A Logical Verification Methodology for Service-Oriented Computing

ALESSANDRO FANTECHI, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
STEFANIA GNESI, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", ISTI - CNR, Pisa
ALESSANDRO LAPADULA, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
FRANCO MAZZANTI, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", ISTI - CNR, Pisa
ROSARIO PUGLIESE, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
FRANCESCO TIEZZI, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

We introduce a logical verification methodology for checking behavioural properties of service-oriented computing systems. Service properties are described by means of SocL, a branching-time temporal logic that we have specifically designed to express in an effective way distinctive aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses. Our approach allows service properties to be expressed in such a way that they can be independent of service domains and specifications. We show an instantiation of our general methodology that uses the formal language COWS to conveniently specify services and the expressly developed software tool CMC to assist the user in the task of verifying SocL formulae over service specifications. We demonstrate feasibility and effectiveness of our methodology by means of the specification and the analysis of a case study in the automotive domain.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; *Model checking*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; *Semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal logic*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Service-oriented computing, Web services, Formal methods, Process calculi, Model checking, Temporal logic

ACM Reference Format:
Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., and Tiezzi, F. 2010. A Logical Verification Methodology for Service-Oriented Computing. ACM Trans. Softw. Eng. Methodol. V, N, Article A (January 2010), 44 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This work is a revised and extended version of [Fantechi et al. 2008], presented at FASE 2008 in Budapest (Hungary), and has been partially funded by the EU project Sensoria (IST-2005-016004).
Author's addresses: A. Fantechi, A. Lapadula, R. Pugliese, and F. Tiezzi, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 65 Viale Morgagni, I-50134 Firenze, Italy (emails: {fantechi,lapadula,tiezzi}@dsi.unifi.it, rosario.pugliese@unifi.it); S. Gnesi and F. Mazzanti, Istituto di Scienza e Tecnologie dell'Informazione (ISTI) "A. Faedo", CNR, 1 Via Moruzzi, I-56124 Pisa, Italy (emails: {stefania.gnesi,franco.mazzanti}@isti.cnr.it)

Contents

1. INTRODUCTION

Service-oriented computing (SOC) is emerging as an evolutionary paradigm for distributed and e-business computing. This new paradigm, which finds its origin in object-oriented and component-based software development, aims at enabling developers to build networks of interoperable and collaborative applications, regardless of the platform where the applications run and of the programming language used to develop them, through the use of independent computational units, called *services*. Services are loosely coupled reusable components, that are built with little or no knowledge about clients and other services involved in their operating environment. In the end, SOC systems deliver application functionalities as services to either end-user applications or other services.

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

A major benefit of using process calculi is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of SOC. In fact, it has been already argued that type systems, observational equivalences, and modal and temporal logics provide adequate tools to address topics relevant to SOC (see, e.g., Meredith and Bjorg [2003], van Breugel and Koshkina [2006]). In particular, modal and temporal logics have long been used to represent properties of concurrent and distributed systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc. (see, e.g., Clarke et al. [1999], Bradfield and Stirling [2001], Grumberg and Veith [2008]). These logics have proved to be suitable to reason about the design of complex computing systems because they provide abstract specifications of these systems. Indeed, logics permit expressing systems properties, while process calculi permit describing system behaviours. Moreover, the application of temporal logics to the analysis of systems is often supported by efficient software tools.

By following this line of research, in this paper we introduce a logical verification methodology for checking behavioural properties of services. We do not put any specific demand on what a service is, rather, for the sake of generality, we take an abstract point of view. We think of services as computing entities which may have an internal state and can interact with each other and with user applications by, e.g., sending/accepting requests, delivering corresponding responses and, on-demand, cancelling requests. By reasoning at this abstraction level, we can single out some significant classes of service properties that can be expressed in such a general way which is independent of service domains and specifications. Thus, we say that a service is:

(1) *available*: if it is always capable to accept a request;
(2) *parallel*: if, after accepting a request, before giving a response it can accept further requests;
(3) *sequential*: if, after accepting a request, it cannot accept further requests before giving a response;
(4) *one-shot*: if, after accepting a request, it cannot accept any further requests;
(5) *off-line*: if it provides an unsuccessful response to each received request;
(6) *cancelable*: if, before a response has been provided, it permits to cancel the corresponding request;

(7) *revocable*: if, after a successful response has been provided, it permits to cancel a previous request;
(8) *responsive*: if it guarantees at least a response to each received request;
(9) *single-response*: if, after accepting a request, it provides no more than one response;
(10) *multiple-response*: if, after accepting a request, it provides more than one response;
(11) *no-response*: if it never provides a response to any accepted request;
(12) *reliable*: if it guarantees a successful response to each received request.

Albeit not exhaustive, the above list contains many desirable properties (see, e.g., van der Aalst et al. [2003], Alonso et al. [2004], Bocchi et al. [2006]) of the externally observable behaviour of services.

The previous properties are stated in terms of the visible *actions* that services may perform. Any of these actions has a *type*, e.g. accept a request, provide a response, etc., and is part of a possibly long-running *interaction* (elsewhere sometimes also called *conversation*) started when a client firstly invokes one of the operations exposed by the service. Thus, according to this abstract point of view, an interaction identifies a collection of actions, each of them corresponding to a single invocation of a service operation. At first sight, then, the service properties could be formulated by properly tailoring an action-based temporal logic among those already proposed in the literature of concurrency theory (see, e.g., Hennessy and Milner [1985], De Nicola and Vaandrager [1990], Stirling [2001]). However, these logics are not expressive enough to, e.g., associate a response action to the request acceptance action that originated the interaction. The possible presence of more request actions sharing the same type and interaction name may prevent this association to occur. Indeed, multiple instances of an interaction can be simultaneously active since service operations can be independently invoked by several clients. Hence, by taking inspiration from SOC emerging standards like WS-BPEL [OASIS WSBPEL TC 2007] and WS-CDL [Kavantzas et al. 2004], to enable the previously mentioned association we use *correlation data* as a third attribute of actions that services can do.

The first contribution of this paper is the definition of the branching-time temporal logic SocL that, by relying on the actions described above, is capable of expressing in an effective way distinctive aspects of services and of formalizing the 'abstract' properties previously stated. SocL falls within a large body of research (see, e.g., Lawford et al. [1996], Chaki et al. [2004; 2005], Baier et al. [2004], Pecheur and Raimondi [2006], ter Beek et al. [2008]) that aims at supporting the analysis of action- and state-based properties of systems. Indeed, SocL formulae predicate properties of systems in terms of states and state changes, of the actions that are enabled in a given state, and of the actions that are performed when moving from one state to another. Thus, the interpretation domain of SocL formulae are *Doubly Labelled Transition Systems* (L^2TSs [De Nicola and Vaandrager 1995]), namely extensions of *Labelled Transition Systems* (LTSs) with a labelling function from states to sets of atomic propositions.

Our second contribution is the introduction of a novel verification methodology of service properties. The properties are initially formalized as SocL formulae while preserving their independence from individual service domains and specifications. Afterwards, these formulae can be tailored to a given specification of a service by means of some *abstraction rules* that relate actions in the specification with actions of the logic. Then, once a way to get a representation of the service specification in terms of an L^2TS has been provided, the verification process can take place. While, in principle, service behaviour could be directly specified using L^2TSs, practical considerations suggest that it is more convenient to resort to some linguistic formalism. In fact, when used as a specification formalism, L^2TSs are too low level and, above all, suffer for lack of compositionality in the sense that they offer no means for constructing the L^2TS of a composed service in terms of the L^2TSs of its components. On the contrary, linguistic terms are more intuitive and concise notations. Using them, services are built in a compositional way by using the operators provided by the language and, furthermore, they are syntactically finite, even when the corresponding semantic model, perhaps defined in terms of L^2TSs, is not. Therefore, in this paper we show an instantiation of our methodology that uses COWS (*Calculus for Orchestration of Web Services* [Lapadula et al. 2007]) as the language to specify and combine services, while modelling their dynamic behaviour.

The third contribution of our work is the software tool CMC, namely a *verification environment* for SocL formulae over COWS specifications of services. Input to CMC are a COWS term, the SocL formula to be checked and a set of abstraction rules that are exploited during the verification process for generating the L^{2}TS model of the COWS term. The model is generated on-the-fly, hence, depending on the formula to be checked, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result. Moreover, when the formula is not satisfied, CMC provides diagnostic information in the form of a computation that makes the verification fail.

We demonstrate feasibility and effectiveness of our methodology by means of the specification and the analysis of a case study, namely an automotive scenario studied within the EU project Sensoria [SENSORIA 2005].

The rest of the paper is organized as follows. Section 2 introduces syntax and semantics of SocL, while Section 3 presents the model checker engine for SocL. Section 4 presents COWS's main features in a step-by-step fashion while modelling a bank service (which is part of the automotive scenario). Section 5 describes the verification environment CMC whose implementation exploits the model checker engine for SocL presented in Section 3. Section 6 presents the automotive scenario, firstly through an informal UML-like description, then through a formal specification written in COWS. The COWS specification is thoroughly analysed in Section 7. Final remarks and comparisons with related work are reported in Section 8, while directions for future work are touched upon in Section 9.

2. THE LOGIC SOCL

SocL is an action- and state-based branching-time logic that makes use of high-level temporal operators drawn from mainstream logics like CTL [Clarke and Emerson 1981], ACTL [De Nicola and Vaandrager 1990] and ACTLW [Meolic et al. 2008]. In this section, we first introduce some preliminary definitions, then define syntax and semantics of SocL and finally show how the logic can be used to formalize the service properties we have mentioned in the Introduction.

2.1. Preliminary definitions

In this section we report the definitions of the semantic structures which the logic relies on. They permit to characterise a service in terms of states and predicates that are true over them, and of state changes and actions performed when moving from one state to another.

Let *Act* be a set of elements called *observable actions*. We will use η to range over 2^{Act}.

Definition 2.1 (*Labelled Transition System*). A Labelled Transition System (LTS) over the set of observable actions *Act* is a quadruple $\langle Q, q_0, Act, R\rangle$, where:

— Q is a set of states;
— $q_0 \in Q$ is the initial state;
— $R \subseteq Q \times 2^{Act} \times Q$ is the transition relation.

Now, if we extend an LTS with a labelling function from states to sets of atomic propositions we get a system with also labels over states, namely a *Doubly Labelled Transition System*.

Definition 2.2 (*Doubly Labelled Transition System*). A Doubly Labelled Transition System (L^{2}TS) over the set of observable actions *Act* and the set of atomic propositions *AP* is a tuple $\langle Q, q_0, Act, R, AP, L\rangle$, where:

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

The main difference between the above definitions and the usual ones (as given e.g. by De Nicola and Vaandrager [1995]) is that transitions are labelled by a set of actions rather than by a single action. In particular, the empty set, i.e. $\emptyset$, that labels transitions during which no observable actions occur, corresponds to the unobservable internal action in the usual definitions. In the sequel, as a matter of notation, instead of $(q, \eta, q') \in R$ we may write $q \xrightarrow{\eta} q'$.

Paths within L^{2}TSs represent service computations and are defined as follows:

Definition 2.3 (*Path*). Let (Q, q_0, Act, R, AP, L) be an L^{2}TS 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 of transitions $(q_1, \eta_1, q_2)(q_2, \eta_2, q_3) \cdots$, with $q_1 = q$ and $(q_i, \eta_i, q_{i+1}) \in R$ for all $i \geq 1$.
— A *full-path* is a path that cannot be further extended: it is infinite or ends in a state without outgoing transitions. We write $path(q)$ for the set of all full-paths from q.
— If (q_i, η_i, q_{i+1}) is the ith transition in a path σ then we will write $\sigma(i)$, $\sigma\{i\}$ and $\sigma(i+1)$ to indicate q_i, η_i and q_{i+1}, respectively.

2.2. SocL syntax and semantics

We start introducing the set of observable actions which SocL is based upon. As we said in the Introduction, the actions of the logic should correspond to the actions performed by service providers and service consumers, and are characterised by three attributes: type, interaction name, and correlation data[1]. Moreover, to enable capturing correlation data used to link together actions executed as part of the same interaction, they may also contain variables, that we call *correlation variables*. In the sequel, we will usually write *val* to denote a generic *correlation value* and *var* to denote a generic correlation variable. For a given correlation variable *var*, its binding occurrence will be denoted by $\underline{var}$; all remaining occurrences, that are called *free*, will be denoted by *var*.

Definition 2.4 (*SocL Actions*). SocL actions have the form $t(i, c)$, where t is the type of the action, i is the name of the interaction which the action is part of, and c is a tuple of correlation values and variables identifying the interaction (i and c can be omitted whenever they do not play any role). We assume that variables in the same tuple are pairwise distinct. We will say that an action is *closed* if it does not contain variables. We will use Act^v to denote the set of all actions, $\underline{\alpha}$ as a generic element of Act^v (underlying emphasises the fact that the action may contain variable binders), and α as a generic action without variable binders. We will use Act to denote the subset of Act^v that only contains closed actions (i.e. actions without variables) and η as a generic subset of Act.

Example 2.5. Action $request(charge, 1234, 1)$ could stand for an action of type *request* starting an (instance of the) interaction *charge* which will be identified through the correlation tuple $\langle 1234, 1\rangle$. A *response* action corresponding to the request above, for example, could be written as $response(charge, 1234, 1)$. Moreover, if some correlation value is unknown at design time, e.g. the identifier 1, a (binder for a) correlation variable *id* can be used instead, as in the action $request(charge, 1234, \underline{id})$. A corresponding response action could be written as $response(charge, 1234, id)$, where the (free) occurrence of the correlation variable *id* indicates the connection with the action where the variable is bound.

To define the syntax of SocL we rely on an auxiliary logic of actions.

Definition 2.6 (*Action formulae*). The language of the action formulae on Act^v is defined as follows:

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

[1]Notice that correlation data are simply regarded as literals and, for such data, the logic supports the equality test only. In fact, we do not need to deal with data types.

As usual, we will use ff to abbreviate $\neg tt$ and $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$.

The syntax above states that an action formula γ can be either an action $\underline{\alpha}$, which may contain variable binders, or an action formula χ, which is a boolean compositions of unobservable internal actions τ and observable actions α without variable binders. As we shall also clarify later, the distinction between action formulae γ and χ is motivated by two reasons: (1) some logical operators can accept as argument only action formulae without variable binders, and (2) actions containing variable binders cannot be composed.

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

Definition 2.7 (*Substitutions*). *Substitutions*, ranged over by ρ, are functions mapping correlation variables to values and are written as collections of pairs of the form var/val. The empty substitution is denoted by $\emptyset$. Application of substitution ρ to a formula ϕ, written $\phi\rho$, has the effect of replacing every free occurrence of var in ϕ with val, for each $var/val \in \rho$.

Definition 2.8 (*Matching function*). The partial function match from $Act^v \times Act$ to substitutions is defined by structural induction by means of auxiliary partial functions defined over syntactic subcategories of Act^v through the following rules:

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

where (e,c) stands for a tuple with first element e, and $\langle\rangle$ stands for the empty tuple. Notably, an action containing free variable occurrences cannot match any closed action.

Example 2.9. Let us consider again the actions introduced in Example 2.5. Then, we have $\mathsf{match}(response(charge, 1234, 1), response(charge, 1234, 1)) = \emptyset$ and also $\mathsf{match}(request(charge, 1234, \underline{id}), request(charge, 1234, 1)) = \{id/1\}$. Instead $\mathsf{match}(request(charge, 1234, \underline{id}), response(charge, 1234, 1))$ is not defined since the actions have different types.

Definition 2.10 (*Action formulae semantics*). The satisfaction relation for action formulae is defined over a set of closed actions and a substitution.

— $\eta \models \underline{\alpha} \blacktriangleright \rho$ iff $\exists\, \alpha' \in \eta$ such that $\mathsf{match}(\underline{\alpha}, \alpha') = \rho$;
— $\eta \models \chi \blacktriangleright \emptyset$ iff $\eta \models \chi$, where the relation $\eta \models \chi$ is defined as follows:
  — $\eta \models tt$ holds always;
  — $\eta \models \alpha$ iff $\alpha \in \eta$;
  — $\eta \models \tau$ iff $\eta = \emptyset$;
  — $\eta \models \neg\chi$ iff not $\eta \models \chi$;
  — $\eta \models \chi \wedge \chi'$ iff $\eta \models \chi$ and $\eta \models \chi'$.

Notation $\eta \models \gamma \blacktriangleright \rho$ means: the formula γ is satisfied over the set of closed actions η under substitution ρ. Since function match (i.e. match_e) is undefined when its first argument contains free variables, the semantics of actions containing free occurrences of correlation variables is undefined as well.

Notice also that satisfiability of a formula under a non-empty substitution may be required only in the first case of the definition above, because the remaining cases deal with formulae that do not contain variable binders. Finally, the action formula τ is satisfied over the empty set of actions.

Table I. Some sample formulae and their interpretations

$\neg\, EX_{tt}\, true$: no transition can be performed.

$EX_{request(charge,1234,\underline{id})}\, AX_{response(charge,1234,id)}\, true$: after a request for the interaction *charge* has been accepted, a correlated response must be immediately performed.

$A(true\ {}_{\neg response(check,1234)}\, U_{\,request(check,1234)}\ true)$: the request for the interaction (check,1234), which must always occur, is never proceeded by a response for the same interaction.

$A(true\ {}_{\neg response(check,1234)}\, W_{\,request(check,1234)}\ true)$: the request for the interaction (check,1234), if it ever occurs, is never proceeded by a response for the same interaction.

To define the syntax of the logic, the last ingredient we need is the set of atomic propositions. They correspond to the properties that can be true over the states of services.

Definition 2.11 (*Atomic propositions*). SocL atomic propositions have the form $p(i,c)$, where p is the name, i is an interaction name, and c is a tuple of correlation values and free variables identifying i (i and c can be omitted whenever they do not play any role). We will use AP to denote the set of all atomic propositions and π as generic element of AP.

Notably, atomic propositions cannot contain variable binders.

Example 2.12. Proposition $accepting\_request(charge)$ indicates that a state can accept requests for interaction *charge*, while proposition $accepting\_cancel(charge, 1234, 1)$ indicates that a state permits to cancel those requests for interaction *charge* identified by the correlation tuple $\langle 1234, 1\rangle$.

Definition 2.13 (*SocL syntax*). The syntax of SocL formulae is defined as follows:

$$\begin{array}{lrcl} \textit{(state formulae)} & \phi & ::= & true \;\mid\; \pi \;\mid\; \neg\phi \;\mid\; \phi\wedge\phi' \;\mid\; E\Psi \;\mid\; A\Psi \\ \textit{(path formulae)} & \Psi & ::= & X_\gamma\phi \;\mid\; \phi\,{}_\chi U_\gamma\,\phi' \;\mid\; \phi\,{}_\chi W_\gamma\,\phi' \end{array}$$

E and A are existential and universal (resp.) *path quantifiers*. X, U and W are the *next*, *(strong) until* and *weak until* operators drawn from those firstly introduced by De Nicola and Vaandrager [1990] and subsequently elaborated by Meolic et al. [2008]. Intuitively, the formula $X_\gamma\phi$ says that in the next state of the path, reached by an action satisfying γ, the formula ϕ holds. The formula $\phi\,{}_\chi U_\gamma\,\phi'$ says that ϕ' holds at some future state of the path reached by a last action satisfying γ, while ϕ holds from the current state until that state is reached and all the actions executed in the meanwhile along the path satisfy χ. The formula $\phi\,{}_\chi W_\gamma\,\phi'$ holds on a path either if the corresponding strong until operator holds or if for all the states of the path the formula ϕ holds and all the actions of the path satisfy χ. Notice that the weak until operator (also called *unless*) is not derivable from the strong until operator since disjunction or conjunction of path formulae is not expressible in the syntax of SocL, similarly to any other pure branching-time temporal logic. Some examples of SocL formulae together with their intuitive meaning are reported in Table I.

