:
   $AG[request(i, \$v)]\, \neg EF < response(i, \%v) > \mathrm{EF} < response(i, \%v) > true$.
8. *Sequential* service:
   $AG[request(i, \$v)]\, A(\neg\, accepting\_request(i)\, {}_{tt}U_{response(i,\%v) \vee fail(i,\%v)} true)$.
9. *Asynchronous* service:
   $AG[request(i, \$v)]\, EF < response(i, \%v) \vee fail(i, \%v) > true$.
10. *Non-persistent* service: $AG[request(i, \$v)]\, AG\, \neg\, accepting\_request(i)$.

The SocL formulation of the properties 1–10 shows that their natural language description can sometimes be interpreted in different ways: hence, formalization within the logic enforces a choice among different interpretations.

3 COWS: Calculus for Orchestration of Web Services

In this section, we report the syntax of COWS and explain the semantics of its primitives in a step-by-step fashion while modelling a bank service scenario, that will be used in the rest of the paper for illustration purposes. Due to lack of space, here we only provide an informal account of the semantics of COWS and refer the interested reader to [14,15] for a formal presentation, for examples illustrating its peculiarities and expressiveness, and for comparisons with other process-based and orchestration formalisms.

The syntax of COWS is presented in Table 1. It is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, o, p, ..., mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, ranged over by e, whose exact syntax is deliberately omitted. We just assume that expressions contain, at least, values and variables, but do not include killer labels (that, hence, are *not* communicable values).

We use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables. Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$ and $x_i \neq x_j$ for each $i \neq j$). In the sequel, we shall use $+$ to abbreviate binary choice and write $[d_1, \ldots, d_n]\, s$ in place of $[d_1] \ldots [d_n]\, s$. We will write $Z \triangleq W$ to assign a symbolic name Z to the term W.

The COWS specification of the bank service is composed of two persistent subservices: *BankInterface*, that is publicly invocable by customers, and *CreditRating*, that is an 'internal' service that can only interact with *BankInterface*. The scenario also involves the processes $Client_1$ and $Client_2$ that model requests for charging the customer's credit card with some amount. Thus, the COWS term representing the scenario is

$$[o_{check}, o_{checkOK}, o_{checkFail}]\,(\,*\,BankInterface \mid *\,CreditRating\,) \mid Client_1 \mid Client_2$$

The main operator is the *parallel composition* $\_\mid\_$ that allows the different components to be concurrently executed and to interact with each other. The *delimitation* operator $[\_]\_$ is used here to declare that o_{check}, $o_{checkOK}$ and $o_{checkFail}$ are (operation) names known to the bank services, and only to them. Moreover, the *replication* operator $*\_$, that spawns in parallel as many copies of its argument term as necessary, is exploited to model the fact that *BankInterface* and *CreditRating* can create multiple instances to serve several requests simultaneously. Now, *BankInterface* and *CreditRating* are defined as follows:

$$\begin{aligned}
BankInterface \triangleq\ & [x_{cust}, x_{cc}, x_{amount}, x_{id}] \\
& p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle. \\
& (\, p_{bank} \cdot o_{check}!\langle x_{id}, x_{cc}, x_{amount}\rangle \\
& \quad \mid p_{bank} \cdot o_{checkOK}?\langle x_{id}\rangle.\ x_{cust} \cdot o_{chargeOK}!\langle x_{id}\rangle \\
& \quad + \ p_{bank} \cdot o_{checkFail}?\langle x_{id}\rangle.\ x_{cust} \cdot o_{chargeFail}!\langle x_{id}\rangle\,) \\
CreditRating \triangleq\ & [x_{id}, x_{cc}, x_a] \\
& p_{bank} \cdot o_{check}?\langle x_{id}, x_{cc}, x_a\rangle. \\
& [p, o]\,(\, p \cdot o!\langle\rangle \mid p \cdot o?\langle\rangle.\ p_{bank} \cdot o_{checkOK}!\langle x_{id}\rangle \\
& \qquad\qquad + \ p \cdot o?\langle\rangle.\ p_{bank} \cdot o_{checkFail}!\langle x_{id}\rangle\,)
\end{aligned}$$

We only comment on *BankInterface*; *CreditRating* is similar and its description is omitted. The *receive-guarded prefix* operator $p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle.\_$ expresses that each interaction with the bank starts with a *receive* activity of the form $p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle$ corresponding to reception of a request emitted by $Client_1$ or $Client_2$. Receives, together with *invokes*, written as $p \cdot o!\langle e_1, \ldots, e_m\rangle$, are the basic communication activities provided by COWS. Besides input parameters and sent values, they indicate an *endpoint*, i.e. a pair composed of a partner name p and an operation name o, through which communication should occur. $p \cdot o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint do match, and causes substitution of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). For example, variables x_{cust}, x_{cc}, x_{amount} and x_{id}, declared local to *BankInterface* by means of the delimitation operator, are initialized by the receive leading the charge activity with data provided by either $Client_1$ or $Client_2$.

Once prompted by a request, *BankInterface* creates one specific instance to serve that request and is immediately ready to concurrently serve other requests. Notably, each instance uses the *choice* operator $\_ + \_$ and exploits communication with *CreditRating* on 'internal' operations o_{check}, $o_{checkOK}$ and $o_{checkFail}$ to model a conditional choice

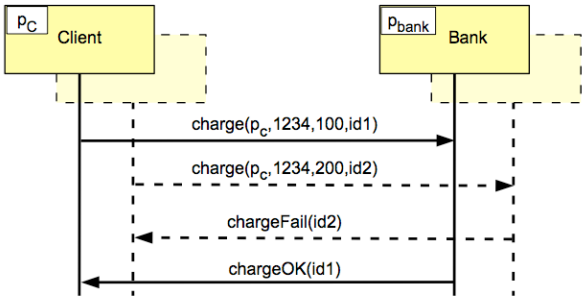

Fig. 1. Graphical representation of the bank scenario

(for the sake of simplicity, the choice between approving or not a request for charging the credit card is here completely non-deterministic). Thus, if after some invocations the service receives a message along the endpoints $p_{bank} \cdot o_{checkOk}$ or $p_{bank} \cdot o_{checkFail}$, a certain number of service instances could be able to accept it. However, the message is routed to the proper instance by exploiting the customer data stored in the variable x_{id} as a correlation value.

To illustrate, define the customer processes as follows:

$$Client_1 \triangleq p_{bank} \cdot o_{charge}!\langle p_C, 1234, 100, id_1\rangle \mid p_C \cdot o_{chargeOK}?\langle id_1\rangle + p_C \cdot o_{chargeFail}?\langle id_1\rangle$$

$$Client_2 \triangleq p_{bank} \cdot o_{charge}!\langle p_C, 1234, 200, id_2\rangle \mid p_C \cdot o_{chargeOK}?\langle id_2\rangle + p_C \cdot o_{chargeFail}?\langle id_2\rangle$$

The processes perform two requests in parallel for charging the credit card 1234 with the amounts 100 and 200. Two different correlation values, id_1 and id_2, are used to correlate the response messages to the corresponding requests. A customized UML sequence diagram depicting a possible run is shown in Figure 1.

The specification of the scenario does not exploit all COWS operators. In particular, the remaining two operators are especially useful when modelling fault handling and compensation behaviours, that, for the sake of simplicity, are not considered in this paper. In fact, *kill* activities of the form **kill**(k), where k is a killer label, can be used to force termination of all unprotected parallel terms inside the enclosing $[k]$, that stops the killing effect. Kill activities run *eagerly* with respect to the other parallel activities but critical code, such as e.g. fault/compensation handlers, can be protected from the effect of a forced termination by using the *protection* operator $\{\_\}$.

4 L^{2}TS Semantics for COWS Terms

The semantics of COWS associates an LTS to a COWS term. We have seen instead that SocL is interpreted over L^{2}TSs. We need therefore to transform the LTS associated to a COWS term into an L^{2}TS by defining a proper labelling for the states of the LTS. This is done by labelling each state with the set of actions that each active subterm of the COWS term would be able to perform immediately. Of course, the transformation preserves the structure of the original COWS LTS. For example, the concrete L^{2}TS obtained by applying this transformation to the bank scenario is shown in Figure 2.