The semantics of SocL formulae is only defined for *closed* formulae, namely those formulae where any free occurrence of a correlation variable is syntactically preceded by its binding occurrence. Given the formulae $X_\gamma\phi'$, $\phi\,{}_\chi U_\gamma\,\phi'$ and $\phi\,{}_\chi W_\gamma\,\phi'$, variables occurring in γ *syntactically precede* the variables occurring in ϕ'. The interpretation domain of SocL formulae are L^{2}TSs over the set of actions *Act* and the set of atomic propositions *AP*. To define the semantics of SocL we use the notion of full-path and the notations $\sigma(i)$ and $\sigma\{i\}$ introduced in Definition 2.3.

Definition 2.14 (*SocL semantics*). Let $\langle Q, q_0, Act, R, AP, L\rangle$ be an L^{2}TS, $q \in Q$ and $\sigma \in path(q)$. The satisfaction relation of closed SocL formulae 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 $\exists \sigma \in path(q) : \sigma \models \Psi$;
— $q \models A\Psi$ iff $\forall \sigma \in path(q) : \sigma \models \Psi$;
— $\sigma \models X_\gamma \phi$ iff $\exists \rho : \sigma\{1\} \models \gamma \blacktriangleright \rho$ and $\sigma(2) \models \phi\rho$;
— $\sigma \models \phi\,_\chi U_\gamma \phi'$ iff $\exists\, j \geq 1$
$\sigma(j) \models \phi$, and $\exists \rho : \sigma\{j\} \models \gamma \blacktriangleright \rho$ and $\sigma(j+1) \models \phi'\rho$,
and $\forall\, 1 \leq i < j : \sigma(i) \models \phi$ and $\sigma\{i\} \models \chi$;
— $\sigma \models \phi\,_\chi W_\gamma \phi'$ iff either $\sigma \models \phi\,_\chi U_\gamma \phi'$ or $\forall\, i \geq 1 : \sigma(i) \models \phi$ and $\sigma\{i\} \models \chi$.

A distinctive feature of SocL is that the satisfaction relation of the next and until operators may define substitutions which are propagated to subformulae. Notably, in the left hand side of the until operators we use χ (i.e., closed actions), instead of γ, to prevent writing such formulae as $\phi_{request(i,\underline{var})} U_\gamma \phi'$ whose semantics would require $request(i, \underline{var})$ to be performed zero or more times before γ, which could produce undefined or multiple defined bindings on var. This motivates the syntactical constraints on the definition of action formulae (Definition 2.6).

Other useful logic 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 $EX_\gamma\, \phi$; this is the *diamond* operator introduced by Hennessy and Milner [1985] and, intuitively, states that it is *possible* to perform an action satisfying γ and thereby reaching a state that satisfies formula ϕ.
— $[\gamma]\,\phi$ stands for $\neg<\gamma>\neg\,\phi$; this is the *box* operator introduced by Hennessy and Milner [1985] and states that no matter how a process performs an action satisfying γ, the state it reaches in doing so will *necessarily* satisfy the formula ϕ.
— Variants of until operators, which do not specify the last action leading to the state at which the formula on the right hand side holds, can be defined as follows:
  — $E(\phi\,_\chi U\, \phi')$ stands for $\phi' \vee E(\phi_\chi U_\chi\, \phi')$;
  — $A(\phi\,_\chi U\, \phi')$ stands for $\phi' \vee A(\phi_\chi U_\chi\, \phi')$;
  — $E(\phi\,_\chi W\, \phi')$ stands for $\phi' \vee E(\phi_\chi W_\chi\, \phi')$;
  — $A(\phi\,_\chi W\, \phi')$ stands for $\phi' \vee A(\phi_\chi W_\chi\, \phi')$.
— $EF\phi$ stands for $E(true\,_{tt} U\phi)$ and means that there is some path that leads to a state at which ϕ holds; that is, ϕ *eventually* holds on some path.
— $EF_\gamma\, \phi$ stands for $E(true_{tt}\, U_\gamma\, \phi)$ and means that there is some path that, by a last action satisfying γ, leads to a state at which ϕ holds; if ϕ is *true*, we say that an action satisfying γ will *eventually* be performed on some path.
— $AF_\gamma\, \phi$ stands for $A(true_{tt}\, U_\gamma\, \phi)$ and means that an action satisfying γ will be performed in the future along every path and at the reached states ϕ holds; if ϕ is *true*, we say that an action satisfying γ will *always eventually* be performed.
— $AG\,\phi$ stands for $\neg\, EF \neg\, \phi$ and states that ϕ holds at every state on every path; that is, ϕ holds *globally*.

2.3. A few patterns of service properties

We now present how the service properties listed in the Introduction can be expressed as formulae in SocL. To do this, we characterise the set of actions Act^v and the set of atomic propositions AP which the logic is based upon as follows.

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

— AP contains (at least) the following three atomic propositions $accepting\_request(i,c)$, $accepting\_cancel(i,c)$ and $accepting\_undo(i,c)$, whose meaning is obvious.

For the sake of readability, in the formalization of the properties we consider correlation tuples composed of only one element.

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

(2) *Parallel* service:
$AG\,[request(i,\underline{var})]\,E(true_{\neg\,(responseOk(i,var)\vee responseFail(i,var))}U\ accepting\_request(i))$.
This formula means that the service can accept several requests simultaneously. Indeed, in every state, if a request is accepted then, in some future state, further requests for the same interaction can be accepted before giving a response to the first accepted request. Notably, the responses belongs to the same interaction i of the accepted request and they are correlated by the variable var.
This is a clear example of the usefulness of the combined approach based on both actions and propositions. In fact, as we shall further clarify in Section 5, the action $request(i,\underline{var})$ corresponds to the acceptance of a request sent by a client, while the proposition $accepting\_request(i)$ indicates that in the current state the service is able to accept a request from some client (but it has not received such request yet). In this way, SocL can easily deal with both performed and potential actions.

(3) *Sequential* service:
$AG\,[request(i,\underline{var})]\,A(\neg\,accepting\_request(i)\,_{tt}\,U_{responseOk(i,var)\vee responseFail(i,var)}true)$.
In this case, the service can accept at most one request at a time. Indeed, after accepting a request, it cannot accept further requests for the same interaction before replying to the accepted request.

(4) *One-shot* service:
$AG\,[request(i)]\,AG\,\neg\,accepting\_request(i)$.
This formula states that the service is not persistent because, after accepting a request, in all future states, it cannot accept any further request.

(5) *Off-line* service:
$AG\,[request(i,\underline{var})]\,AF_{responseFail(i,var)}\ true$.
This formula states that whenever the service accepts a request, it always eventually provides an unsuccessful response.

(6) *Cancelable* service:
$AG\,[request(i,\underline{var})]\,A(accepting\_cancel(i,var)\,_{tt}W_{responseOk(i,var)\vee responseFail(i,var)}true)$.
This formula means that the service is ready to accept a cancellation required by the client (fairness towards the client) before possibly providing a response to the accepted request. A different formulation is given by:
$AG\,[responseOk(i,\underline{var})]\,\neg EF <cancel(i,var)> true$
meaning that the service cannot accept a cancellation after responding to a request (fairness towards the service).

(7) *Revocable* service:
$EF_{responseOk(i,\underline{var})}\,EF(accepting\_undo(i,var))$
Again, we can have two interpretations. While the previous formula expresses a sort of weak revocability (i.e., after a successful response has been provided, the service can eventually accept an undo of the corresponding request), the following one corresponds to a stronger interpretation
$AG\,[responseOk(i,\underline{var})]\,A(accepting\_undo(i,var)\,_{tt}W_{undo(i,var)}true)$

since it guarantees that the service can always accept an undo of the request after providing the response.

(8) *Responsive* service:
$AG\ [request(i, \underline{var})]\ AF_{responseOk(i,var) \vee responseFail(i,var)}\ true.$
The formula states that whenever the service accepts a request, it always eventually provides at least a (successful or unsuccessful) response.

(9) *Single-response* service:
$AG\ [request(i, \underline{var})]\ \neg EF_{responseOk(i,var) \vee responseFail(i,var)}\ EF_{responseOk(i,var) \vee responseFail(i,var)}\ true.$
The formula means that whenever the service accepts a request, it cannot provide two or more correlated (successful or unsuccessful) responses, i.e. it can only provide at most a single response.

(10) *Multiple-response* service:
$AG\ [request(i, \underline{var})]\ AF_{responseOk(i,var) \vee responseFail(i,var)}\ AF_{responseOk(i,var) \vee responseFail(i,var)}\ true.$
Differently from the previous formula, here the service always eventually provides two or more responses.

(11) *No-response* service:
$AG\ [request(i, \underline{var})]\ \neg EF_{responseOk(i,var) \vee responseFail(i,var)}\ true.$
This formula means that the service never provides a (successful or unsuccessful) response to any accepted request.

(12) *Reliable* service:
$AG\ [request(i, \underline{var})]\ AF_{responseOk(i,var)}\ true.$
This formula guarantees that the service always eventually provides a successful response to each accepted request.

The SocL formulation of the above properties is instructive in that it witnesses that the natural language descriptions of the properties can sometimes be interpreted in different ways: therefore, formalization within the logic enforces a choice among different interpretations. Notably, the formulation is given in terms of abstract actions and state predicates thus, rather than specific properties, the properties we have considered so far represent sorts of generic patterns or classes of properties. From time to time, type/name, interaction and correlation tuple of actions and propositions have to be projected on the actual actions performed by the specific service to be analysed. They, however, can be easily instantiated, as shown in Section 7.1, and such instantiation can be in principle automated.

3. MODEL CHECKING FOR SOCL

To assist the verification process of SocL formulae over L^{2}TSs we have developed a bounded, on-the-fly model checking engine. Figure 1 illustrates the input expected by the SocL model checker and the output it produces. In this section, we illustrate the principles upon which the model checking engine is based.

The first principle regards the ‘bounded’ approach. The evaluation of a formula is achieved iteratively: each iteration is allowed to explore the model up to a certain depth until a final boolean answer can be given. The schema of the procedure is shown in Figure 2. At each iteration, the value of the user defined parameter `MaxDepth` (representing the maximal evaluation depth for the current iteration) is doubled. The procedure terminates only whenever the variable `Result` (line `06`), as modified by the last call of `Eval`, obtains a `True` or `False` value. An important point is that the restarting of an evaluation is not a completely new evaluation process, but is a process that takes advantage of the recorded results of subcomputations already performed in previous iterations to speed up the current evaluation. Of course, if the formula to be evaluated requires to potentially explore the whole state space (as it is the case for formulae of the form $AG\,\phi$, unless a state is found that does not satisfy ϕ), the procedure would never terminate if the model has infinite states.

This approach, initially introduced to overcome the problem of infinite state spaces, turns out to be quite useful also in the case of finite state systems since it allows to generate smaller explanations for

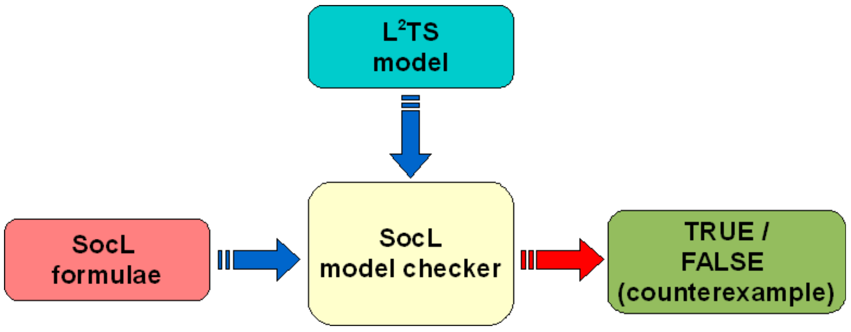


Fig. 1. Input/output behaviour of the SocL model checker.

```
function Evaluate($\phi$,L2TS) return EvaluationResult is
  Result: EvaluationResult;
  MaxDepth: Natural := default_initial_value;
begin
  Eval($\phi$,L2TS.InitialState,0,Result);
  while Result = Aborted loop
    MaxDepth := MaxDepth*2;
    Eval($\phi$,L2TS.InitialState,0,MaxDepth,Result);
  end loop;
  return Result;
end Evaluate;
```

Fig. 2. Bounded evaluation schema.

the result than the corresponding classical unbounded depth-first version. For example, if we are just looking for a deadlock state, a plain unbounded approach will return as proof the first deadlock state encountered according to a plain depth-first exploration of the model, while the bounded approach will return a deadlock state close to the initial state of the model (because all alternative paths up to the current `MaxDepth` are explored before proceeding more deeply in the exploration).

The second principle regards the 'on-the-fly' approach: the evaluation of a formula, starting from the initial state of the L^{2}TS, proceeds in a top-down way with respect to the formula structure, and in a depth-first way with respect to the model structure, in agreement with the current maximal evaluation depth limit. The relevant fact of this top-down/depth-first traversal of the formula/model is that only the truly necessary substates and subformulae are analyzed, limiting in this way the 'on demand' generations to a subset of the subformulae instantiations and to a subset of the model state space. Depending on the type of logical operator, a specific evaluation routine is invoked, as shown in Figure 3. Notably, while `EDepth` (the current evaluation depth) and `MaxDepth` are passed to all the specific evaluation subroutines, the local variable `RecDepth` (initialized with the value `0` and representing the recursion depth upon which the current `Result` depends) is passed as parameter only to the subroutines evaluating recursive operators.

Figure 4 shows the detailed schema of the evaluation algorithm for the universally quantified 'next' operator. When the evaluation of a formula of the form $AX_\gamma\phi$ begins, it is firstly checked (line `05`) if the maximum evaluation depth has been already reached, in which case an `Aborted` value is returned. The second check is performed (lines `06-09`) to see whether the computation has already been done, in which case the previously saved result is returned. After these checks, the analysis of the possible outgoing transitions from the current state begins. If no outgoing transitions exist (i.e. the state is a 'final' state) then the formula is evaluated `False`; this final computation result is *saved* and this boolean value returned (lines `11-13`). If the current state has some outgoing

```
procedure Eval(φ,CurrentState,EDepth,MaxDepth,out Result) is
  RecDepth: Natural :=0;
begin
  case $\phi$ is
    when true   ->  Result := True;
    when $\pi$   ->  EvalPredicate($\phi$,CurrentState,EDepth,MaxDepth,Result);
    when $\neg\ \phi'$ ->  EvalNegation($\phi$,CurrentState,EDepth,MaxDepth,Result);
    when $\phi 1 \wedge \phi 2$ -> EvalConjunction($\phi$,CurrentState,EDepth,MaxDepth,Result);
    when EX $\gamma\ \phi'$ -> EvalExistNext($\phi$,CurrentState,EDepth,MaxDepth,Result);
    when AX $\gamma\ \phi'$ -> EvalForallNext($\phi$,CurrentState,EDepth,MaxDepth,Result);
    when E $\phi 1\ \chi$ U $\gamma\ \phi 2$  ->
      EvalExistsUntil($\phi$,CurrentState,EDepth,MaxDepth,RecDepth,Result);
    when A $\phi 1\ \chi$ U $\gamma\ \phi 2$  ->
      EvalForallUntil($\phi$,CurrentState,EDepth,MaxDepth,RecDepth,Result);
    when E $\phi 1\ \chi$ W $\gamma\ \phi 2$  ->
      EvalExistsWUntil($\phi$,CurrentState,EDepth,MaxDepth,RecDepth,Result);
    when A $\phi 1\ \chi$ W $\gamma\ \phi 2$  ->
      EvalForallWUntil($\phi$,CurrentState,EDepth,MaxDepth,RecDepth,Result);
  end case;
end Eval;
```

Fig. 3. From generic to operator specific evaluation mapping.

transitions then all of them (line 16) must satisfy the action formula γ (line 17). To check this, all the transitions are analysed in sequence. For each outgoing transition from the current state:

— A set of substitutions Θ is calculated (line 19) by matching the actual transition label with the (possibly parametric) action formula γ; if γ does not contain variables then Θ only contains the empty substitution.
— It is then checked (lines 21-33) that the application of at least one substitution in Θ to ϕ generates a formula which holds in the current state of the transition; the application of the empty substitution returns the formula unchanged.

If a transition is found which does not satisfy γ then we can immediately conclude that the formula $AX_\gamma\phi$ is false, in which case the cause of the failure is recorded (line 45) and the result is set to `False` (line 46).

The recording of the cause of the failure is necessary for the generation of the full explanation of the final evaluation result, if requested by the user. Whenever the application of a substitution generates a subformula that holds in the target state of the transition, then the information of this partial successful subcomputation is recorded (lines 24-27) and the analysis of further substitutions for this transition is stopped (line 28). We cannot however yet return `True` until all the transitions have been analyzed. If, on the contrary, the application of all substitutions generates subformulae that do not hold in the target state of the transition (line 34) then all the already recorded information about successful subcomputations are replaced with the information relative to the failed transition and the result of evaluation of the original formula is set to `False` (lines 35-38). There is a last case to be considered: if

— the application of at least one of the substitutions generates a subformula whose evaluation with the current `EDepth` limit returns `Aborted`, and
— the application of all the other substitutions generates subformulae whose evaluation returns `Aborted` or `False`

then it will be no longer possible to return `True` as evaluation result, but we go on analysing the remaining transitions in case a definitive negative answer can still be provided (lines 40-43).

```
procedure EvalForallNext
          ($AX\gamma\ \phi$, CurrentState, EDepth, MaxDepth, out Result) is
  TmpResult, LastResult: EvaluationResult;
begin
  if EDepth = MaxDepth then Result := Aborted; return; end if;
  if <$AX\gamma\ \phi$, CurrentState> → SavedResult exists then
     Result := SavedResult;
     return;
  end if;
  if OutgoingTransitions(CurrentState) is empty then
    save <$AX\gamma\ \phi$, CurrentState> → False;
    Result := False;
    return;
  end if;
  Result := True;
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, $\gamma$) then
      TargetState := Transition.TargetState;
      $\Theta$ := TransitionSubstitutions(Transition.Label, $\gamma$);
      TmpResult := False;
      foreach substitution $\rho$ in $\Theta$ do
       $\phi$' := ApplySubstitution($\phi$ , $\rho$);
       Eval($\phi$', TargetState, EDepth+1, LastResult);
       if LastResult = True then
         add_explanation <$AX\gamma\ \phi$, CurrentState> →
                        (Transition.Label,<$\phi$', TargetState>);
         TmpResult := True;
         exit;  -- do not continue the analysis of the other substitutions
       elsif LastResult = Aborted then
         -- continue the analysis of the other substitutions
         TmpResult := Aborted;
        end if;
      end loop;
      if TmpResult = False then
        set_explanation <$AX\gamma\ \phi$, CurrentState> →
                     (Transition.Label,<$\phi$', TargetState>);
        Result := False;
        exit;  -- do not continue the analysis of the other transitions
      end if;
      if TmpResult = Aborted then
        Result := Aborted;
        -- continue the analysis in case we can still prove a failure
      end if;
    else
      set_explanation <$AX\gamma\ \phi$, CurrentState> → (Transition.Label,<>);
      Result := False;
      exit;
    end if;
  end loop;
  save <$AX\gamma\ \phi$, CurrentState> → Result;
  return;
end EvalForallNext;
```

Fig. 4. Universally quantified ‘next’ operator evaluation schema.

If for each transition we find a successful substitution we return `True`, otherwise, if for some transition we are not able to verify its success, we return `Aborted`. In any case, the evaluation result is *saved* and then returned (lines `50-51`).

The 'next' operator of the logic is quite simple and for this reason it has been chosen to show the complexities introduced by the incremental bounded evaluation, by the support of parametric formulae, and by the need to collect data for the final explanation of the evaluation result. The main complexity of the logical verification algorithm, however, is encountered when we need to evaluate a recursive operator, like the '*until*'. To show how to manage this additional complexity we present in Figure 5 the simplified schema of the evaluation algorithm for the existentially quantified 'until' operator. To keep the schema understandable, we abstract away from already seen details about existence of evaluation depth limits, dynamic instantiation of parametric subformulae, and explanation-oriented data collection. Also, parameters `EDepth` and `MaxDepth` are left for compatibility with the schema in Figure 3, although they are not actually used since all the 'bounded' aspects of the evaluation have been omitted for simplicity.