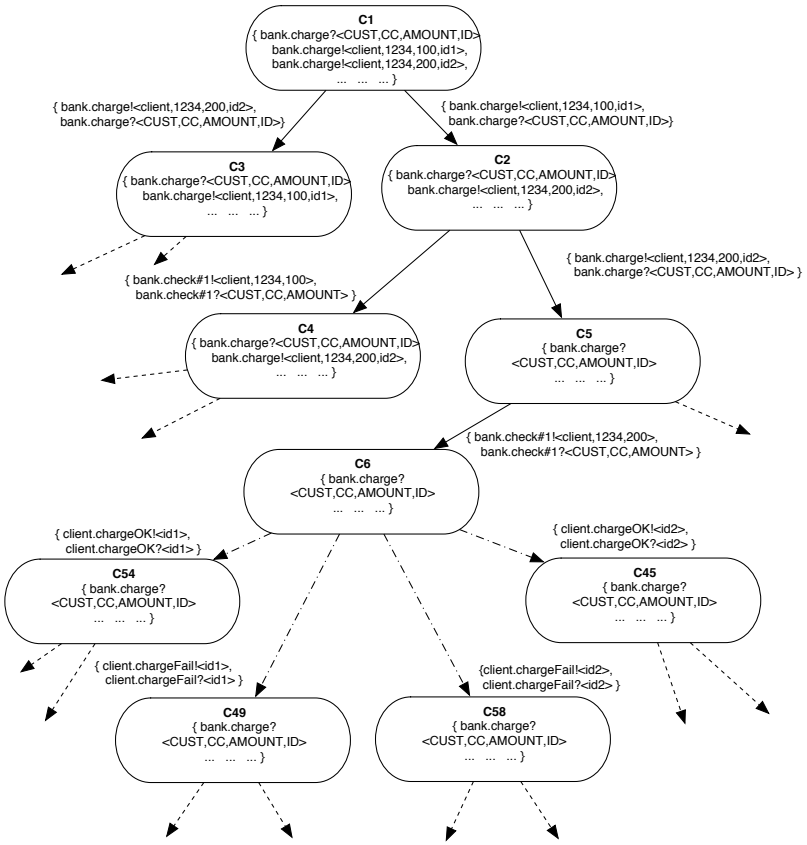

Fig. 2. Excerpt of the L^2TS for the bank scenario with concrete labels

Both in the original LTS and in the L^2TS obtained as explained before, transitions are labelled by 'concrete' actions, i.e. those actions occurring in the COWS term. Notice also that labels corresponding to communications retain all information contained in the two synchronising invoke and receive activities. However, since we are interested in verifying abstract properties of services, such as those shown in Section 2, we need to abstract away from unnecessary details by transforming concrete actions in 'abstract' ones. This is done by applying a set of suitable abstraction rules to the concrete actions. Specifically, these rules replace concrete labels on the transitions with actions belonging to the set Act, i.e. $request(i,c)$, $response(i,c)$, $cancel(i,c)$ and $fail(i,c)$, that better represent their semantics meaning. This way, different concrete actions can be mapped into the same SocL action. Moreover, the rules replace the concrete labels on the states with predicates belonging to the set AP, e.g. $accepting\_request(i)$ and $accepting\_cancel(i,c)$, that say if the service is able to accept a specific request or a cancellation of a previous request. The transformation only involves the concrete actions we want to observe. Indeed, concrete actions that are not replaced by their abstract counterparts cannot be observed.

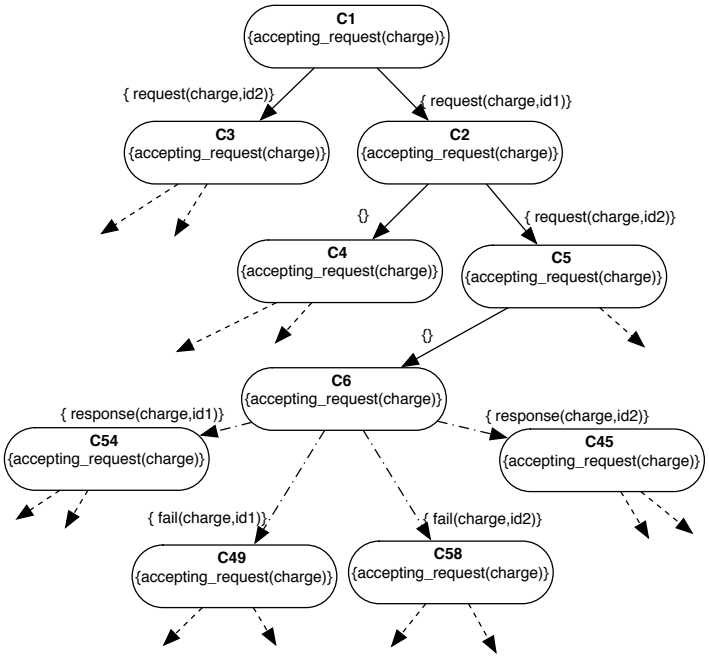

Fig. 3. Excerpt of the COWS specification of the banking example (abstract model)

For example, the abstract L^2TS of the bank scenario shown in Figure 3 is obtained by applying to the concrete L^2TS of Figure 2 the following abstraction rules:

$$\begin{aligned}
&\mathit{Action}: \mathit{charge}\langle *, *, *, \$1\rangle \rightarrow \mathit{request}(\mathit{charge}, \langle \$1\rangle)\\
&\mathit{Action}: \mathit{chargeOK}\langle \$1\rangle \rightarrow \mathit{response}(\mathit{charge}, \langle \$1\rangle)\\
&\mathit{Action}: \mathit{chargeFail}\langle \$1\rangle \rightarrow \mathit{fail}(\mathit{charge}, \langle \$1\rangle)\\
&\mathit{State}: \mathit{charge} \rightarrow \mathit{accepting_request}(\mathit{charge})
\end{aligned}$$

The first rule prescribes that whenever a concrete action $bank.charge!\langle v_1, v_2, v_3, v_4\rangle$ matching $charge\langle *, *, *, \$1\rangle$ and producing the substitution $\{\$1/v_4\}$ occurs in the label of a transition, then it is replaced by the abstract SocL action $request(charge, \langle v_4\rangle)$. Variables "$\$n$" (with n natural number) can be used to defined generic (templates of) abstraction rules. Also the wildcard " $*$ " can be used for increasing flexibility. The last rule applies to concrete labels of states instead of transitions and acts similarly. Notably, (internal) communications between the bank subservices are not transformed and, thus, become unobservable.

Of course, the set of "*Action* :" and "*State* :" rules is not defined once and for all, but is application-dependent and, thus, must be defined from time to time. Indeed, it embeds information, like the intended semantics of each action and the predicates on the states, that are not coded into the COWS specification.

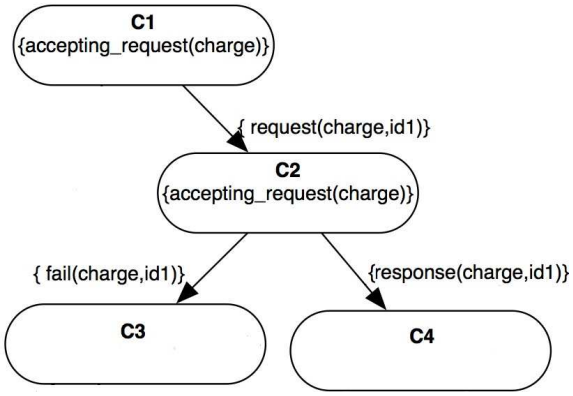

Fig. 4. An example of L^2TS

5 Model Checking COWS Specifications

To assist the verification process of SocL formulae over L^2TS, we are developing CMC, an efficient model checker for SocL that can be used to verify properties of services specified in COWS. A prototypical version of CMC can be experimented via a web interface available at the address `http://fmt.isti.cnr.it/cmc/`.