When the evaluation of a formula of the form $E(\phi_1 \,{}_{\chi}U_{\gamma}\, \phi_2)$ begins, it is firstly checked if a 'definitive' or 'temporary' result is already available (lines `05-17`). If a 'temporary' result for this evaluation already exists, then it means that the evaluation of this formula in this state has already been previously started at a more external recursion level (precisely at the recursion depth given by *somedepth*). In this case, we return not only the found temporary result, but also an updated recursion depth which identifies the original (still in progress) computation on which this temporary result actually depends. If no such result exists, then this evaluation has been encountered for the first time and the analysis of the formula continues.

If the current state is a final state, the formula is definitely `False`, therefore this value is saved and returned (lines `18-22`). Otherwise, we check if the current state satisfies the subformula ϕ_1. In case of failure, `False` is immediately saved and returned (lines `24-27`). Otherwise, we check if a transition from the current state exists whose label satisfies the action formula γ (line `31`) and whose target state satisfies the formula ϕ_2 (line `33`). If such a transition exists, then the evaluation of the original formula is definitely `True`, thus the result is saved and returned (lines `34-35`). Otherwise, we check if a transition satisfying the action formula χ exists such that its target state recursively satisfies the whole formula $E(\phi_1 \,{}_{\chi}U_{\gamma}\, \phi_2)$.

Before doing that, to avoid looping in presence of cycles in the model, the fact that this computation is now in progress must be saved together with its default (temporary) value `False` (according to the least fixed point semantics of the operator) and together with a reference to the current recursion depth of the evaluation of the formula (line `40`). Then, the search of transitions over which to apply recursion (lines `41-53`) can begin. Notice that the new recursive calls of procedure `EvalExistUntil` (lines `44-45`) are issued with an incremented `RecDepth` parameter (the value is stored in `LastDepth`) which allows to keep track of the current recursion depth.

If a successful application of recursion is found, then a definitive `True` result for the evaluation can be established and such value is immediately saved and returned (lines `47-48`). If no successful transition (or application of recursion) is found, there are two possibilities: either all the `False` results returned by the various recursive calls are 'definitive' values, in which case the evaluation definitely returns `False`, or some of the returned `False` results are just temporary values actually depending from outer level recursive evaluations which are still in progress. These temporary values can be identified because they are associated to the value stored in `LastDepth` that is lower than the current value of `RecDepth`. In this case, just a temporary `False` value is returned, together with a `MinDepth` value corresponding to the outmost level of recursion on which the result depends (lines `55-57`).

There are some important aspects, related to the behavior of the *update* operation that are not explicitly expressed by the shown algorithm. When a 'definitive' `False`/`True` value is saved for the current evaluation at a certain level of recursion depth (e.g. line `59`) each other previously saved temporary value (related to the result of deeper nested recursions) which happens to depend on the current recursion depth is replaced by the current definitive result. Moreover, at the end of the

```
procedure EvalExistUntil(E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState,EDepth,MaxDepth,
                              in out RecDepth, out Result) is
  MinDepth, LastDepth: Natural := RecDepth;
begin
  if <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ True exists then
    Result := True;      -- found definitive True value
    return;
  end if;
  if <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ False exists then
    Result := False;    -- found definitive False value
    return;
  end if;
  if <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ <False, somedepth> exists then
    Result := False;      -- found temporary False value
    RecDepth := somedepth;
    return;
  end if;
  if OutgoingTransitions(CurrentState) is empty then
    save <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ False;    -- save definitive
    Result := False;
    return;
  end if;
  Eval($\phi1$ , CurrentState,EDepth,MaxDepth,Result);
  if Result = False then
    save <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ False; -- save definitive
    return;
  end if;
  -- check the termination condition
  Result := False;
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, $\gamma$) then
      Eval($\phi2$, Transition.TargetState,EDepth+1,MaxDepth,Result);
      if Result = True then
        save <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ True; -- save definitive
        return;
      end if;
    end if;
  end loop;
  -- check for possible continuation of recursion
  save <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState>$\rightarrow$ <False,RecDepth>; -- save temporary
  foreach Transition in OutgoingTransitions(CurrentState) do
    if Satisfies(Transition.Label, $\chi$) then
     LastDepth := RecDepth+1;
     EvalExistUntil(E $\phi1$ $\chi$ U $\gamma$ $\phi2$,
       Transition.TargetState,EDepth+1,MaxDepth,LastDepth,Result);
     if Result = True then
       save <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ True; -- save definitive
       return;
     elsif Result = False and LastDepth < MinDepth then
       MinDepth := LastDepth;
     end if;
    end if;
  end loop;
  if MinDepth < RecDepth then
   update <E $\phi1$ $\chi$ U $\gamma$ $\phi2$,CurrentState>$\rightarrow$ <False,MinDepth>; -- temporary
   RecDepth := MinDepth;
   return;
  else
   update <E $\phi1$ $\chi$ U $\gamma$ $\phi2$, CurrentState> $\rightarrow$ False; -- save definitive
   return;
  end if;
end EvalExistUntil;
```

Fig. 5. Existentially quantified ‘until’ operator evaluation schema.

Table II. COWS syntax

Killer labels: $k, k', \ldots$	*Elements (Killer labels/Variables/Names):* $e, e', \ldots$
Expressions: $\epsilon, \epsilon', \ldots$	*Variables/Names:* $u, u', \ldots$
Variables: $x, y, \ldots$	*Variables/Values:* $w, w', \ldots$
Values: $v, v', \ldots$	
Names: $n, m, \ldots$	*Endpoints:*
Partners: $p, p', \ldots$	without variables: $p \bullet o, \ldots$
Operations: $o, o', \ldots$	may contain variables: $u \bullet u', \ldots$

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

loop, whenever an updated, but still temporary, value replaces (line 55) the initial temporary value (line 40) for the current computation (hence the initially saved `RecDepth` is replaced with a smaller `MinDepth`), also all the other already saved temporary results which refer to the original value of `RecDepth` must be updated by replacing this value with the value of `MinDepth`.

When input models are finite states, the overall complexity of the SocL model-checking algorithm, in case of non-parametric formulae, is comparable to that of the best on-the-fly model checking algorithms [Stirling and Walker 1989; Bhat et al. 1995; Fernandez et al. 1996]: it is then linear with respect to the size of the state space and the number of operators in the formula. Of course, in case of parametric formulae, it will also depend on the number of instantiations of all parametric subformulae.

4. COWS: A CALCULUS FOR ORCHESTRATION OF WEB SERVICES

COWS is a process calculus for specifying and combining services that has been recently developed inside the EU project Sensoria. Its design has been influenced by some principles underlying the OASIS standard WS-BPEL for orchestration of web services. In fact, COWS allows concurrent service instances to share (part of) the state, permits programming stateful interactions by correlating different actions, and enables management of long-running transactions.

In this section, we report the syntax of COWS and explain its semantics in a step-by-step fashion while modelling a bank service which is part of the automotive scenario of Section 6. We refer the interested reader to Tiezzi [2009] for an in-depth formal account of COWS's semantics, for many examples illustrating peculiarities and expressiveness of the calculus, and for comparisons with other process-based and orchestration formalisms.

4.1. Syntax

The syntax of COWS is presented in Table II. It is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels*, the set of *values* and the set of *variables*. The set of values is left unspecified; however, we assume that it includes the set of *names*, mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, 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, cannot be communicated). Partner names and operation names can be combined to designate *endpoints*, written $p \bullet o$, and can be communicated, but dynamically received names can only be used for service invocation. Indeed, endpoints of receive activities are identified statically because their syntax only permits using names and not variables.

Notation $\bar{\cdot}$ stands for tuples of homogeneous elements, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). As in the logic, we assume that variables in the same tuple are pairwise distinct. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[e_1, \ldots, e_n]\, s$ in place of $[e_1] \ldots [e_n]\, s$. We will write $I \triangleq s$ to assign a name I to the term s.

The delimitation operator permits to define and restrict the scope of an element: $[e]\, s$ binds the element e in the scope s. It is the only *binding* construct. In fact, to enable concurrent threads within each service instance to share (part of) the state, receive activities in COWS bind neither names nor variables, which is different from most process calculi. Instead, the range of application of the substitutions (of variables with values) generated by communication is regulated by the delimitation operator, that additionally permits to generate 'fresh' names (as the restriction operator of the π-calculus) and to delimit the field of action of kill activities. Thus, the occurrence of an element is *free* if it is not under the scope of a delimitation for it. Two terms are *alpha-equivalent* if one can be obtained from the other by consistently renaming bound elements. As usual, we identify terms up to alpha-equivalence.

4.2. Example: a bank service

To informally explain the semantics of COWS we now model the bank service that is part of the case study described in Section 6. The COWS specification of the bank service is composed of two persistent subservices: *BankInterface*, that is publicly invocable by customers, and *CreditRating*, that instead is an 'internal' service that can only interact with *BankInterface*. To show the behaviour of the bank service we also consider the terms $Client_1$ and $Client_2$ that model a pair of mutually dependent requests for charging a customer's credit card with some amount. This mutual dependence is given by the requirement that both clients must agree to charge the credit card in order for the operation to successfully complete: if one of the charge requests by $Client_1$ and $Client_2$ fails, no effect is produced, with a sort of 'transactional' behaviour.

Thus, the COWS term representing the considered scenario is

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

The main operator is the *parallel composition* $\_ \mid \_$ that allows the different components to be concurrently executed and to interact with each other. The first occurrence of the *delimitation* operator $[\_]\_$ is used to declare that o_{check}, $o_{checkOK}$ and $o_{checkFail}$ are (operation) names known to the bank subservices, and only to them. 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. The second delimitation declares the killer label k: it is shared by both clients and is used to coordinate them for ensuring the transactional behaviour we mentioned above.

BankInterface and *CreditRating* are defined as follows:

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

Let us start commenting on *BankInterface*. The leading *receive-guarded prefix* operator $p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle.\_$ means that each interaction with the bank starts with a *receive* activity of the form $p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle$ corresponding to reception of a request emitted by one of the clients. Receives, together with *invokes*, written as $p \bullet o!\langle \epsilon_1, \ldots, \epsilon_m \rangle$, 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. $p \bullet o$ 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, the receive along the endpoint $p_{bank} \bullet o_{charge}$ initializes the variables x_{cust}, x_{cc}, x_{amount} and x_{id}, declared local to *BankInterface* by the delimitation operator, with data provided by one of the clients.

Whenever prompted by a client request, *BankInterface* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each instance forwards the request to *CreditRating*, by invoking the 'internal' operation o_{check} through the invoke activity $p_{bank} \bullet o_{check}!\langle x_{id}, x_{cc}, x_{amount} \rangle$, then waits for a reply on one of the other two internal operations $o_{checkOK}$ and $o_{checkFail}$, by exploiting the *receive-guarded choice* operator $\_+\_$, and finally sends the reply back to the client by means of a final invoke activity using the partner name of the client stored in the variable x_{cust}. In case of a positive answer, the possibility of revoking the request through invocation of operation o_{revoke} is enabled (in fact, should the other request fail, the customer charge operation should be cancelled in order to implement the wanted transactional behaviour). Revocation causes deletion of the reply to the client, if this has still to be performed. Indeed, execution of *kill* activities like $\mathbf{kill}(k')$ causes termination of all parallel terms inside the enclosing $[k']$ operator, that stops the killing effect. Notably, if an invocation along the endpoints $p_{bank} \bullet o_{checkOk}$, $p_{bank} \bullet o_{checkFail}$ or $p_{bank} \bullet o_{revoke}$ takes place after a certain number of service instances have been created, then it could be received by any of these instances. Hence, to synchronise with the proper instance, an appropriate customer datum stored in the variable x_{id} is exploited as a correlation value.

Service *CreditRating* takes care of checking clients' requests and decides if they can be authorized or not. For the sake of simplicity, the choice between approving or not a request is left here completely non-deterministic.

The customer processes are defined as follows:

$$\begin{array}{lll} Client_1 & \triangleq & p_{bank} \bullet o_{charge}!\langle p_C, 1234, 100, id_1 \rangle \\ & & \mid\ p_C \bullet o_{chargeOK}?\langle id_1 \rangle.\, s_1 + p_C \bullet o_{chargeFail}?\langle id_1 \rangle\, .(\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_2 \rangle|\!\} \mid \mathbf{kill}(k)\,) \\ Client_2 & \triangleq & p_{bank} \bullet o_{charge}!\langle p_C, 1234, 200, id_2 \rangle \\ & & \mid\ p_C \bullet o_{chargeOK}?\langle id_2 \rangle.\, s_2 + p_C \bullet o_{chargeFail}?\langle id_2 \rangle\, .(\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_1 \rangle|\!\} \mid \mathbf{kill}(k)\,) \end{array}$$

The two processes perform two requests in parallel for charging the credit card 1234 with the amounts 100 and 200. Two different values, id_1 and id_2, are used to correlate the response messages to the corresponding requests, and to coordinate compensation behaviours for revoking credit card payments if one of the two requests fails. Note that to this purpose $Client_1$ knows id_2 and viceversa, and this implements the required mutual dependence. $Client_i$'s continuation, for $i \in \{1, 2\}$, is s_i in case the operation is authorized, otherwise it consists of invoking the revocation operation of the bank instance corresponding to the other client's request and killing the other client's remaining behaviour by executing a kill activity. As we said before, execution of $\mathbf{kill}(k)$ forces termination of all parallel terms inside the enclosing $[k]$. It is also worth noticing that $\mathbf{kill}(k)$ runs *eagerly* with respect to the other parallel activities within the enclosing $[k]$. However, critical activities, such as e.g. the compensation activities $p_{bank} \bullet o_{revoke}!\langle id_1 \rangle$ and $p_{bank} \bullet o_{revoke}!\langle id_2 \rangle$, can be protected from the effect of a forced termination by application of the *protection* operator $\{\!|\_|\!\}$.

4.3. Computational steps and LTSs

The operational semantics of the language assigns an LTS to each COWS term (without free variables/labels). To define such LTSs, a set of inference rules is used that explain how terms can interact with their execution environment and which computational activities, i.e. forced termination or internal communication, their subterms can be engaged in. The LTS of a term has hence the term itself as initial state, the states reachable from the initial state through sequences of computational steps as set of states, and the set of such computational steps as transition relation (therefore, the operational semantics follows a 'reduction' style). As an important consequence of only considering actual computational steps instead of all potential steps, we have that in many cases the LTS associated to a service is finite-state, even when the replication operator is used to define persistent services (in particular, all LTSs considered in this paper are finite-state). Indeed, while service providers are often persistent, their clients usually have finite behaviours and make a finite number of requests, hence their interactions produce a finite number of service instances and all such terms are involved in a finite number of finite length computations.

An excerpt of the LTS associated by the operational semantics to the COWS term modelling the scenario illustrated in Section 4.2 is shown in Figure 6. In particular, transitions correspond to computational steps and are hence labelled by actions of the form †, in case of forced terminations, or of the form $p \bullet o\, \bar{v}$, in case of communication of values $\bar{v}$ along the endpoint $p \bullet o$. In this figure, as well as in the Figures 7–10, arrows of the form $\longrightarrow$ denote individual computational steps and arrows of the form $\cdots\cdots\!\succ$ denote further unspecified computational steps.

As an example, consider the computation where $Client_1$'s request succeeds while $Client_2$'s request fails (the states it involves are highlighted in the figure by a dark gray background). After the instances of *CreditRating* communicate these decisions to the corresponding instances of *BankInterface*, the term describing the scenario is

$$\begin{array}{l} [o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating} \\ \qquad\qquad \mid [k']\,(\, p_C \bullet o_{chargeOK}!\langle id_1\rangle \mid p_{bank} \bullet o_{revoke}?\langle id_1\rangle.\, \mathbf{kill}(k')\,) \\ \qquad\qquad \mid p_C \bullet o_{chargeFail}!\langle id_2\rangle\,) \\ \mid [k]\,(\, p_C \bullet o_{chargeOK}?\langle id_1\rangle.\, s_1 + p_C \bullet o_{chargeFail}?\langle id_1\rangle.(\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_2\rangle|\!\} \mid \mathbf{kill}(k)\,) \\ \qquad \mid p_C \bullet o_{chargeOK}?\langle id_2\rangle.\, s_2 + p_C \bullet o_{chargeFail}?\langle id_2\rangle.(\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_1\rangle|\!\} \mid \mathbf{kill}(k)\,)\,) \end{array}$$

and corresponds to state q_{18}. After $p_C \bullet o_{chargeOK}!\langle id_1\rangle$ has been processed by $Client_1$, we are in state q_{19} and the above term becomes

$$\begin{array}{l} [o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating} \\ \qquad\qquad \mid [k']\,(\, p_{bank} \bullet o_{revoke}?\langle id_1\rangle.\, \mathbf{kill}(k')\,) \\ \qquad\qquad \mid p_C \bullet o_{chargeFail}!\langle id_2\rangle\,) \\ \mid [k]\,(\, s_1 \\ \qquad \mid p_C \bullet o_{chargeOK}?\langle id_2\rangle.\, s_2 + p_C \bullet o_{chargeFail}?\langle id_2\rangle.(\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_1\rangle|\!\} \mid \mathbf{kill}(k)\,)\,) \end{array}$$

After $Client_2$ consumes the invocation $p_C \bullet o_{chargeFail}!\langle id_2\rangle$, we are in state q_{23} and the term becomes:

$$\begin{array}{l} [o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating} \\ \qquad\qquad \mid [k']\,(\, p_{bank} \bullet o_{revoke}?\langle id_1\rangle.\, \mathbf{kill}(k')\,)\,) \\ \mid [k]\,(\, s_1 \\ \qquad \mid \{\!|p_{bank} \bullet o_{revoke}!\langle id_1\rangle|\!\} \mid \mathbf{kill}(k)\,) \end{array}$$

Now, since kill activities must be executed eagerly, the enabled $\mathbf{kill}(k)$ is executed leading to state q_{24} and the term becomes:

$$\begin{array}{l} [o_{check}, o_{checkOK}, o_{checkFail}]\,(\, * \mathit{BankInterface} \mid * \mathit{CreditRating} \\ \qquad\qquad \mid [k']\,(\, p_{bank} \bullet o_{revoke}?\langle id_1\rangle.\, \mathbf{kill}(k')\,)\,) \\ \mid [k]\, \{\!|p_{bank} \bullet o_{revoke}!\langle id_1\rangle|\!\} \end{array}$$

Notably, the invoke activity $p_{bank} \bullet o_{revoke}!\langle id_1\rangle$ is protected and, hence, can still be performed for compensating the effect of the $Client_1$'s (succeeded) request also after the kill has been executed.

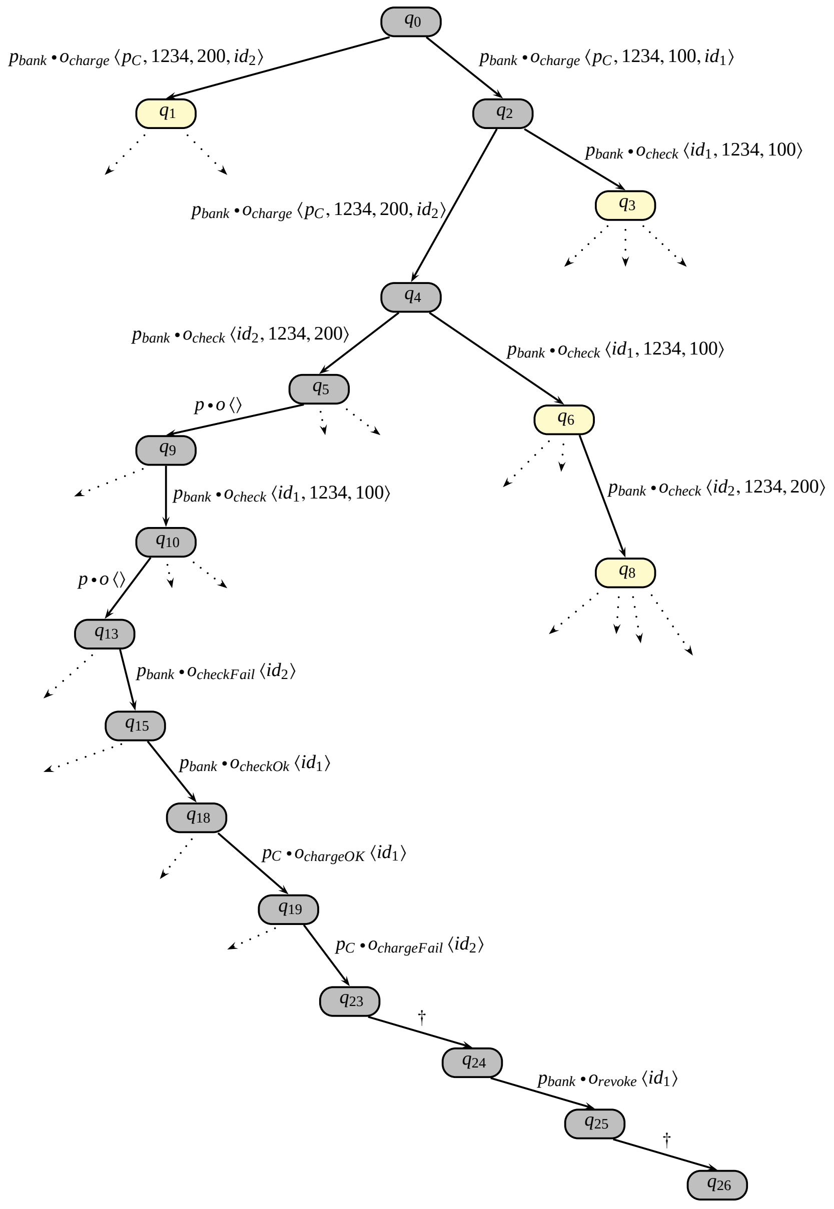

q_0
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 200, id_2 \rangle$
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 100, id_1 \rangle$
q_1
q_2
$p_{bank} \bullet o_{check} \langle id_1, 1234, 100 \rangle$
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 200, id_2 \rangle$
q_3
q_4
$p_{bank} \bullet o_{check} \langle id_2, 1234, 200 \rangle$
$p_{bank} \bullet o_{check} \langle id_1, 1234, 100 \rangle$
q_5
$p \bullet o \langle \rangle$
q_6
q_9
$p_{bank} \bullet o_{check} \langle id_1, 1234, 100 \rangle$
$p_{bank} \bullet o_{check} \langle id_2, 1234, 200 \rangle$
q_{10}
$p \bullet o \langle \rangle$
q_8
q_{13}
$p_{bank} \bullet o_{checkFail} \langle id_2 \rangle$
q_{15}
$p_{bank} \bullet o_{checkOk} \langle id_1 \rangle$
q_{18}
$p_C \bullet o_{chargeOK} \langle id_1 \rangle$
q_{19}
$p_C \bullet o_{chargeFail} \langle id_2 \rangle$
q_{23}
$\dagger$
q_{24}
$p_{bank} \bullet o_{revoke} \langle id_1 \rangle$
q_{25}
$\dagger$
q_{26}

Fig. 6. LTS for the bank scenario.

Therefore, the term evolves to

$$\begin{aligned}&[o_{check}, o_{checkOK}, o_{checkFail}]\,(\,*\,BankInterface \mid *\,CreditRating\\ &\qquad\qquad\qquad \mid\ [k']\ \mathbf{kill}(k')\,)\\ &\mid\ [k]\,\mathbf{0}\end{aligned}$$

which corresponds to state q_{25}. Finally, after execution of $\mathbf{kill}(k')$ we reach state q_{26} and get the stuck term

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

5. CMC: A VERIFICATION ENVIRONMENT FOR COWS SPECIFICATIONS

CMC is a verification environment for SocL formulae over COWS specifications of services. In this section we illustrate the principles underlying its design and its use.

5.1. From LTS to L^{2}TS

To implement CMC by fully exploiting the model checker engine for SocL, we enrich the LTS modelling the semantics of a term with a function labelling each state with the set of communication activities that any active subterm of the COWS term corresponding to that state can potentially perform immediately. These information, indeed, are not present in the LTSs' transition relations which only take into account actual computational steps, but can be easily retrieved from the syntax of terms. They are important to effectively describe/verify service properties since they correspond to atomic propositions, in as much as computational steps correspond to actions of the logic. Of course, the transformation of LTSs into L^{2}TSs preserves the structure (i.e. set of states, initial state and transition relation) of the original LTS. In particular, for both transition systems, transitions correspond to computational steps and are hence labelled by actions of the form † or of the form $p \bullet o\, \bar{v}$.

To transform an LTSs into an L^{2}TS we rely on the auxiliary function $\mathcal{L}(s)$, which returns the set of communication activities that the term s can potentially perform immediately, i.e. not syntactically preceded by other activities and not preempted by parallel kill activities. Formally, it is inductively defined as follows:

$$\mathcal{L}(p \bullet o!\bar{\epsilon}) = \{\, p \bullet o!\bar{\epsilon} \,\} \qquad \mathcal{L}(p \bullet o?\bar{w}\,.\, s) = \{\, p \bullet o?\bar{w} \,\} \qquad \mathcal{L}(g + g') = \mathcal{L}(g) \cup \mathcal{L}(g')$$

$$\mathcal{L}(\mathbf{0}) = \mathcal{L}(\mathbf{kill}(k)) = \emptyset \qquad \mathcal{L}(\{\!|s|\!\}) = \mathcal{L}(*\, s) = \mathcal{L}(s) \qquad \mathcal{L}(s \mid s') = \mathcal{L}(s) \cup \mathcal{L}(s')$$

$$\mathcal{L}([e]\, s) = \mathcal{L}(s) \quad \text{if } e \neq k \qquad \mathcal{L}([k]\, s) = \begin{cases} \mathcal{L}(s) & \text{if noKill}(s,k) \\ \emptyset & \text{otherwise} \end{cases}$$

where predicate noKill(s, k) (also exploited by the rules of the COWS's operational semantics [Tiezzi 2009]), which holds true if s cannot immediately perform an activity $\mathbf{kill}(k)$, is inductively defined as follows:

$$\text{noKill}(\mathbf{kill}(k), k) = \mathbf{false} \qquad \text{noKill}(s \mid s', k) = \text{noKill}(s,k) \wedge \text{noKill}(s',k)$$

$$\text{noKill}(\mathbf{kill}(k'), k) = \mathbf{true} \quad \text{if } k \neq k' \qquad \text{noKill}([e]\, s, k) = \text{noKill}(s,k) \quad \text{if } e \neq k$$

$$\text{noKill}(u \bullet u'!\bar{\epsilon}, k) = \mathbf{true} \qquad \text{noKill}([k]\, s, k) = \mathbf{true}$$

$$\text{noKill}(g, k) = \mathbf{true} \qquad \text{noKill}(\{\!|s|\!\}, k) = \text{noKill}(*\, s, k) = \text{noKill}(s,k)$$

All cases of the above definitions are quite straightforward. We only remark that $\mathcal{L}([k]\, s)$ returns $\emptyset$ when noKill(s, k) is false because, due to the prioritised semantics of kill activities, the execution of the communication activities within s is blocked, regardless of such activities are protected or not.

The next example shows how the function $\mathcal{L}(\_)$ is exploited during the transformation process to retrieve from syntactical terms state information to be added to an LTS.

Example 5.1. Consider the following COWS term:

$$s \triangleq p \bullet o!\langle v\rangle \mid [x]\,(\, p \bullet o?\langle x\rangle \mid p \bullet o'?\langle x\rangle.\, [y]\, p \bullet o''?\langle x, y\rangle\,)$$

The corresponding LTS and L^{2}TS are as in Figure 7. The LTS on the left hand side says that the term s can actually perform only the computational step corresponding to the communication of the value v along the endpoint $p \bullet o$. However, besides the activities $p \bullet o!\langle v\rangle$ and $p \bullet o?\langle x\rangle$, the COWS term can potentially perform also the receive activity $p \bullet o'?\langle x\rangle$. Thus, to record this information, the state q_0, corresponding to the term s, of the L^{2}TS on the right hand side is labelled by a set containing all the three potential activities (since $\mathcal{L}(s) = \{\, p \bullet o!\langle v\rangle,\ p \bullet o?\langle x\rangle,\ p \bullet o'?\langle x\rangle \,\}$). Similarly, the state q_1, corresponding to the term $s' \triangleq p \bullet o'?\langle v\rangle.\, [y]\, p \bullet o''?\langle v, y\rangle$, is labelled by the merely

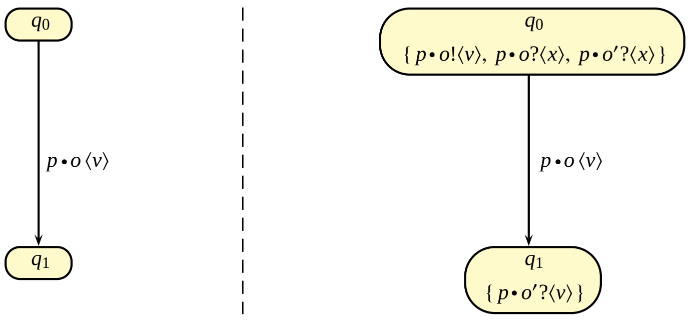

q_0
$p \bullet o \langle v \rangle$
q_1
q_0
$\{ p \bullet o!\langle v \rangle,\ p \bullet o?\langle x \rangle,\ p \bullet o'?\langle x \rangle \}$
$p \bullet o \langle v \rangle$
q_1
$\{ p \bullet o'?\langle v \rangle \}$

Fig. 7. From LTS to L^2TS.

potential receive activity $p \bullet o'?\langle v \rangle$ (since $\mathcal{L}(s') = \{ p \bullet o'?\langle v \rangle \}$). Notably, the receive $p \bullet o''?\langle v, y \rangle$ is missing in the L^2TS because it cannot be immediately performed by the COWS term, neither before nor after the computation identified by the transition labelled by $p \bullet o \langle v \rangle$.

It is worth remarking that, according to the on-the-fly verification approach described in Section 3, both the LTS and the corresponding L^2TS associated to a COWS term are generated on-the-fly, on demand of the model checking engine. Therefore, depending on the formula to be evaluated, only a fragment of the overall state space of such models might need to be generated.

5.2. From concrete L^2TS to abstract L^2TS

Now, consider again the bank service presented in the previous section. Its LTS is shown in Figure 8, while the associated L^2TS is shown in Figure 9. In the Figures 8–10, arrows of the form $\dashrightarrow$ denote multi-step computations. Transitions of both systems are labelled by 'concrete' information generated by the operational rules of the calculus. Since we are interested in verifying abstract properties of services, such as those shown in Section 2, we need to abstract away unnecessary details from the L^2TS model of a service. This is done by using a set of suitable abstraction rules that permit to replace concrete actions on the transitions with 'abstract' actions of SocL, i.e. $request(i, c)$, $responseOk(i, c)$, $responseFail(i, c)$, $cancel(i, c)$ and $undo(i, c)$. Similar rules permit to replace the concrete activities labelling the states with predicates of SocL, e.g. $accepting\_request(i)$, $accepting\_cancel(i, c)$, and $accepting\_undo(i, c)$. Of course, in doing these further transformations, different concrete actions can be mapped into the same SocL action. Moreover, the transformations may involve only those concrete actions/activities that are considered worthwhile to be observed to carry on the analysis of interest. Those that are not replaced by their abstract counterparts may not be observed.

The abstraction procedure must however preserve those names and values occurring within concrete actions/activities of COWS specifications that are important to express properties of service behaviour. To capture such names and values, transformation rules can make use of 'metavariables', written as names starting with the character "\$"; otherwise, they can use the wildcard " $*$ ". To minimise the introduction of new notations, we write $\underset{\cdot}{v}$ to indicate that v can be either a value, or a metavariable, or the wildcard (this notation also applies to tuples, actions and predicates with a similar meaning). Take also into account that the wildcard can only occur in the left hand side of the abstraction rules. Formally, abstraction rules follow the templates:

$$\begin{array}{rlcll} Action & \underset{\cdot}{p} \bullet \underset{\cdot}{o}\, \bar{\underset{\cdot}{v}} & \rightarrow & \underset{\cdot}{\alpha} & (1) \\ State & \underset{\cdot}{p} \bullet \underset{\cdot}{o}?\bar{\underset{\cdot}{w}} & \rightarrow & \underset{\cdot}{\pi} & (2) \\ State & \underset{\cdot}{p} \bullet \underset{\cdot}{o}!\bar{\underset{\cdot}{v}} & \rightarrow & \underset{\cdot}{\pi} & (3) \end{array}$$

where $\underset{\cdot}{\alpha}$ is a closed action and $\underset{\cdot}{\pi}$ is a closed atomic proposition of the logic SocL (except for, possibly, the occurrence of some of the metavariables introduced in the left hand side of the same

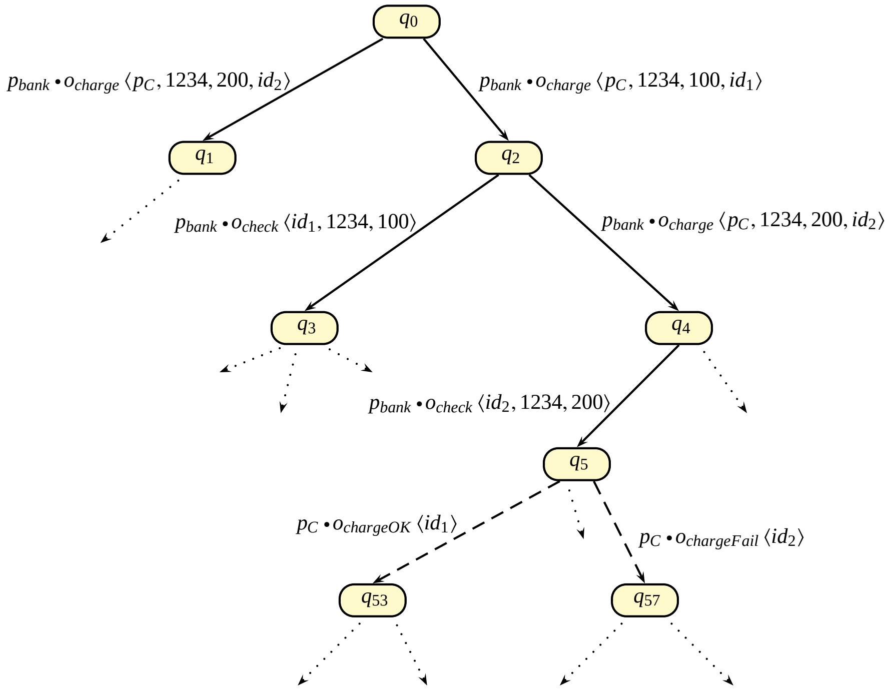

q_0
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 200, id_2 \rangle$
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 100, id_1 \rangle$
q_1
q_2
$p_{bank} \bullet o_{check} \langle id_1, 1234, 100 \rangle$
$p_{bank} \bullet o_{charge} \langle p_C, 1234, 200, id_2 \rangle$
q_3
q_4
$p_{bank} \bullet o_{check} \langle id_2, 1234, 200 \rangle$
q_5
$p_C \bullet o_{chargeOK} \langle id_1 \rangle$
$p_C \bullet o_{chargeFail} \langle id_2 \rangle$
q_{53}
q_{57}

Fig. 8. Excerpt of the LTS for the bank scenario.

rule). Rules following the template (1) apply to concrete actions of transitions, while the remaining ones apply to concrete activities labelling states.

To define the effect of the application of abstraction rules to an L^{2}TS, we exploit an auxiliary, partial function $\mathsf{match}_t(\_,\_)$ that checks the matching between tuples of the form $(\underset{.}{p}, \underset{.}{o}, \underset{.}{\bar{v}})$, drawn from the left hand sides of abstraction rules, and tuples of the form $(p, o, \bar{w})$, drawn from concrete actions/activities, and, in case of success, returns a substitution. This function is defined by the following rules:

$$\mathsf{match}_t(v, v) = \emptyset \qquad \mathsf{match}_t(*, v) = \emptyset \qquad \mathsf{match}_t(\$n, v) = \{\$n/v\}$$

$$\mathsf{match}_t(*, x) = \emptyset \qquad \frac{\mathsf{match}_t(\underset{.}{v}_1, w_1) = \rho_1 \quad \mathsf{match}_t(\underset{.}{\bar{v}}_2, \bar{w}_2) = \rho_2}{\mathsf{match}_t((\underset{.}{v}_1, \underset{.}{\bar{v}}_2), (w_1, \bar{w}_2)) = \rho_1 \uplus \rho_2}$$

When using the left hand side of a rule to build the tuple $\langle \underset{.}{p}, \underset{.}{o}, \underset{.}{\bar{v}} \rangle$, if any of $\underset{.}{p}$ or $\underset{.}{o}$ is missing, it is replaced by the wildcard, while if $\underset{.}{\bar{v}}$ is missing, it is replaced by one or more tuples of wildcards of appropriate length (as drawn from the COWS specification according to the tuples of values that can be exchanged along the endpoints matching $\underset{.}{p} \bullet \underset{.}{o}$). In practice, each abstraction rule applies to the largest possible set of concrete actions/activities according to function $\mathsf{match}_t(\_,\_)$. Omitting any of the elements in the left hand side of the rule corresponds then to enlarging its application domain.