CMC is implemented by exploiting an on-the-fly algorithm which permits to achieve (in the most cases) the 'linear' complexity typical of on-the-fly model checking algorithms. Indeed, depending on the formula to be checked, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result [4,11,17]. Moreover, in case of parametric formulae, only a subset of their possible instantiations will be generated as requested by the on-the-fly evaluation.

The basic idea behind CMC is that, given a state of an L^2TS, the validity of a SocL formula on that state can be established by checking the satisfiability of the state predicates, by analyzing the transitions allowed in that state, and by establishing the validity of some subformula in some of the next reachable states. This schema has been extended with appropriate data-collection activities in order to be able to produce, in the end, also a clear and detailed explanation of the returned results (i.e. a *counterexample*), and with appropriate formula instantiation activities in order to deal with parametric formulae.

To show the peculiarity of our framework with respect to parametric formulae evaluation, we illustrate the process of establishing the satisfiability of the SocL formula

$$\phi = EX_{request(charge,\langle \$id \rangle)}\, AX_{response(charge,\langle \%id \rangle)}\, true$$

on the abstract L^2TS of Figure 4. We have therefore to check if the following holds:

$$C1 \models EX_{request(charge,\langle \$id \rangle)}\, AX_{response(charge,\langle \%id \rangle)}\, true$$

Table 2. Verification results

Property	Result	States
Available	TRUE	274
Reliable	FALSE	37
Responsive	TRUE	274
Permanently Broken	FALSE	12
Temporarily Broken	FALSE	274
Unavailable	FALSE	18

Property	Result	States
Fair 1	FALSE	3
Fair 2	TRUE	274
Non-ambiguous	TRUE	274
Sequential	FALSE	3
Asynchronous	TRUE	274
Non-persistent	FALSE	3

Thus, the model checking algorithm tries to find a next state reachable with an action matching $request(charge, \langle \$id \rangle)$. Since $C1 \xrightarrow{request(charge,\langle id1 \rangle)} C2$, then, for the semantics of action formulae, we have:

$$request(charge, \langle id1 \rangle) \models request(charge, \langle \$id \rangle) \triangleright \rho$$

where the produced substitution ρ is

$$\rho = \mathsf{m}\,(request(charge, \langle \$id \rangle), request(charge, \langle id1 \rangle)) = \mathsf{m}\,(\$id, id1) = \{id/id1\}$$

It remains then to check if $C2 \models AX_{response(charge,\langle \%id \rangle)}\, true \cdot \rho$ that is, by applying the substitution, if

$$C2 \models AX_{response(charge,\langle id1 \rangle)}\, true$$

Since $C2 \xrightarrow{response(charge,\langle id1 \rangle)} C4$, by a trivial matching between the action formula and the action on the transition, we get that the subformula $X_{response(charge,\langle \%id \rangle)}\, true \cdot \rho$ is satisfied on this path. But if we take the other path, i.e. $C2 \xrightarrow{fail(charge,\langle id1 \rangle)} C3$, we fail to find a matching, hence the same subformula is not satisfied on this path. Therefore, since the subformula is under a universal quantification, we conclude that ϕ is not satisfied.

Coming back to the abstract properties introduced in Section 1 – and formalized in SocL in Section 2 – the results of the verification on the bank service scenario are summarized in Table 2, where we also report the number of states considered during the evaluation. The instantiation of the generic patterns of formulae of Section 2 over the bank service has been obtained by just replacing any occurrence of i with *charge*. Thus, e.g., the formula predicating responsiveness of the bank service becomes:

$$AG\,[request(charge, \$v)]\, AF_{response(charge,\%v) \vee fail(charge,\%v)}\, true$$

The results show that the bank service exhibits the desired characteristics to be responsive, not broken, available, non ambiguous, and to admit parallel and iterated requests. Reliability is a too strong request for our service which can explicitly fail: indeed, responsiveness is sufficient to guarantee the expected behavior. Fairness properties are not significant for this service, that does not offer the possibility to cancel a request. Finally, the service is persistent, and we can understand why just looking at the counterexample generated when verifying the corresponding property:

```
-----------------------------------------
The formula:  AG  [ request ] AG  not (accepting_request(charge))
   is FOUND_FALSE in State C1
because
 the formula:  [ request ] AG  not (accepting_request(charge))
   is FOUND_FALSE in State C1
because
  C1  -->  C2 { bank.charge!,bank.charge? } {{ request(charge,id1)}}
 and the formula:  AG  not accepting_request(charge)
   is FOUND_FALSE in State C2
because
 the formula:  not accepting_request(charge)
   is FOUND_FALSE in State C2
because
 the formula: ASSERT(accepting_request(charge))
   is FOUND_TRUE in State C2
-----------------------------------------
```