For example, the abstract L^{2}TS of the bank scenario shown in Figure 10 is obtained by applying to the concrete L^{2}TS of Figure 9 the following abstraction rules:

$$\begin{array}{rlcl}
\mathit{Action} & p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle & \rightarrow & \mathit{request}(\mathit{charge}, \$id) \\
\mathit{Action} & * \bullet o_{chargeOK}, \langle \$id \rangle & \rightarrow & \mathit{responseOk}(\mathit{charge}, \$id) \\
\mathit{Action} & * \bullet o_{chargeFail}, \langle \$id \rangle & \rightarrow & \mathit{responseFail}(\mathit{charge}, \$id) \\
\mathit{State} & p_{bank} \bullet o_{charge}? & \rightarrow & \mathit{accepting\_request}(\mathit{charge})
\end{array}$$

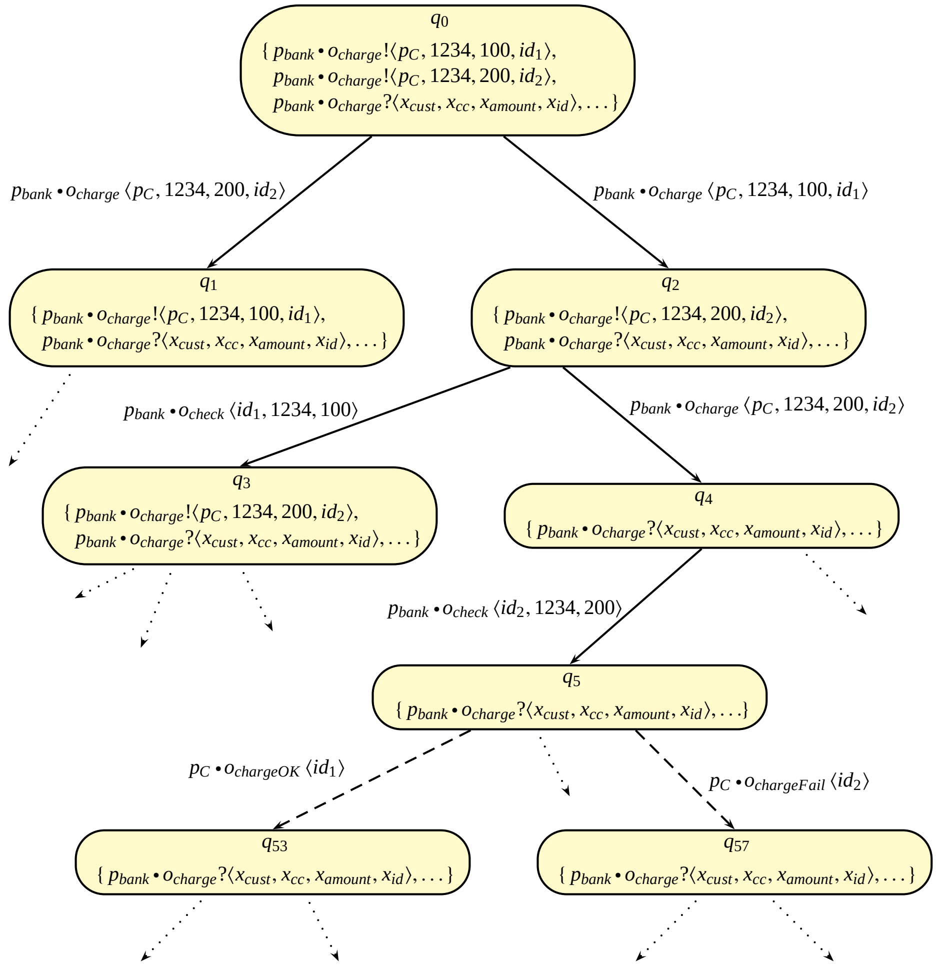

q_0
$\{\, p_{bank} \bullet o_{charge}!\langle p_C, 1234, 100, id_1\rangle,$
$p_{bank} \bullet o_{charge}!\langle p_C, 1234, 200, id_2\rangle,$
$p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
$p_{bank} \bullet o_{charge}\,\langle p_C, 1234, 200, id_2\rangle$
$p_{bank} \bullet o_{charge}\,\langle p_C, 1234, 100, id_1\rangle$
q_1
$\{\, p_{bank} \bullet o_{charge}!\langle p_C, 1234, 100, id_1\rangle,$
$p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
q_2
$\{\, p_{bank} \bullet o_{charge}!\langle p_C, 1234, 200, id_2\rangle,$
$p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
$p_{bank} \bullet o_{check}\,\langle id_1, 1234, 100\rangle$
$p_{bank} \bullet o_{charge}\,\langle p_C, 1234, 200, id_2\rangle$
q_3
$\{\, p_{bank} \bullet o_{charge}!\langle p_C, 1234, 200, id_2\rangle,$
$p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
q_4
$\{\, p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
$p_{bank} \bullet o_{check}\,\langle id_2, 1234, 200\rangle$
q_5
$\{\, p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
$p_C \bullet o_{chargeOK}\,\langle id_1\rangle$
$p_C \bullet o_{chargeFail}\,\langle id_2\rangle$
q_{53}
$\{\, p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$
q_{57}
$\{\, p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle, \ldots \}$

Fig. 9. Excerpt of the L^{2}TS for the bank scenario with concrete labels.

Thus, as a consequence of the application of the first rule, the concrete action $p_{bank} \bullet o_{charge}\,\langle p_C, 1234, 200, id2\rangle$ that matches the left hand side of the rule and produces the substitution $\{\$id/id2\}$, is replaced by the SocL abstract action $request(charge, id2)$ that is obtained by applying the produced substitution to the right hand side of the rule. Similarly, as a consequence of the application of the last rule, the concrete action $p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{id}\rangle$ labelling state q_0 and matching the left hand side of the rule is replaced by the SocL atomic proposition $accepting\_request(charge)$. Notably, concrete actions corresponding to (internal) communications between the bank subservices are not transformed and, thus, become unobservable (the corresponding action in the abstract L^{2}TS is $\emptyset$).

Of course, the sets of transformation rules are not defined once and for all, but are application-dependent and, thus, must be defined from time to time. Indeed, they embed information, like the intended semantics of each action and the predicates on the states, that are not coded into the COWS specification. Therefore, to verify abstract properties of a service-oriented system, one has to provide a specification of the system consisting of a COWS term and a set of abstraction rules; then, the construction of the associated L^{2}TS can be automatically performed. One advantage of keeping the specification separated in two parts is that, once the concrete part (i.e. the COWS term) is

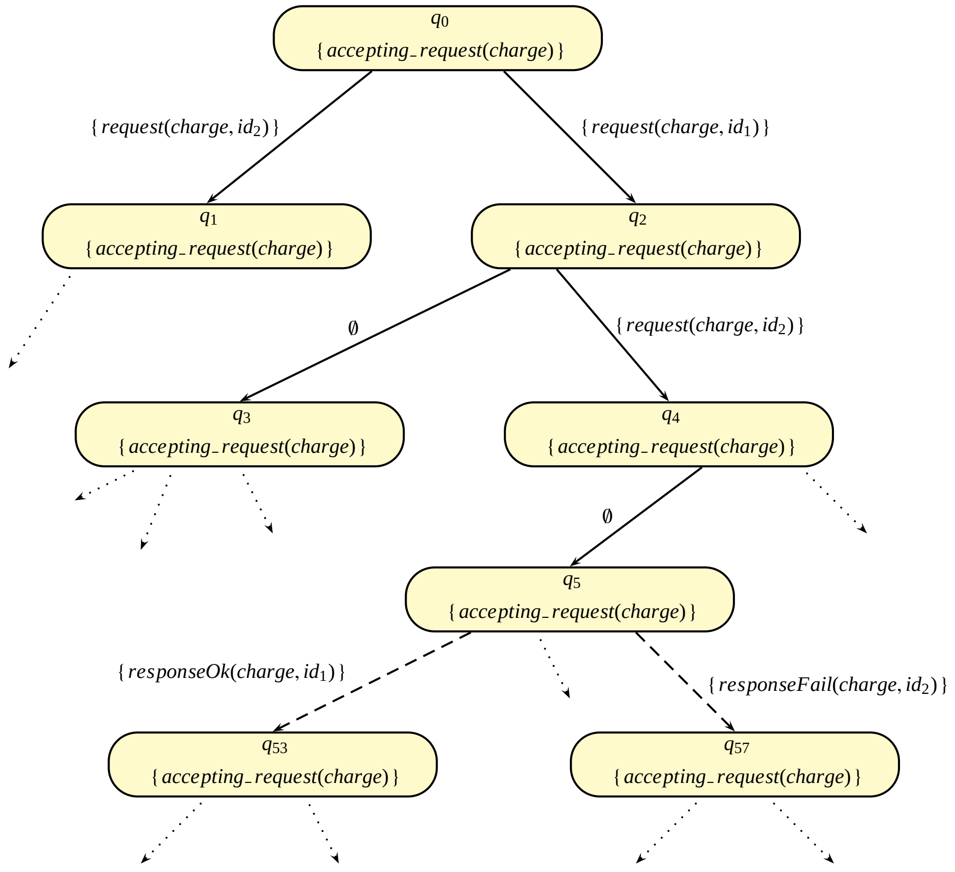

q_0 { $accepting\_request(charge)$ }
{ $request(charge, id_2)$ }
{ $request(charge, id_1)$ }
q_1 { $accepting\_request(charge)$ }
q_2 { $accepting\_request(charge)$ }
$\emptyset$
{ $request(charge, id_2)$ }
q_3 { $accepting\_request(charge)$ }
q_4 { $accepting\_request(charge)$ }
$\emptyset$
q_5 { $accepting\_request(charge)$ }
{ $responseOk(charge, id_1)$ }
{ $responseFail(charge, id_2)$ }
q_{53} { $accepting\_request(charge)$ }
q_{57} { $accepting\_request(charge)$ }

Fig. 10. Excerpt of the L^{2}TS for the bank scenario with abstract labels.

given, many different abstractions could be considered in order to analyse the system with respect to different views (possibly, at different levels of abstraction). Notably, our notion of abstraction only aims at making the specification and analysis of service-oriented systems simpler and more flexible, differently from homonymous approaches (see, e.g., Clarke et al. [1994], Dams et al. [1997]) that instead aim at reducing the state space of models. Indeed, as we said before, our abstraction preserves the structure of the model under analysis, i.e. the concrete LTS and its corresponding abstract L^{2}TS have the same numbers of states and edges.

5.3. COWS model checking

In the end, CMC embeds the SocL model checker presented in Section 3 below a layer that automates the (on-the-fly) generation of abstract L^{2}TSs from services described in COWS, thus enabling the verification of SocL formulae over COWS terms (see Figure 11). By the way, CMC can also be used as an *interpreter* for COWS: it takes a COWS term as an input and analyses it syntactically; if the analysis succeeds, the tool allows the user to interactively explore the computations arising from the term. Figure 12 shows a screenshot of the CMC's interface, where a COWS term, a set of abstraction rules and a SocL formula are represented.

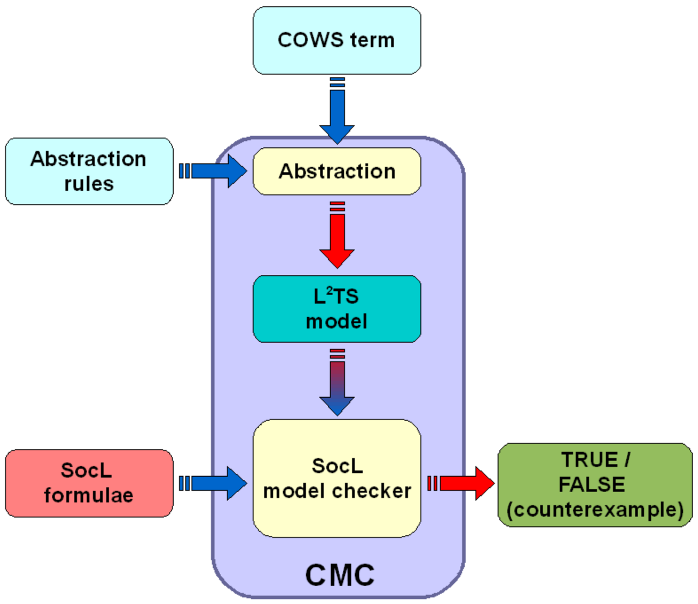


Fig. 11. An high-level overview of CMC.

A prototypical version[2] of the tool can be experimented via the web interface at address `http://fmt.isti.cnr.it/cmc/`, or by downloading from the same address a binary distribution (for Linux, Solaris, Windows or Mac OS X platforms).

The CMC core consists of a command-line-oriented version of the model checker, which is a stand-alone program written in Ada. This executable core is wrapped with a set of CGI scripts handled by a web server to provide it with an HTML-oriented GUI. The core has also been wrapped into an appropriate set of Java classes, thus obtaining a Java stand-alone application equipped with an handy graphical interface and a plugin for the Eclipse environment.

The current version of CMC is not targeted to the verification of extremely large systems, although our tool can perform an exhaustive state analysis of systems with several tens of thousands of states. Notably, due to the on-the-fly nature of its model checking procedure, CMC does not necessarily need to generate and explore the whole state space. This feature improves CMC performance and makes it able to also deal with infinite state systems.

6. CASE STUDY: AN AUTOMOTIVE SCENARIO

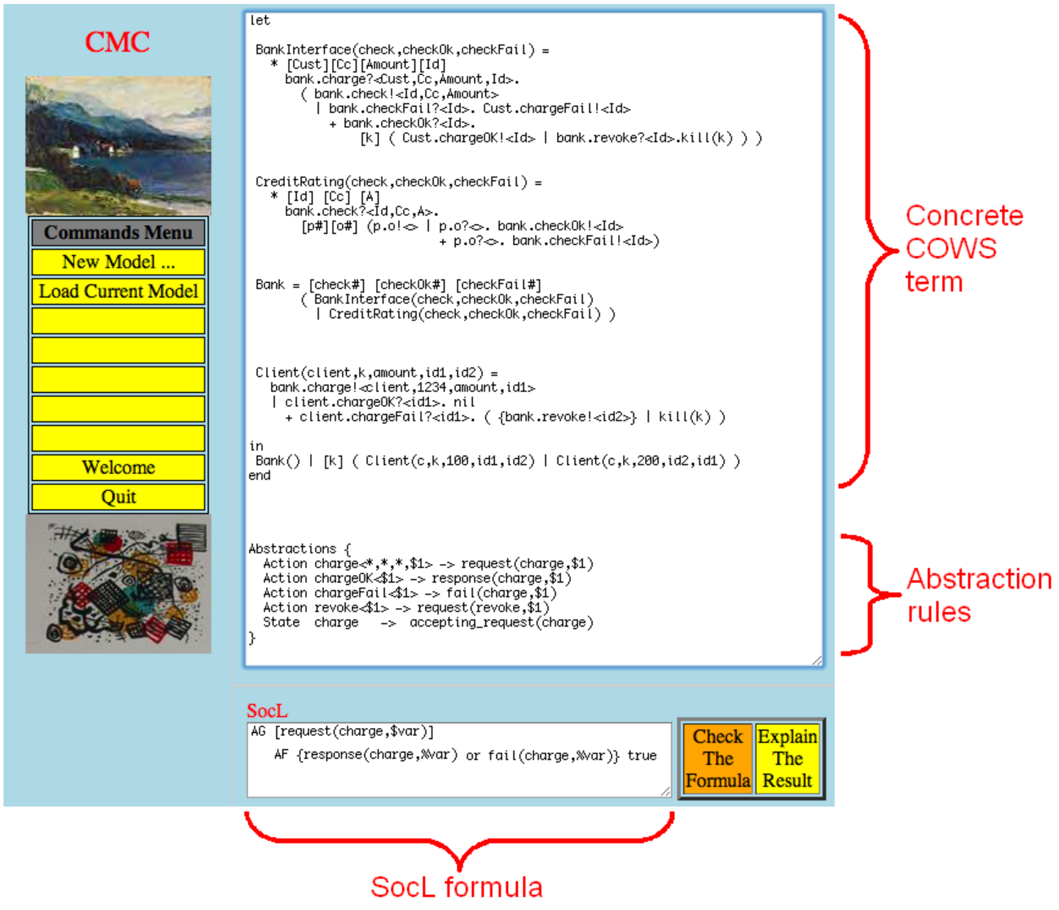
In this section, we introduce the case study that will be used to illustrate our verification methodology. We start by providing an informal description of the scenario by also making use of UML-like diagrams, then we present a formal specification in COWS.

6.1. Scenario description

The scenario is inspired to one of the case studies in the area of automotive systems defined and analysed within the EU project Sensoria [Koch 2007] and describes some functionalities that will be likely available in the near future. A brief description follows.

> While a driver is on the road with her/his car, the vehicle's *sensors monitor* reports a severe failure, which results in the car being no longer driveable. At this point, the *failure handler* installed in the in-vehicle computer system invokes an *assistance* service

[2]At the moment of writing, the current version of CMC is v0.7q.



```
let

 BankInterface(check,checkOk,checkFail) =
   * [Cust][Cc][Amount][Id]
     bank.charge?<Cust,Cc,Amount,Id>.
       ( bank.check!<Id,Cc,Amount>
         | bank.checkFail?<Id>. Cust.chargeFail!<Id>
           + bank.checkOk?<Id>.
               [k] ( Cust.chargeOK!<Id> | bank.revoke?<Id>.kill(k) ) )


 CreditRating(check,checkOk,checkFail) =
   * [Id] [Cc] [A]
     bank.check?<Id,Cc,A>.
       [p#][o#] (p.o!<> | p.o?<>. bank.checkOk!<Id>
                        + p.o?<>. bank.checkFail!<Id>)


 Bank = [check#] [checkOk#] [checkFail#]
       ( BankInterface(check,checkOk,checkFail)
         | CreditRating(check,checkOk,checkFail) )



 Client(client,k,amount,id1,id2) =
   bank.charge!<client,1234,amount,id1>
   | client.chargeOK?<id1>. nil
     + client.chargeFail?<id1>. ( {bank.revoke!<id2>} | kill(k) )

in
 Bank() | [k] ( Client(c,k,100,id1,id2) | Client(c,k,200,id2,id1) )
end



Abstractions {
  Action charge<*,*,*,$1> -> request(charge,$1)
  Action chargeOK<$1> -> response(charge,$1)
  Action chargeFail<$1> -> fail(charge,$1)
  Action revoke<$1> -> request(revoke,$1)
  State  charge   ->  accepting_request(charge)
}
```

```
AG [request(charge,$var)]

   AF {response(charge,%var) or fail(charge,%var)} true
```

Fig. 12. A screenshot of CMC web interface.

that, in its turn, contacts some garage, car rental and towing truck services, and tries to order them. To be authorised to order services, the assistance service has to deposit on behalf of the owner of the car a security payment, which will be given back if ordering the services fails.

A UML-like activity diagram of the assistance service using UML4SOA, an UML profile for service-oriented systems [Mayer et al. 2008], is shown in Figure 13. As usual, bars denote fork and join nodes, while diamonds denote decision and merge nodes. The assistance service is instantiated by a request from an in-vehicle computer system, received through the UML action SevereFailureAssistance, and consequently orchestrates the other services to reach its goal. The request is uniquely identified by the value of the input parameter id, which is subsequently used for correlation purposes. Then, the created instance invokes the *bank* to charge the driver's credit card with the security deposit amount. This is modelled by the action CardCharge for charging the credit card whose number is provided as an output parameter of the action call. If the credit card charge fails (because, e.g., there are not enough funds in the driver's bank account), the driver is informed by means of the FailureNotification action.

Services ordering is modelled by the UML actions OrderGarage, OrderTowTruck and RentCar. When the assistance service makes an appointment with the garage, the diagnostic data are automatically transferred to the garage, which could then be able, e.g., to identify the spare parts needed

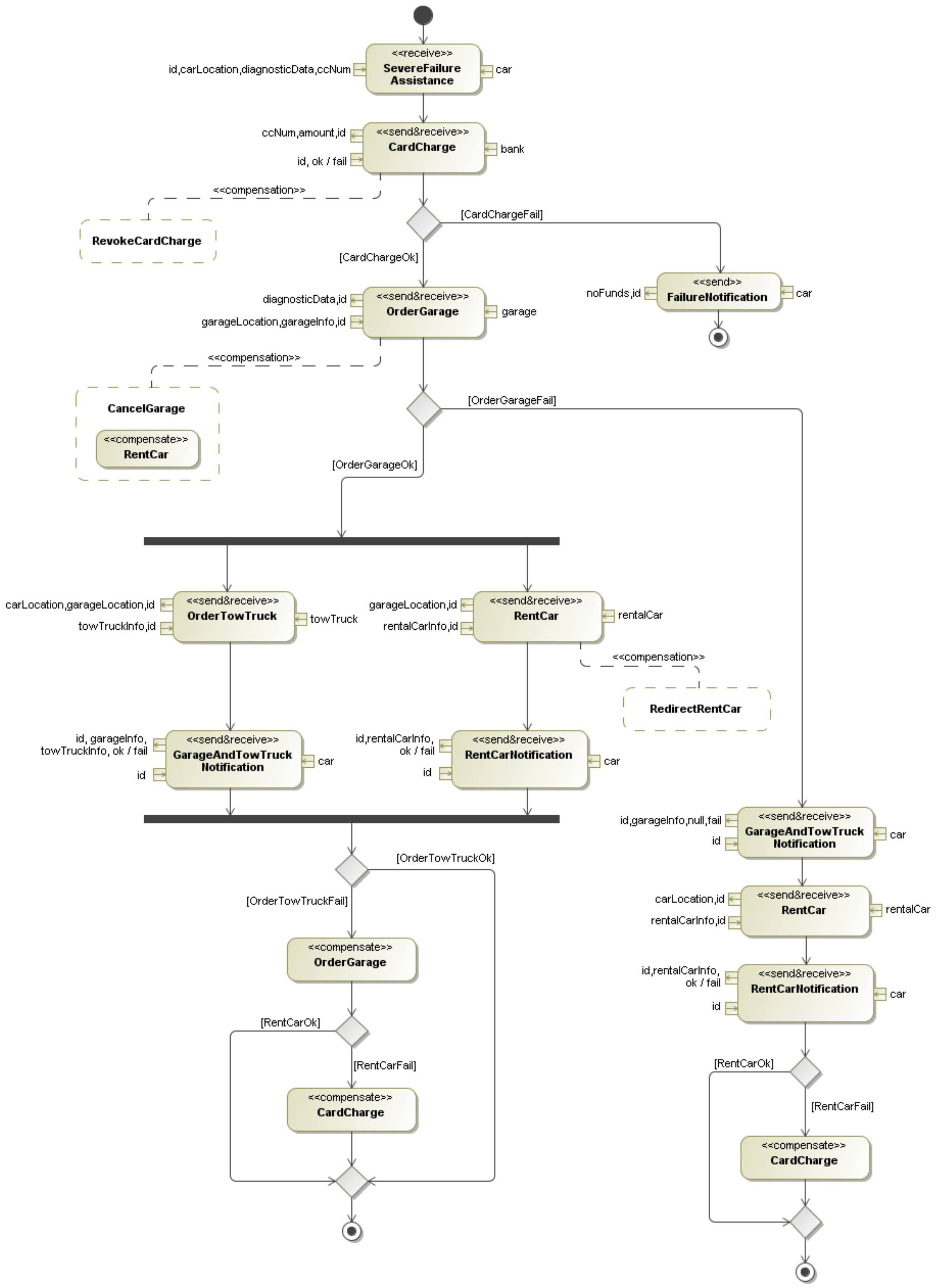


Fig. 13. The assistance service.

to perform the repair. If the order of the garage fails, the assistance service tries to make an appointment with the rental car service, by indicating the location of the stranded vehicle, where the car has to be handed over to the driver. Instead, if the order of the garage succeeds, the service concurrently tries to make an appointment with the rental service, by indicating the garage location as destination for the rental car, and with the towing service, providing the locations of the stranded vehicle and of the garage in order to tow the vehicle to the garage.

Besides interactions among services, the workflow described in Figure 13 also includes activities using concepts developed for long running business transactions (in e.g. OASIS WSBPEL TC [2007]). These activities entail fault and compensation handling, sort of specific activities attempting to reverse the effects of previously committed activities, which are an important aspect of SOC applications. According to the UML4SOA profile, the installation of a compensation handler (represented by a dashed box) is modelled by a dashed edge labelled by the stereotype ≪compensation≫, and its activation by an activity labelled by ≪compensate≫. Specifically, in the considered scenario:

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

6.2. COWS specification

The automotive scenario can be modelled by the following COWS term:

$$Car_1 \mid Car_2 \mid Assistance \mid Bank \mid OnRoadRepairServices$$

where, to better illustrate the use of correlation, we consider two cars reporting a severe failure. Notably, *Bank* is the COWS term $[o_{check}, o_{checkOK}, o_{checkFail}]\,(\,*\,BankInterface \mid *\,CreditRating\,)$ defined in Section 4.2.

The COWS term Car_i representing an in-vehicle computer system is defined as follows:

$$SensorsMonitor_i \mid GpsSystem_i \mid FailureHandler_i$$

When a severe failure (e.g. an engine failure) occurs, a signal (raised by $SensorsMonitor_i$) triggers the execution of the $FailureHandler_i$ and activates the corresponding 'recovery' service. $FailureHandler_i$, the most important component of the in-vehicle platform, is

$$\begin{array}{l} *\,[k, x_{diagnosticData}]\,(\,p_{car\_i} \bullet o_{engineFailure}?\langle x_{diagnosticData}\rangle.\, s_{engfail} \\ \qquad\qquad\qquad + \; p_{car\_i} \bullet o_{lowOilFailure}?\langle x_{diagnosticData}\rangle.\, s_{lowoil} \; + \; \ldots\,) \end{array}$$

This term picks one of those alternative recovery behaviours whose execution can start immediately. The recovery behaviour $s_{engfail}$ executed when an engine failure occurs is

$$\begin{array}{l} p_{gps\_i} \bullet o_{reqLocation}!\langle id_i\rangle \\ \mid [x_{carLocation}]\, p_{car\_i} \bullet o_{respLocation}?\langle id_i, x_{carLocation}\rangle. \\ \quad (\,p_{assistance} \bullet o_{severeFailure}!\langle p_{car\_i}, id_i, x_{carLocation}, x_{diagnosticData}, ccNum_i\rangle \\ \qquad \mid [x_{failure}]\, p_{car\_i} \bullet o_{failureNotification}?\langle x_{failure}, id_i\rangle.\, \mathbf{kill}(k) \\ \qquad \mid [x_{garageAndTowTruckResponse}, x_{garageInfo}, x_{towTruckInfo}] \\ \qquad\quad p_{car\_i} \bullet o_{garageAndTowTruckNotification}?\langle x_{garageAndTowTruckResponse}, x_{garageInfo}, x_{towTruckInfo}, id_i\rangle. \\ \qquad\qquad p_{assistance} \bullet o_{garageAndTowTruckNotificationAck}!\langle id_i\rangle \\ \qquad \mid [x_{rentalCarResponse}, x_{rentalCarInfo}] \\ \qquad\quad p_{car\_i} \bullet o_{rentCarNotification}?\langle x_{rentalCarResponse}, x_{rentalCarInfo}, id_i\rangle. \\ \qquad\qquad (\,p_{assistance} \bullet o_{rentCarNotificationAck}!\langle id_i\rangle \mid p_{car\_i} \bullet o_{rentalCarRedirected}?\langle id_i\rangle\,)\,) \end{array}$$

Basically, the recovery service contacts $GpsSystem_i$, to get the car's location, invokes the *Assistance* service, by providing its partner name p_{car_i}, a correlation identifier id_i, the car's location (stored in

$x_{carLocation}$), the diagnostic data (stored in $x_{diagnosticData}$) and the driver's credit card number $ccNum_i$, and, finally, waits for the messages notifying the positive or negative outcomes of the request processing. For the sake of presentation, we relegate the specification of the remaining components of the in-vehicle platform, i.e. *SensorsMonitor*$_i$ and *GpsSystem*$_i$, to Fantechi et al. [2010].

The COWS term *Assistance*, modelling the assistance service, is defined as follows:

$$
\begin{array}{l}
*\ [x_{car}, x_{id}, x_{carLocation}, x_{diagnosticData}, x_{ccNum}] \\
\ p_{assistance} \cdot o_{severeFailure}?\langle x_{car}, x_{id}, x_{carLocation}, x_{diagnosticData}, x_{ccNum}\rangle. \\
\quad (\, p_{bank} \cdot o_{charge}!\langle p_{assistance}, x_{ccNum}, amount, x_{id}\rangle \\
\quad\ \mid\ p_{assistance} \cdot o_{chargeFail}?\langle x_{id}\rangle. \\
\qquad\qquad x_{car} \cdot o_{failureNotification}!\langle noFunds, x_{id}\rangle \\
\qquad\ +\ p_{assistance} \cdot o_{chargeOK}?\langle x_{id}\rangle. \\
\qquad\qquad (\, Ordering\ \mid\ p_{assistance} \cdot o_{undo}?\langle cardCharge, x_{id}\rangle.\, p_{bank} \cdot o_{revoke}!\langle x_{id}\rangle\,)\,)
\end{array}
$$

The replication operator is used here for specifying that the assistance service is persistent, i.e. it is capable of creating multiple instances to serve several requests simultaneously. Once instantiated, the service contacts the service *Bank* to charge the driver's credit card with a security amount. Whenever charging the credit card fails, the service sends a notification message to the invoking car and terminates. Otherwise, it installs a compensation handler that takes care of revoking the credit card charge, and proceeds with the ordering phase. Activation of the compensation activity requires a signal *cardCharge* (i.e. an internal message identified by x_{id} to guarantee uniqueness) along $p_{assistance} \cdot o_{undo}$ and, as we will see soon, takes place whenever both garage and car rental orders fail.

Ordering tries to order three suitable on road services (i.e. *Garage*$_i$, *TowTruck*$_j$ and *RentalCar*$_k$), by first contacting the garage and, then, the car rental and (possibly) the tow truck. It is defined as follows:

$$
\begin{array}{l}
[x_{garageLocation}, x_{garageInfo}] \\
(\, p_{garage\_i} \cdot o_{orderGarage}!\langle p_{assistance}, x_{diagnosticData}, x_{id}\rangle \\
\ \mid\ p_{assistance} \cdot o_{orderGarageFail}?\langle x_{garageInfo}, x_{id}\rangle. \\
\qquad (\, x_{car} \cdot o_{garageAndTowTruckNotification}!\langle fail, x_{garageInfo}, null, x_{id}\rangle \\
\qquad\ \mid p_{assistance} \cdot o_{garageAndTowTruckNotificationAck}?\langle x_{id}\rangle. \\
\qquad\qquad (\, RentalCarOrdering_{atStrandedCar} \\
\qquad\qquad\ \mid p_{assistance} \cdot o_{orderRentCarFail}?\langle x_{id}\rangle.\ p_{assistance} \cdot o_{undo}!\langle cardCharge, x_{id}\rangle\,)\,) \\
\quad + p_{assistance} \cdot o_{orderGarageOK}?\langle x_{garageLocation}, x_{garageInfo}, x_{id}\rangle. \\
\qquad (\, p_{assistance} \cdot o_{undo}?\langle orderGarage, x_{id}\rangle. \\
\qquad\qquad (\, p_{garage\_i} \cdot o_{cancel}!\langle x_{id}\rangle \mid p_{assistance} \cdot o_{undo}!\langle rentCar, x_{id}\rangle\,) \\
\qquad\ \mid TowTruckOrdering \mid RentalCarOrdering_{atGarage} \\
\qquad\ \mid p_{assistance} \cdot o_{end}?\langle x_{id}\rangle.\ p_{assistance} \cdot o_{end}?\langle x_{id}\rangle. \\
\qquad\qquad p_{assistance} \cdot o_{orderTowTruckFail}?\langle x_{id}\rangle. \\
\qquad\qquad\quad (\, p_{assistance} \cdot o_{undo}!\langle orderGarage, x_{id}\rangle \\
\qquad\qquad\quad\ \mid p_{assistance} \cdot o_{orderRentCarFail}?\langle x_{id}\rangle. \\
\qquad\qquad\qquad p_{assistance} \cdot o_{undo}!\langle cardCharge, x_{id}\rangle\,)\,)\,)
\end{array}
$$

If ordering the garage fails, the rental car service is invoked by indicating that the rental car has to be handed over to the driver at the stranded vehicle location (term *RentalCarOrdering*$_{atStrandedCar}$). Whenever also ordering the rental car fails (indicated by the receipt of a signal along the endpoint $p_{assistance} \cdot o_{orderRentCarFail}$), the compensation of the credit card charge is invoked by sending a signal *cardCharge* along the endpoint $p_{assistance} \cdot o_{undo}$. Otherwise, in case of success of the garage ordering, the tow truck and rental car (to be handed over at the garage location, term *RentalCarOrdering*$_{atGarage}$) orderings concurrently start. Moreover, a compensation handler is installed; it will be activated whenever tow truck ordering fails and, in that case, attempts to cancel the garage order (by invoking operation o_{cancel}) and to compensate the rental car order (by sending a signal *rentCar* along the endpoint $p_{assistance} \cdot o_{undo}$). Finally, when both ordering activities terminate (the endpoint $p_{assistance} \bullet o_{end}$ is used to appropriately synchronise their terminations), if ordering the tow truck failed, the compensation of the garage order is invoked, and whenever also ordering the rental car failed, also the compensation of the credit card charge is activated.

TowTruckOrdering is simply defined as follows:

$$
\begin{array}{l}
[x_{towTruckInfo}] \\
(\, p_{towTruck\_j} \bullet o_{orderTowTruck} !\langle p_{assistance}, x_{carLocation}, x_{garageLocation}, x_{id} \rangle \\
\quad |\; p_{assistance} \bullet o_{orderTowTruckFail} ?\langle x_{towTruckInfo}, x_{id} \rangle . \\
\qquad\qquad (\, x_{car} \bullet o_{garageAndTowTruckNotification} !\langle fail, x_{garageInfo}, x_{towTruckInfo}, x_{id} \rangle \\
\qquad\qquad\; |\; p_{assistance} \bullet o_{garageAndTowTruckNotificationAck} ?\langle x_{id} \rangle . \\
\qquad\qquad\qquad (\, p_{assistance} \bullet o_{orderTowTruckFail} !\langle x_{id} \rangle \;|\; p_{assistance} \bullet o_{end} !\langle x_{id} \rangle \,)\,) \\
\quad\; + p_{assistance} \bullet o_{orderTowTruckOK} ?\langle x_{towTruckInfo}, x_{id} \rangle . \\
\qquad\qquad (\, x_{car} \bullet o_{garageAndTowTruckNotification} !\langle ok, x_{garageInfo}, x_{towTruckInfo}, x_{id} \rangle \\
\qquad\qquad\; |\; p_{assistance} \bullet o_{garageAndTowTruckNotificationAck} ?\langle x_{id} \rangle . \; p_{assistance} \bullet o_{end} !\langle x_{id} \rangle \,)\,)
\end{array}
$$

while $RentalCarOrdering_{atGarage}$ is:

$$
\begin{array}{l}
[x_{rentalCarInfo}] \\
(\, p_{rentalCar\_k} \bullet o_{rentCar} !\langle p_{assistance}, x_{garageLocation}, x_{id} \rangle \\
\quad |\; p_{assistance} \bullet o_{rentCarFail} ?\langle x_{rentalCarInfo}, x_{id} \rangle . \\
\qquad\qquad (\, x_{car} \bullet o_{rentCarNotification} !\langle fail, x_{rentalCarInfo}, x_{id} \rangle \\
\qquad\qquad\; |\; p_{assistance} \bullet o_{rentCarNotificationAck} ?\langle x_{id} \rangle . \\
\qquad\qquad\qquad (\, p_{assistance} \bullet o_{orderRentCarFail} !\langle x_{id} \rangle \;|\; p_{assistance} \bullet o_{end} !\langle x_{id} \rangle \,)\,) \\
\quad\; + p_{assistance} \bullet o_{rentCarOK} ?\langle x_{rentalCarInfo}, x_{id} \rangle . \\
\qquad\qquad (\, p_{assistance} \bullet o_{undo} ?\langle rentCar, x_{id} \rangle . \\
\qquad\qquad\qquad (\, p_{rentalCar\_k} \bullet o_{redirect} !\langle x_{carLocation}, x_{id} \rangle \;|\; x_{car} \bullet o_{rentalCarRedirected} !\langle x_{id} \rangle \,) \\
\qquad\qquad\; |\; x_{car} \bullet o_{rentCarNotification} !\langle ok, x_{rentalCarInfo}, x_{id} \rangle \\
\qquad\qquad\; |\; p_{assistance} \bullet o_{rentCarNotificationAck} ?\langle x_{id} \rangle . \; p_{assistance} \bullet o_{end} !\langle x_{id} \rangle \,)\,)
\end{array}
$$

The term $RentalCarOrdering_{atStrandedCar}$ is defined similarly; the major difference is that the rental car is requested to be handed over at the stranded vehicle location rather than at the garage one.

OnRoadRepairServices results from the composition of various on road services and is

$$Garage_1 \;|\; Garage_2 \;|\; TowTruck_1 \;|\; TowTruck_2 \;|\; RentalCar_1 \;|\; RentalCar_2$$

Garage services can be defined as follows

$$
\begin{array}{rl}
Garage_i \;\triangleq & * \; [x_{cust}, x_{data}, x_{id}] \\
& \quad p_{garage\_i} \bullet o_{orderGarage} ?\langle x_{cust}, x_{data}, x_{id} \rangle . \\
& \qquad (\, p_{garage\_i} \bullet o_{checkOK} !\langle x_{id} \rangle \;|\; p_{garage\_i} \bullet o_{checkFail} !\langle x_{id} \rangle \\
& \qquad\; |\; p_{garage\_i} \bullet o_{checkFail} ?\langle x_{id} \rangle . \, x_{cust} \bullet o_{orderGarageFail} !\langle garageFailureInfo_i, x_{id} \rangle \\
& \qquad\quad + p_{garage\_i} \bullet o_{checkOK} ?\langle x_{id} \rangle . \\
& \qquad\qquad\qquad [k]\,(\, x_{cust} \bullet o_{orderGarageOK} !\langle garageLocation_i, garageInfo_i, x_{id} \rangle \\
& \qquad\qquad\qquad\quad |\; p_{garage\_i} \bullet o_{cancel} ?\langle x_{id} \rangle . \, \mathbf{kill}(k) \,)\,)
\end{array}
$$

where, for simplicity, success or failure of garage orders is modelled by means of a non-deterministic choice. The other on road services can be modelled in a similar way.

The complete specification, written in the 'machine readable' syntax accepted by CMC and including the services not explicitly shown in this section, can be found in Fantechi et al. [2010]. The corresponding LTS, analysed in the next section, has about 50000 states.

7. ANALYSIS OF THE CASE STUDY

In this section, we demonstrate feasibility and effectiveness of our methodology by using it to analyse the automotive scenario specified in the previous section. Firstly, we verify over the COWS specification of the scenario the abstract properties of services introduced in Section 1 and formalized as SocL formulae in Section 2. Then, by applying a different set of abstraction rules, we target

Table III. Assistance service verification results

Property	Validity	States	Execution Time
(1) Available	true	48627	4m 41s
(2) Parallel	true	48627	4m 39s
(3) Sequential	false	11	0m 00.05s
(4) One-shot	false	11	0m 00.05s
(5) Off-Line	false	9860	0m 03s
(6) Cancelable	false	11	0m 00.05s
(7) Revocable	false	48627	4m 39s
(8) Responsive	true	48627	4m 39s
(9) Single-response	false	441	0m 00.2s
(10) Multiple-response	false	6829	0m 32s
(11) No-response	false	77	0m 00.05s
(12) Reliable	false	5911	0m 26s
All formulae (1)..(12) in a single session	-	48627	4m 51s

our analysis to specific services of the scenario (e.g. the bank service). Finally, by changing again the abstraction rules, we focus on low-level service behaviours, such as compensation handling.

7.1. Analysis of service properties

We start the analysis of the automotive scenario by verifying if its main service, i.e. the *Assistance* service, enjoys the abstract properties expressed as SocL formulae in Section 2.3. To this aim, we focus our observations on the assistance service by providing to CMC the following abstraction rules:

$$\begin{array}{lll}
Action & p_{assistance} \bullet o_{severeFailure}, \langle *, \$id, *, *, * \rangle & \rightarrow request(road\_assistance, \$id) \\
Action & * \bullet o_{garageAndTowTruckNotification}, \langle ok, *, *, \$id \rangle & \rightarrow responseOk(road\_assistance, \$id) \\
Action & * \bullet o_{rentCarNotification}, \langle ok, *, \$id \rangle & \rightarrow responseOk(road\_assistance, \$id) \\
Action & * \bullet o_{failureNotification}, \langle *, \$id \rangle & \rightarrow responseFail(road\_assistance, \$id) \\
Action & * \bullet o_{garageAndTowTruckNotification}, \langle fail, *, *, \$id \rangle & \rightarrow responseFail(road\_assistance, \$id) \\
Action & * \bullet o_{rentCarNotification}, \langle fail, *, \$id \rangle & \rightarrow responseFail(road\_assistance, \$id) \\
State & p_{assistance} \bullet o_{severeFailure}? & \rightarrow accepting\_request(road\_assistance)
\end{array}$$

According to this abstraction, the service accepts a request for the interaction *road_assistance* when it receives an assistance request message from a car computer system, and replies with a positive response when an order (of garage/tow truck or of rental car) succeeds. Indeed, in this case, the driver may continue its journey. If during the request processing some operation fails (e.g. the bank does not accept the request of charging the driver's credit card), the service replies with a negative response. More specifically, in case of complete success, two actions $responseOk(road\_assistance, \$car)$ are performed and no action $responseFail(road\_assistance, \$car)$ is observed, while, in case of complete failure, some actions $responseFail(road\_assistance, \$car)$ are performed and no action $responseOk(road\_assistance, \$car)$ is observed. By means of the last abstraction rule, each state that can accept requests for interaction *road_assistance* is labelled by the atomic proposition $accepting\_request(road\_assistance)$.