6 Concluding Remarks

We have introduced a logical verification framework for checking functional properties of service-oriented applications specified using COWS. Our approach consists in: first, singling out a set of abstract properties describing desirable peculiar features of services; then, expressing such properties as SocL formulae; finally, verifying satisfaction of these properties by a COWS service specification by exploiting the model checker CMC. We refer the interested reader to the full version [10] of this paper for additional details on our logical verification framework and for further case studies.

One advantage of our approach is that, since the logic interpretation model (i.e. L^2TSs) is independent from the service specification language (i.e. COWS), it can be easily tailored to be used in conjunction with other SOC specification languages. To this aim, one has to define first an LTS-based operational semantics for the language of interest and then a suitable set of abstraction rules mapping the concrete actions of the language into the abstract actions of SocL. Another advantage is that SocL permits expressing properties about any kind of interaction pattern, such as *one–way*, *request–response*, *one request–multiple responses*, *one request-one of two possible responses*, etc. Indeed, properties of complex interaction patterns can be expressed by correlating SocL observable actions using interaction names and correlation values.

With respect to pure action-based or pure state-based temporal logics, action/state-based temporal logics facilitate the task of formalizing properties of concurrent systems, where it is often necessary to specify both state information and evolution in time by actions. Moreover, the use of L^2TSs as model of the logic helps to reduce the state space and, hence, the memory used and the time spent for verification. In [3], we have introduced the action/state-based branching time temporal logic UCTL that was originally tailored to express properties over UML statecharts. UCTL has been already used in [1] to describe some properties of services specified in SRML [12]. The main difference of SocL with respect to UCTL is that the former permits specifying parametric formulae, allowing correlation between service requests and responses to be expressed.

We leave for future work the extension of our framework to support a more compositional verification methodology. In fact, we are currently only able to analyse systems of services 'as a whole', i.e. we cannot analyse isolated services (e.g. a provider service without a proper client). This is somewhat related to the original semantics of COWS that follows a 'reduction' style; we are now defining an alternative operational semantics that should permit to overcome this problem.

Acknowledgements. We thank the anonymous referees for their useful comments.

References

1. Abreu, J., Bocchi, L., Fiadeiro, J., Lopes, A.: Specifying and composing interaction protocols for service-oriented system modelling. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 358–373. Springer, Heidelberg (2007)
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer, Heidelberg (2004)
3. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In: FMICS 2007. LNCS, Springer, Heidelberg (to appear, 2007)
4. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for ctl*. In: LICS, pp. 388–397. IEEE Computer Society Press, Los Alamitos (1995)
5. Bocchi, L., Fantechi, A., Gönczy, L., Koch, N.: Prototype language for service modelling: Soa ontology in structured natural language. In: Sensoria deliverable D1.1a (2006)
6. Chaki, S., Clarke, E.M., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., Veith, H.: State/event software verification for branching-time specifications. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 53–69. Springer, Heidelberg (2005)
7. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
8. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Form. Asp. Comp. 17(4), 461–483 (2005)
9. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (1995)
10. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze (2007), `http://rap.dsi.unifi.it/cows`
11. Fernandez, J., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 348–359. Springer, Heidelberg (1996)
12. Fiadeiro, J., Lopes, A., Bocchi, L.: A formal approach to service component architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Heidelberg (2006)
13. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001 and ETAPS 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)

14. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
15. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze (2007), http://rap.dsi.unifi.it/cows
16. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
17. Stirling, C., Walker, D.: Local model checking in the modal μ-calculus. In: TAPSOFT 1989. LNCS, vol. 354, pp. 369–383. Springer, Heidelberg (1989)