Now, by using CMC, we automatically check the SocL formulae (1)..(12) of Section 2.3 over the obtained abstract L^{2}TS; to do this, we instantiate those generic formulae over the recovery service by simply replacing any occurrence of the generic interaction name *i* with *road_assistance*. The results of the verification of these properties with CMC are summarized in Table III, where we also report the number of states considered during the evaluation and the execution time taken by CMC for the evaluation of a formula (as a single formula per CMC session). Time measures appearing in Tables III-VI have been collected using the command line version of CMC on an Apple MacMini computer (2 GHz Intel Core 2 Duo and 4 GB of memory).

The results show that the assistance service is available (indeed several instances of the service can be created at any moment) and, hence, parallel, non sequential and non one-shot. Since there exist some execution paths that lead to a complete success, i.e. no negative responses are provided, the service is not off-line. Moreover, since it does not permit cancelling requests and does not accept undo calls (i.e. the atomic propositions $accepting\_cancel(road\_assistance, var)$ and $accepting\_undo(road\_assistance, var)$ do not hold in any state), it is trivially non cancelable and non revocable. The service also exhibits the desired characteristic to be responsive, i.e. it provides at least a response, but it is not reliable because it may produce no positive responses to a request. Finally, the fact that the service is neither single-response nor multiple-response means that in some cases it provides two responses (e.g. in case of complete success) and in other cases it provides only one response (e.g. when the credit card charge is denied by the bank).

Due to the on-the-fly nature of the model checker, the verification times depend directly on the number of generated states: for the presented case study, such times range from a fraction of second to the order of few minutes. Actually, most of the execution time (about 97%) is being spent by the activity of generating the abstract L^2TS of the model, and only a fraction of the execution time is directly related to the complexity of the evaluation of the formula. This fact is clearly put in evidence when all 12 formulae are evaluated inside the same CMC session. In this case, the L^2TS is generated once and used several times for the evaluation of the various formulae. The result of this experiment is shown in the last line of Table III; we can observe that only 4 minutes and 51 seconds are needed to evaluate all 12 formulae, versus the 4 minutes and 41 seconds needed to evaluate just the first formula.

An important role of model checkers is the ability to provide a small fragment of the system state space as a witness or as a counterexample for a checked property. During the design phase, when it is likely that the specification still contains errors and precise guidance in identifying them is essential, the possibility of using the model checker in a way similar to a debugger turns out to be very useful. In particular, while at validation time a “success” result from the model checker would be a sufficient response, during development, in case of a “failure” result, the user usually needs not only a counterexample in term of abstract occurred actions (the labels of our L^2TS) but also a precise view of how the COWS term actually evolved, with an indication of the concrete actions which were executed at each computational step of the counterexample.

Once a SocL formula has been checked, CMC can provide, if requested, all the details related to the specific witness/counterexample which explains in depth the result of the evaluation of the formula. In Figure 14 we show the counterexample generated by CMC for the formula checking the One-shot property for the assistance service. This formula asserts that in each state reached after accepting a $road\_assistance$ request the service is not able to accept further requests. The counterexample indeed shows how the evaluation of the formula proceeds along the system path C1 $\longrightarrow$ C2 $\longrightarrow$ C4 $\longrightarrow$ C6 $\longrightarrow$ C8, which is a full path, along which a request is accepted (transition C6 $\longrightarrow$ C8), leading to a state (C8) where a further request can be indeed accepted (proposition $accepting\_request(road\_assistance)$ holds). At each evolution step both the abstract and concrete actions are reported.

We can also investigate in more detail the request-response relation for the assistance service. To do this, firstly we define a different abstraction of the scenario by providing CMC with the following rules:

$$\begin{array}{ll}
Action & p_{assistance} \bullet o_{severeFailure}, \langle *, \$id, *, *, * \rangle \\
& \quad \rightarrow request(road\_assistance, \$id) \\
Action & * \bullet o_{garageAndTowTruckNotification}, \langle ok, *, *, \$id \rangle \\
& \quad \rightarrow responseOk(road\_assistance, \$id, truckGarage) \\
Action & * \bullet o_{rentCarNotification}, \langle ok, *, \$id \rangle \\
& \quad \rightarrow responseOk(road\_assistance, \$id, rentalCar) \\
Action & * \bullet o_{failureNotification}, \langle *, \$id \rangle \\
& \quad \rightarrow responseFail(road\_assistance, \$id, truckGarage)
\end{array}$$

```
------------------------------------------------------------------------------------------------
 The formula:
  AG [ request(road_assistance,$var) ] AG  not accepting_request(road_assistance)
 is FOUND_FALSE in State C1
 This happens because:
  C1  -->  C2 /* car1.engineFailure!<diagnosticData1>,car1.engineFailure?<DiagnosticData> */
  C2  -->  C4 /* gpsCar1.reqLocation!<id1>,gpsCar1.reqLocation?<Id> */
  C4  -->  C6 /* car1.respLocation!<id1, carLocation1>,car1.respLocation?<id1,CarLocation> */
 and the formula:
  [ request(road_assistance,$var) ] AG  not accepting_request(road_assistance)
 is FOUND_FALSE in State C6

 This happens because:
  C6  -->  C8 { request(road_assistance,id1) }
      /* assistance.severeFailure!<car1,id1,carLocation1,diagnosticData1,ccNum1>,
         assistance.severeFailure?<Car,Id,CarLocation,DiagnosticData,CcNum> */
 and the formula:
  AG  not accepting_request(road_assistance)
   is FOUND_FALSE in State C8
 because the formula:
  not accepting_request(road_assistance)
 is FOUND_FALSE in State C8
 because  the formula:
  accepting_request(road_assistance)
 is FOUND_TRUE in State C8
------------------------------------------------------------------------------------------------
```

Fig. 14. Counterexample for the One-shot property of the assistance service.

$$\begin{aligned}
&Action \ \ * \bullet o_{failureNotification}, \langle *, \$id\rangle \\
&\qquad\qquad \rightarrow responseFail(road\_assistance, \$id, rentalCar) \\
&Action \ \ * \bullet o_{garageAndTowTruckNotification}, \langle fail, *, *, \$id\rangle \\
&\qquad\qquad \rightarrow responseFail(road\_assistance, \$id, truckGarage) \\
&Action \ \ * \bullet o_{rentCarNotification}, \langle fail, *, \$id\rangle \\
&\qquad\qquad \rightarrow responseFail(road\_assistance, \$id, rentalCar)
\end{aligned}$$

The obtained L^{2}TS differs from that previously introduced for the presence of an additional argument (that can be either the value *truckGarage* or the value *rentalCar*, and indicates which order succeeds or fails) in the correlation tuple of the response actions. Now, we can verify for example if, once requested, the service always provides at least one response about the status of the garage/tow truck ordering and at least one response about the status of the car renting:

$$\text{(F1)} \quad AG\ [request(road\_assistance, \underline{var})] \\ AF_{responseOk(road\_assistance,var,truckGarage)\ \vee\ responseFail(road\_assistance,var,truckGarage)}\ true$$

$$\text{(F2)} \quad AG\ [request(road\_assistance, \underline{var})] \\ AF_{responseOk(road\_assistance,var,rentalCar)\ \vee\ responseFail(road\_assistance,var,rentalCar)}\ true$$

Similarly, we can verify that a positive response is never followed by a negative one (and vice versa) for the same order:

$$\text{(F3)} \quad AG\ [responseOk(road\_assistance, \underline{var}, \underline{order})] \neg\ EF_{responseFail(road\_assistance,var,order)}\ true$$

$$\text{(F4)} \quad AG\ [responseFail(road\_assistance, \underline{var}, \underline{order})] \neg\ EF_{responseOk(road\_assistance,var,order)}\ true$$

All these last four properties are indeed satisfied by the second abstraction of the service, as shown in Table IV.

Table IV. Assistance service (detailed view) verification results

Property	Validity	States	Execution Time
(F1) Responsive about garage/tow truck order status	true	48627	4m 39s
(F2) Responsive about rental car order status	true	48627	4m 41s
(F3) A positive response is never followed by a negative one	true	48627	4m 36s
(F4) A negative response is never followed by a positive one	true	48627	4m 41s
All formulae (F1)..(F5) in a single session	-	48627	4m 58s

7.2. Analysis of other services of the automotive scenario

By changing again the abstraction rules applied to the concrete L^{2}TS modelling the automotive scenario, we can verify the abstract properties of services introduced in Section 1 (and possibly some specific variants of them) also over other services appearing in the scenario. For example, we consider here *GpsSystem$_1$*, *Bank* and *RentalCar$_1$*, and apply the following rules:

$$\begin{array}{rlcl}
Action & p_{gps\_1} \bullet o_{reqLocation}, \langle \$id \rangle & \rightarrow & request(gps1, \$id) \\
Action & * \bullet o_{respLocation}, \langle \$id, * \rangle & \rightarrow & responseOk(gps1, \$id) \\
State & p_{gps\_1} \bullet o_{reqLocation}? & \rightarrow & accepting\_request(gps1) \\[1ex]
Action & p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle & \rightarrow & request(charge, \$id) \\
Action & * \bullet o_{chargeOK}, \langle \$id \rangle & \rightarrow & responseOk(charge, \$id) \\
Action & * \bullet o_{chargeFail}, \langle \$id \rangle & \rightarrow & responseFail(charge, \$id) \\
Action & p_{bank} \bullet o_{revoke}, \langle \$id \rangle & \rightarrow & undo(charge, \$id) \\
State & p_{bank} \bullet o_{charge}? & \rightarrow & accepting\_request(charge) \\
State & p_{bank} \bullet o_{revoke}?\langle \$id \rangle & \rightarrow & accepting\_undo(charge, \$id) \\[1ex]
Action & p_{rentalCar\_1} \bullet o_{rentCar}, \langle *, *, \$id \rangle & \rightarrow & request(rental\_car1, \$id) \\
Action & * \bullet o_{rentCarOK}, \langle *, \$id \rangle & \rightarrow & responseOk(rental\_car1, \$id) \\
Action & * \bullet o_{rentCarFail}, \langle *, \$id \rangle & \rightarrow & responseFail(rental\_car1, \$id) \\
State & p_{rentalCar\_1} \bullet o_{rentCar}? & \rightarrow & accepting\_request(rental\_car1)
\end{array}$$

For the obtained abstraction of the case study, by using CMC, we verify that the service *GpsSystem$_1$* is available and reliable (properties (F5) and (F6) in Table V, which are expressed as instantiations of the generic formulae (1) and (12) of Section 2.3 where i has been replaced by the interaction name $gps1$). Similarly, we check that services *Bank* and *RentalCar$_1$* are available (properties (F7) and (F10), instantiations of formula (1) with i replaced by $charge$ and $rental\_car1$, respectively), and that *Bank* is strong revocable, i.e. after a successful response to a credit card charge request, the bank accepts undo requests for the successfully completed transaction (property (F9), instantiation of the strong variant of formula (7) with i replaced by $charge$). Finally, we prove that *Bank* and *RentalCar$_1$* satisfy the following formulae stating that, after the services have accepted a request, they always provide a *single* (either positive or negative) response:

(F8) $AG\ [request(charge, \underline{id})]$
$$AF_{responseOk(charge,id)\ \vee\ responseFail(charge,id)}\ \neg\ EF_{responseOk(charge,id)\ \vee\ responseFail(charge,id)}\ true$$

(F11) $AG\ [request(rental\_car1, \underline{customer})]$
$$AF_{responseOk(rental\_car1,customer)\ \vee\ responseFail(rental\_car1,customer)}$$
$$\neg\ EF_{responseOk(rental\_car1,customer)\ \vee\ responseFail(rental\_car1,customer)}\ true$$

The results of the verification of these properties with CMC are summarized in Table V.

7.3. Orchestration and compensation properties

The properties we have introduced and checked so far imply a sort of *black-box* view of the individual services. In fact, their statements are general and given in terms of the externally observable behaviour of services. Of course, whenever details on the internal architecture of a given service,

Table V. *GpsSystem*, *Bank* and *RentalCar₁* verification results

Property	Validity	States	Execution Time
(F5) *GpsSystem* is always available	true	48627	4m 42s
(F6) *GpsSystem* always replies with successful responses	true	48627	4m 39s
(F7) *Bank* is always available	true	48627	4m 41s
(F8) *Bank* is 'single responsive'	true	48627	4m 42s
(F9) *Bank* is a strong revocable service	true	48627	4m 44s
(F10) *RentalCar₁* is always available	true	48627	4m 41s
(F11) *RentalCar₁* is 'single responsive'	true	48627	4m 42s
All formulae (F5)..(F11) in a single session	-	48627	5m 21s

in terms of its subcomponents, are known, i.e. when the service is a sort of *white-box*, further behavioural properties can be stated in terms of the behaviours of these subcomponents. In general, these properties can express desirable orchestration or compensation behaviours. In the following, we show some examples of formalization of this kind of properties in SocL in the context of our automotive scenario.

In this analysis of the case study the abstraction is obtained by applying the following rules:

$$\begin{array}{lll}
Action & \$car \bullet o_{engineFailure} & \rightarrow request(road\_assistance, \$car) \\
Action & \$car \bullet o_{towTruckOK} & \rightarrow responseOk(road\_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarOK} & \rightarrow responseOk(road\_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow responseFail(road\_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{chargeFail} & \rightarrow responseFail(road\_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{notFound} & \rightarrow responseFail(road\_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{notFound} & \rightarrow responseFail(road\_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{garageFail} & \rightarrow responseFail(road\_assistance, \$car, truckGarage) \\
Action & \$car \bullet o_{rentalCarFail} & \rightarrow responseFail(road\_assistance, \$car, rentalCar) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow responseFail(road\_assistance, \$car, truckGarage) \\
 & & \\
Action & p_{bank} \bullet o_{charge}, \langle *, *, *, \$id \rangle & \rightarrow request(charge, \$id) \\
Action & * \bullet o_{chargeOK}, \langle \$id \rangle & \rightarrow responseOk(charge, \$id) \\
Action & * \bullet o_{chargeFail}, \langle \$id \rangle & \rightarrow responseFail(charge, \$id) \\
Action & p_{bank} \bullet o_{revoke}, \langle \$id \rangle & \rightarrow undo(charge, \$id) \\
 & & \\
Action & \$car \bullet o_{garageOK} & \rightarrow responseOk(garage, \$car) \\
Action & * \bullet o_{cancel}, \langle \$car \rangle & \rightarrow undo(garage, \$car) \\
Action & \$car \bullet o_{towTruckFail} & \rightarrow responseFail(towtruck, \$car)
\end{array}$$

The above set of rules is obtained by putting together some of the rules previously introduced with some new rules for capturing interactions with garage, tow truck and rental car services.

Now, we can check the following properties for the automotive scenario.

— After a successful credit card charge, the rental car will be booked, or the garage and tow truck will be ordered, or the credit chard charge will be revoked.

$$\text{(F12)} \quad AG\ [responseOk(charge, \underline{id})]\ AF_{responseOk(road\_assistance,id,rentalCar) \vee responseOk(road\_assistance,id,truckGarage) \vee undo(charge,id)}\ true$$

— It cannot happen that, after the driver's credit card has been charged and some service ordered, the credit card charge is revoked.

$$\text{(F13)} \quad \neg\ EF_{responseOk(charge,\underline{id})}\ EF_{responseOk(road\_assistance,id,rentalCar) \vee responseOk(road\_assistance,id,truckGarage)}\ EF_{undo(charge,id)}\ true$$

Table VI. Orchestration and compensation properties verification results

Property	Validity	States	Execution Time
(F12) After a successful credit card charge, the rental car will be booked, or the garage and tow truck will be ordered, or the credit chard charge will be revoked	true	48627	4m 38s
(F13) It cannot happen that, after the driver's credit card has been charged and some service ordered, the credit card charge is revoked	true	48627	4m 45s
(F14) It cannot happen that, after the credit card has been charged and then revoked, some order succeeds	true	48627	4m 43s
(F15) After the garage has been booked, if the tow truck service is not available then the garage is revoked	true	48627	4m 44s
All formulae (F12)..(F15) in a single session	-	48627	5m 08s

— It cannot happen that, after the credit card has been charged and then revoked, some order succeeds.

$$\text{(F14)} \quad \neg\, EF_{responseOk(charge,\underline{id})}\; EF_{undo(charge,id)}\; EF_{responseOk(road\_assistance,id,rentalCar)\,\vee\, responseOk(road\_assistance,id,truckGarage)}\; true$$

— After the garage has been booked, if the tow truck service is not available then the garage is revoked.

$$\text{(F15)} \quad AG\,[responseOk(garage,\underline{var})]\; AG\,(\,[responseFail(towtruck,var)]\; AF_{undo(garage,var)}\; true\,)$$

The results of this verification are summarized in Table VI.

8. FINAL REMARKS AND RELATED WORK

In this section, we review and compare related work, and point out several distinctive aspects of our approach to the verification of SOC systems.

We use a class of *rich* L^{2}TSs as semantic model of the behavior of SOC systems. Our L^{2}TSs have both labelled transitions, typical of LTSs, and labelled states, as in Kripke structures. Usefulness of such a kind of models has been by now recognized in many contexts. For example, P/T systems [Kindler and Vesper 1998] have been introduced for Petri Nets modelling, State-Event LTSs [Lawford et al. 1996] for real time modeling, Action-and-State Labelled Markov Chains [Baier et al. 2004] for stochastic modeling, Multilabelled Transition Systems [De Nicola and Loreti 2007] for nominal calculi, Labelled Kripke Structures [Müller-Olm et al. 1999; Chaki et al. 2004; Chaki et al. 2005] for software verification. However, almost all the efforts towards the formal verification of SOC designs through model checking (a wide survey of the approaches based on WS-BPEL is presented by van Breugel and Koshkina [2006]), rely on either LTSs or Kripke structures. In this way, a direct representation of the state/event duality which naturally belongs to SOC systems is lost. Moreover, our L^{2}TSs are rich because their states are labelled by sets of structured predicates and their transitions are labelled by sets of structured events. Both events and predicates have the form of tuples of data values (static labels, integers, boolean, strings, names), which can either result from the evaluation of statically encoded literals or from the dynamic evaluation of the result of a computation (as e.g. the dynamic creation of a unique new name).

Our methodology shows a novel use of temporal logics and model checkers. The properties to be checked are expressed in the logic SocL in a way that can be independent from the model of the system under analysis; then, through an abstraction process, the model is tailored to be checked against the properties of interest. In fact, the approach usually followed in concurrency theory (see, e.g., Moller and Stevens [1999], Victor and Moller [1994], Bouali et al. [1994]) is quite different: typically the properties are formalized in the logic after the system under consideration has been specified and in terms of the actions occurring in it, which implicitly requires the effort of identifying the ‘intended meaning’ of the actions occurring in the specification. Instead, we make this effort explicit by means of the definition of appropriate abstraction rules. This has some important advantages that make our approach largely applicable. On the one hand, the formulae predicating, e.g., availability or responsiveness of a service are the same irrespective of the domain of the service, which may be a bank, a booking service, a shipping service or any other kind of service. On the other hand, they are also independent of the precise syntax of actions in the service specification.

Another point in favour 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 L^2TS-based operational semantics for the language of interest and then a suitable set of abstraction rules mapping the concrete actions of the language into the abstract actions of SocL. Therefore, the choice made in the presented case study, of starting from a UML4SOA description of the architecture and later proceeding with a COWS formalization, is not a key of our approach, since after the abstraction step the origin of the concrete model becomes rather irrelevant. For example, ter Beek et al. [2008] have formalized a similar automotive case study as a set of UML state machines but, after an appropriate abstraction step, that formalization could undergo the same verifications shown here.

We have chosen to tailor our methodology to COWS, rather than to a different service specification language, for two main reasons. As regards higher level specification languages, equipping them directly with a semantics in terms of L^2TSs would be a very challenging task. Instead, we could apply our methodology also to some of them, e.g. WS-BPEL, SRML [Fiadeiro et al. 2006] and UML4SOA [Mayer et al. 2008], by profitably exploiting their encodings into COWS [Lapadula et al. 2007; Bocchi et al. 2009; Banti et al. 2010]. As regards the many other process calculi for SOC that have been proposed in the literature (see e.g. [Carbone et al. 2007; Lanese et al. 2007; Boreale et al. 2008; Bruni et al. 2008; Vieira et al. 2008; Guidi et al. 2006; Busi et al. 2006]), our favour towards COWS, besides the existence of the above mentioned encodings, is motivated by its mechanisms and primitives that have proven to be particularly expressive for modelling the behaviour of service-oriented applications (see e.g. the scenarios modelled by Lapadula et al. [2008]). In fact, kill activities are effective for representing ordinary and exceptional process terminations, while protection permits to naturally represent exception and compensation handlers that are supposed to run after normal computations terminate. Even more crucially, the correlation mechanism permits to automatically correlate messages belonging to the same interaction, hence avoiding to mix up messages from different service instances.

Other innovative aspects of our approach are related to SocL. To take advantage of the richness of the underlying semantic model, that is L^2TS, SocL integrates the action paradigm with propositions that are true over states. This facilitates the task of formalizing properties of service-oriented systems that in pure action-based or pure state-based logics can be quite cumbersome to write down, since it is often necessary to specify both state information and evolution in time by actions. SocL has many commonalities with UCTL [ter Beek et al. 2008], a logic recently designed to express properties of UML statecharts. In fact, they share the same logical operators, combine the action paradigm with predicates that are true over states, and are both interpreted on L^2TSs by exploiting the same on-the-fly model checking engine. The main difference with UCTL is that SocL formulae are parameterized by data values, which make them suitable for effectively expressing service properties that are based on correlation data. As we mentioned in the Introduction, in the loosely coupled context of SOC systems, correlation is emerging as a powerful mechanism for linking together actions executed by a component as part of the same interaction. This simple concept has many instantiations. For example, in WS-Addressing [Gudgin et al. 2006] correlation data are implicitly dealt with (by the underlying communication protocols) thus resulting in a less flexible mechanism with respect to that provided by WS-BPEL, where instead correlation data must be explicitly dealt with by the developer and must be included among the data used for invoking service operations. These different levels of abstraction are somehow reflected by the process calculi for SOC proposed so far which may be roughly classified in session-based, like those proposed by Carbone et al. [2007], Lanese et al.

[2007], Boreale et al. [2008], Bruni et al. [2008], Vieira et al. [2008], and correlation-based, like those proposed by Lapadula et al. [2006], Guidi et al. [2006], Busi et al. [2006], Lapadula et al. [2007], respectively.

Other relevant proposals of action- and state-based logics are SE-LTL [Chaki et al. 2004] and SE-AΩ [Chaki et al. 2005]. The former logic is basically an extension of LTL that permits referring both to states and events of Labelled Kripke Structures. Thus, it mainly differs from SocL because it is linear-time and is not structured in terms of state, path and action formulae. The authors show that the problem of verifying SE-LTL formulae over Labelled Kripke Structures can be reduced to that of verifying LTL formulae over Kripke Structures. Instead, SE-AΩ is a universal branching-time temporal logic for which an efficient model checking algorithm directly handling both states and events has been provided. Differently from SocL, negation can be only applied to atomic propositions, and the set of logic operators is not fixed in advance, since any ω-regular language can serve as a (universally quantified) temporal operator. Above all, both SE-LTL and SE-AΩ do not permit specifying parametric formulae.

Actually, most of the technical complexity of our logic comes from its parametricity, which however has proved to be fundamental to capture correlation data during the evaluation of formulae and, hence, to link together actions belonging to the same interaction. Anyway, logics with parametric formulae have already been proposed in the literature. Many logics for expressing properties of value-passing processes (e.g. Lin [1993], Hennessy and Liu [1995], Dam [1996; 2003], Ferrari et al. [2003], Yang et al. [2004], Tiu [2005], Mateescu and Thivolle [2008]) permit the instantiation of subformulae with dynamically generated values. Most of them deal with systems capable of communicating with their execution context and use parametric formulae that can only express properties of such communications, i.e. they cannot state properties depending on the messages exchanged in communications between components of the system under analysis. Moreover, communication does not exploit pattern-matching which is instead quite useful to check actions correlation in the service-oriented setting. Differently, in the use of the logic SocL typical of the verifications made with CMC, properties of internal communications can be stated and verified since the labels of the corresponding transitions carry sufficient information; instead, potential communications with the context are not even taken into account because the operational semantics of COWS does not allow them to take place. We consider this as a major limitation of our approach and plan to overcome it in the near future (see the next section). Parametric formulae have been also used in the logic for expressing properties of calculi with process distribution and remote actions introduced by De Nicola and Loreti [2004], and in its extension with stochastic features [De Nicola et al. 2007]. However, the parameters are only used in a limited way for capturing source and/or target localities of actions and making the properties depend on their identities. As far as we know, ours is the first 'general purpose' framework which is able to deal with formulae parametrization and general, dynamic, on-the-fly, possibly multiple, formulae instantiation.

Two other related logic-based approaches to the specification and verification of service-oriented systems have been proposed by Fu et al. [2005] and by Abreu et al. [2007]. The first one presents the static analysis tool WSAT that takes as an input service specifications written in e.g. WS-BPEL and WSDL, and, after a few translation steps, produces a Promela specification. This specification is then used as a model to verify the desired system properties, written in LTL, through the model checker SPIN. While this approach satisfactorily captures the control flows of static web services, differently from ours it does not handle many important behavioral aspects as e.g. dynamic establishment of communication channels (to dynamically determine the peer to talk to), dynamic service instantiation and correlation sets. The second approach employs SRML [Fiadeiro et al. 2006] to specify service-oriented architectures and introduces a logic to express the properties of interaction protocols. The approach is quite different from ours because while the SRML logic is targeted to the description of the properties of a single client-server long-running interaction, we focus on the analysis of service-oriented systems possibly involving many client-server interactions. Moreover, the SRML logic is based on a fixed, standardized set of interaction and communication mechanisms while in our case the user is free to define its own set of communication and interaction patterns.

In fact, 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.

9. CONCLUSIONS

We have tackled the problem of analysing the functional behaviour of formal specifications of service-oriented computing systems. To this aim, we have defined SocL, a temporal logic capable of representing distinctive aspects of services and have used it to express a set of desirable functional properties of services. We have also developed a bounded, on-the-fly model checker engine for this logic and, on top of it, we have built CMC to check satisfaction of SocL formulae by COWS terms. By means of a case study we have illustrated an application of our logical verification methodology: first, single out a set of abstract properties describing desirable specific features of the service under analysis; then, express such properties as SocL formulae; finally, exploit CMC for verifying satisfaction of the formulae by the COWS specification of the service.

The approach presented here has been fully implemented as a freely available multi-platform interpreter and model-checker designed around efficient verification procedures, which exploits the on-the-fly evaluation approach to minimize the need of state space generation. Our goal was to develop an in-house model-checking engine which can serve as a test bed to experiment with different temporal logics for service-oriented systems, with efficient (hence, scalable) on-the-fly verification procedures (even if the verification of extremely large systems was not considered among the short term goals), and with different kinds of user interfaces. For example, CMC web-based interface nicely integrates model exploration, verification and minimization in an interactive context, independent of the client platform (it just needs a browser) and does not require installation of any piece of software. Instead, CMC core command-line-oriented binaries permit an easy integration of the tool within other frameworks (like Eclipse plugins or graphical Java interfaces).

Our approach is not intended to be applied as it is to the development of industrial service-oriented applications; nevertheless, the usage of COWS, a formalism for which mappings have been already provided for different industry level SOC modelling languages (WS-BPEL [Lapadula et al. 2007], UML4SOA [Banti et al. 2010], SRML [Bocchi et al. 2009]), paves the way for such development. Furthermore, the patterns of service properties defined in Section 2.3 give an abstract view, independent from actual service descriptions, and the usage of such patterns can be in principle automated so that a user need not to delve into the technical details of SocL. The approach and the tool presented can hence serve as the (formally sound) basis on which an industrial strength SOC development and verification environment could be built.

Finally, we leave for future work the extension of our environment to support a more compositional verification methodology. In fact, systems of services can currently be analysed only ‘as a whole’, since computations requiring communication with external (i.e. not explicitly modelled) components are not taken into account. This is related to the original semantics of COWS that follows a ‘reduction’ style (albeit the transition labels are rather informative); the ‘symbolic’ operational semantics of COWS introduced by Pugliese et al. [2009] should permit to overcome this limitation and we intend to rely on it for implementing future versions of CMC.

REFERENCES

Abreu, J., Bocchi, L., Fiadeiro, J., and Lopes, A. 2007. Specifying and composing interaction protocols for service-oriented system modelling. In *FORTE*. LNCS Series, vol. 4574. Springer, 358–373.

Alonso, G., Casati, F., Kuno, H. A., and Machiraju, V. 2004. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer.

Baier, C., Cloth, L., Haverkort, B. R., Kuntz, M., and Siegle, M. 2004. Model Checking Action- and State-Labelled Markov Chains. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 701–710.

Banti, F., Pugliese, R., and Tiezzi, F. 2010. An accessible verification environment for UML models of services. *Journal of Symbolic Computation*. To appear.

Bhat, G., Cleaveland, R., and Grumberg, O. 1995. Efficient on-the-fly model checking for CTL*. In *LICS*. IEEE Computer Society, 388–397.

Bocchi, L., Fantechi, A., Gönczy, L., and Koch, N. 2006. Prototype language for service modelling: SOA ontology in structured natural language. Sensoria deliverable D1.1a.

Bocchi, L., Fiadeiro, J., Lapadula, A., Pugliese, R., and Tiezzi, F. 2009. From Architectural to Behavioural Specification of Services. In *FESCA*. ENTCS Series, vol. 253/1. Elsevier, 3–21.

Boreale, M., Bruni, R., De Nicola, R., and Loreti, M. 2008. Sessions and Pipelines for Structured Service Programming. In *FMOODS*. LNCS Series, vol. 5051. Springer, 19–38.

Bouali, A., Gnesi, S., and Larosa, S. 1994. JACK: Just Another Concurrency Kit. The integration Project. *Bulletin of the EATCS 54*, 207–223.

Bradfield, J. and Stirling, C. 2001. *Handbook of Process Algebra* (Bergstra, J. A. and Ponse, A. and Smolka, Scott A. Eds.). North-Holland, Chapter Modal logics and mu-calculi: an introduction, 293–330.

Bruni, R., Lanese, I., Melgratti, H., and Tuosto, E. 2008. Multiparty sessions in SOC. In *COORDINATION*, D. Lea and G. Zavattaro, Eds. LNCS Series, vol. 5052. Springer, 67–82.

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. 2006. Choreography and orchestration conformance for system design. In *COORDINATION*. LNCS Series, vol. 4038. Springer, 63–81.

Carbone, M., Honda, K., and Yoshida, N. 2007. Structured communication-centred programming for web services. In *ESOP*. LNCS Series, vol. 4421. Springer, 2–17.

Chaki, S., Clarke, E., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., and Veith, H. 2005. State/event software verification for branching-time specifications. In *IFM*. LNCS Series, vol. 3771. Springer, 53–69.

Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., and Sinha, N. 2004. State/event-based software model checking. In *IFM*. LNCS Series, vol. 2999. Springer, 128–147.

Clarke, E. and Emerson, E. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*. LNCS Series, vol. 131. Springer, 52–71.

Clarke, E., Grumberg, O., and Long, D. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst. 16,* 5, 1512–1542.

Clarke, E. M., Grumberg, O., and Peled, D. 1999. *Model Checking*. MIT Press.

Dam, M. 1996. Model checking mobile processes. *Inf. Comput. 129,* 1, 35–51.

Dam, M. 2003. Proof Systems for Pi-Calculus Logics. In *Logic for Concurrency and Synchronisation*. Trends in Logic, Studia Logica Library. Kluwer, 145–212.

Dams, D., Gerth, R., and Grumberg, O. 1997. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst. 19,* 2, 253–291.

De Nicola, R., Katoen, J., Latella, D., Loreti, M., and Massink, M. 2007. Model checking mobile stochastic logic. *Theor. Comput. Sci. 382,* 1, 42–70.

De Nicola, R. and Loreti, M. 2004. A modal logic for mobile agents. *ACM Trans. Comput. Log. 5,* 1, 79–128.

De Nicola, R. and Loreti, M. 2007. Multi Labelled Transition Systems: A Semantic Framework for Nominal Calculi. In *LMCS*. ENTCS Series, vol. 169. Elsevier, 133–146.

De Nicola, R. and Vaandrager, F. 1990. Action versus state based logics for transition systems. In *Proceedings of the Ecole de Printemps on Semantics of Concurrency*. LNCS Series, vol. 469. Springer, 407–419.

De Nicola, R. and Vaandrager, F. 1995. Three logics for branching bisimulation. *J. ACM 42,* 2, 458–487.

Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., and Tiezzi, F. 2008. A model checking approach for verifying COWS specifications. In *FASE*. LNCS Series, vol. 4961. Springer, 230–245.

Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., and Tiezzi, F. 2010. Specification and Analysis of an Automotive Scenario. Tech. rep., DSI, Università di Firenze. http://rap.dsi.unifi.it/cows/papers/automotiveScenario_in_cows.pdf.

Fernandez, J., Jard, C., Jéron, T., and Viho, C. 1996. Using on-the-fly verification techniques for the generation of test suites. In *CAV*. LNCS Series, vol. 1102. Springer, 348–359.

Ferrari, G. L., Gnesi, S., Montanari, U., and Pistore, M. 2003. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol. 12,* 4, 440–473.

Fiadeiro, J., Lopes, A., and Bocchi, L. 2006. A formal approach to service component architecture. In *WS-FM*. LNCS Series, vol. 4184. Springer, 193–213.

Fu, X., Bultan, T., and Su, J. 2005. Synchronizability of conversations among web services. *IEEE Trans. Software Eng. 31,* 12, 1042–1055.

Grumberg, O. and Veith, H., Eds. 2008. *25 Years of Model Checking - History, Achievements, Perspectives*. LNCS Series, vol. 5000. Springer.

Gudgin, M., Hadley, M., and Rogers, T. 2006. Web Services Addressing 1.0 - Core. Tech. rep., W3C. May. W3C Recommendation.

Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., and Zavattaro, G. 2006. SOCK: a calculus for service oriented computing. In *ICSOC*. LNCS Series, vol. 4294. Springer, 327–338.

Hennessy, M. and Liu, X. 1995. A modal logic for message passing processes. *Acta Informatica 32,* 4, 375–393.

Hennessy, M. and Milner, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32,* 1, 137–161.

Kavantzas, N., Burdett, D., and Ritzinger, G. 2004. Web Services Choreography Description Language version 1.0. Tech. rep., W3C. http://www.w3.org/TR/ws-cdl-10/.

Kindler, E. and Vesper, T. 1998. ESTL: A Temporal Logic for Events and States. In *Application and Theory of Petri Nets*. LNCS Series, vol. 1420. Springer, 365–384.

Koch, N. 2007. Automotive case study: UML specification of on road assistance scenario. Sensoria report, http://rap.dsi.unifi.it/sensoria/files/FAST_report_1_2007_ACS_UML.pdf.

Lanese, I., Vasconcelos, V., Martins, F., and Ravara, A. 2007. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*. IEEE Computer Society, 305–314.

Lapadula, A., Pugliese, R., and Tiezzi, F. 2006. A WSDL-based type system for WS-BPEL. In *COORDINATION*. LNCS Series, vol. 4038. Springer, 145–163.

Lapadula, A., Pugliese, R., and Tiezzi, F. 2007. A Calculus for Orchestration of Web Services. In *ESOP*. LNCS Series, vol. 4421. Springer, 33–47. Full version available at http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf.

Lapadula, A., Pugliese, R., and Tiezzi, F. 2008. Specifying and Analysing SOC Applications with COWS. In *Concurrency, Graphs and Models*. LNCS Series, vol. 5065. Springer, 701–720.

Lawford, M., Ostroff, J., and Wonham, W. 1996. Model Reduction of Modules for State-Even Temporal Logics. In *FORTE*. IFIP Conference Proceedings Series, vol. 69. Chapman & Hall, 263–278.

Lin, H. 1993. A verification tool for value-passing processes. In *Symposium on Protocol Specification, Testing and Verification*. North-Holland, 79–92.

Mateescu, R. and Thivolle, D. 2008. A model checking language for concurrent value-passing systems. In *Formal Methods*. LNCS Series, vol. 5014. Springer, 148–164.

Mayer, P., Schroeder, A., and Koch, N. 2008. Mdd4soa: Model-driven service orchestration. In *EDOC*. IEEE Computer Society Press, 203–212.

Meolic, R., Kapus, T., and Brezocnik, Z. 2008. ACTLW - an action-based computation tree logic with unless operator. *Elsevier Information Sciences 178,* 6, 1542–1557.

Meredith, L. and Bjorg, S. 2003. Contracts and types. *Commun. ACM 46,* 10, 41–47.

Moller, F. and Stevens, P. 1999. Edinburgh Concurrency Workbench user manual. http://homepages.inf.ed.ac.uk/perdita/cwb/.

Müller-Olm, M., Schmidt, D., and Steffen, B. 1999. Model-checking: A tutorial introduction. In *SAS*. LNCS Series, vol. 1694. Springer, 330–354.

OASIS WSBPEL TC. 2007. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS. April. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

Pecheur, C. and Raimondi, F. 2006. Symbolic Model Checking of Logics with Actions. In *MoChArt*. LNCS Series, vol. 4428. Springer, 113–128.

Pugliese, R., Tiezzi, F., and Yoshida, N. 2009. A Symbolic Semantics for a Calculus for Service-Oriented Computing. In *PLACES*. ENTCS Series, vol. 241. Elsevier, 135–164.

SENSORIA. 2005. Software engineering for service-oriented overlay computers. http://sensoria.fast.de.

Stirling, C. 2001. *Modal and Temporal Properties of Processes*. Springer.

Stirling, C. and Walker, D. 1989. Local model checking in the modal μ-calculus. In *TAPSOFT*. LNCS Series, vol. 354. Springer, 369–383.

ter Beek, M., Fantechi, A., Gnesi, S., and Mazzanti, F. 2008. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *FMICS*. LNCS Series, vol. 4916. Springer, 133–148.

ter Beek, M., Gnesi, S., Koch, N., and Mazzanti, F. 2008. Formal verification of an automotive scenario in service-oriented computing. In *ICSE*. ACM, 613–622.

Tiezzi, F. 2009. Specification and analysis of service-oriented applications. Ph.D. thesis, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze. http://rap.dsi.unifi.it/cows.

Tiu, A. 2005. Model Checking for pi-Calculus Using Proof Search. In *CONCUR*. LNCS Series, vol. 3653. Springer, 36–50.

van Breugel, F. and Koshkina, M. 2006. Models and verification of BPEL. Tech. rep., Department of Computer Science and Engineering, York University. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. 2003. Workflow patterns. *Distributed and Parallel Databases 14,* 1, 5–51.

Victor, B. and Moller, F. 1994. The Mobility Workbench — a tool for the π-calculus. In *CAV*. LNCS Series, vol. 818. Springer, 428–440.

Vieira, H., Caires, L., and Seco, J. C. 2008. The conversation calculus: A model of service-oriented computation. In *ESOP*. LNCS Series, vol. 4960. Springer, 269–283.

Yang, P., Ramakrishnan, C. R., and Smolka, S. A. 2004. A logical encoding of the π-calculus: model checking mobile processes using tabled resolution. *International Journal on Software Tools for Technology Transfer 6,* 1, 38–66.