Tools and Verification*

Massimo Bartoletti[1], Luís Caires[2], Ivan Lanese[3], Franco Mazzanti[4], Davide Sangiorgi[3], Hugo Torres Vieira[2], and Roberto Zunino[5]

[1] Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy
bart@unica.it
[2] CITI and Dep. de Informatica, FCT, Universidade Nova de Lisboa, Portugal
luis.caires@di.fct.unl.pt, htv@fct.unl.pt
[3] Focus Team, Università di Bologna/INRIA, Italy
{lanese,davide.sangiorgi}@cs.unibo.it
[4] ISTI-CNR, Pisa, Italy
franco.mazzanti@isti.cnr.it
[5] Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento, Italy
zunino@disi.unitn.it

Abstract. This chapter presents different tools that have been developed inside the Sensoria project. Sensoria studied qualitative analysis techniques for verifying properties of service implementations with respect to their formal specifications. The tools presented in this chapter have been developed to carry out the analysis in an automated, or semi-automated, way.

We present four different tools, all developed during the Sensoria project, exploiting new techniques and calculi from the Sensoria project itself.

1 Introduction

This chapter presents a set of tools that have been developed inside the Sensoria project for analysis and verification of service-oriented systems. The tools allow the application of novel analysis techniques for service-oriented systems that have been studied inside the project. Those tools are (partly) based on calculi and models described in Chapter 2-1. Also, they have been validated by applying them to the Sensoria case studies (described in Chapter 0-3), as illustrated in Chapter 7-4 for the COWS Model Checker (CMC). This experimentation has provided useful feedback for improving the tools themselves.

We describe four different tools in detail, all developed within the Sensoria project and based on new techniques and calculi introduced in the project itself. While referring to the next sections and to the publications in the bibliography for a more detailed description of the tools and of the underlying theory, we give here a short outline of each of them.

* This work has been partially sponsored by the project Sensoria, IST-2005-016004.

M. Wirsing and M. Hölzl (Eds.): Sensoria Project, LNCS 6582, pp. 408–427, 2011.

CMC and UMC model checkers: CMC (COWS Model Checker) and UMC (UML Model Checker) are two prototypical instantiations of a common logical verification framework for the analysis of functional properties of service-oriented systems. Both tools have the goal of model-checking properties specified in Socl (the Service Oriented Computing Logic), and they differ just for the underlying computational models, which are built out from COWS (see Chapter 2-1) specifications in the case of CMC, and UML statecharts in the case of UMC. In both cases, the specifications are mapped onto Doubly Labeled Transition Systems, in which transitions are labeled by sets of observable events. The on-the-fly model checking technique is used to avoid statespace explosion. In this chapter we describe the tools themselves, while the underlying logic and the algorithms exploited by them have been described in Chapter 4-2.

ChorSLMC: ChorSLMC (Choreography Spatial Logic Model Checker) is a verification tool for service-based systems implemented as an extension to SLMC, a framework for model checking distributed systems against properties expressible in dynamic-spatial logic. Descriptions of participants may be specified either in the Conversation Calculus [29] (see also Chapter 2-1), a core calculus for service-oriented computing developed within the SENSORIA project, or in a fragment of WS-BPEL [1], while choreographic descriptions may be specified in an abstract version of WS-CDL [33]. The tool may also be used on service-based systems to check other interesting properties of typical distributed systems, using the core dynamic-spatial logic available in SLMC.

LocUsT: the LocUsT tool is a model checker for *usages*, abstract descriptions of the behavior of services. Usages are expressed in a simple process calculus. They over-approximate all the possible execution traces of a service, focusing on resource creation and access. *Usage policies* are then used to express constraints on the use of resources, by identifying the forbidden patterns. A policy is represented through a finite state automaton parametrized over resources. LocUsT takes as input a usage and a policy, and decides whether a trace of the usage that violates some instantiation of the policy exists.

The CMC and UMC tools are strongly related, just differing on the format of the description of the model of the system to be analyzed, and concentrate on verifying behavioral properties expressed in Socl logic. ChorSLMC also concentrates on behavioral properties, but they are verified by checking conformance to a choreographic description. LocUsT instead tackles a different problem, concentrating more on the security aspects, and allowing to check that resources are used according to a specified policy.

2 CMC-UMC Verification of Service-Oriented Models

CMC (COWS Model Checker) and UMC (UML Model Checker) [23, 27] are two prototypical instantiations of a common logical verification framework for the verification of functional properties of service-oriented systems. They differ

just for the underlying computational models which are built out from COWS [20, 21] specifications in the case of CMC, and out from UML [28] statecharts in the case of UMC. For verification of service-oriented models we do not intend just the final "validation" step of a completed architecture design, but rather a formal support during all the steps of the incremental design phase (hence when running designs are still likely to be incomplete and with high probability to contain mistakes). Indeed CMC/UMC have been developed having in mind the needs of a system designer which intends to take advantage of formal approaches to achieve an early validation of the system requirements and an early detection of design errors. From this point of view the design of CMC/UMC has been driven by the desire to achieve the following goals (or, at least, to experiment in the following directions):

- The support of a good user experience (easiness of use) in the computer-aided application of formal methods.
- The support of abstraction mechanisms allowing to observe the system at an high level of abstraction, hiding all the irrelevant and unnecessary computational details.
- The possibility to explore step by step the possible system evolutions and the possibility to generate a "summary" of system behavior in terms of minimal abstract traces.
- The possibility to investigate detailed and complex system properties using a parametric branching time temporal logic supported by an on-the-fly model checker.
- The possibility of obtaining an understandable explanation of the model-checking results.

In the following we will briefly present the achieved results with respect to the above five points.

User Experience. Several kinds of user interfaces have been experimented in the attempt to make possible the access to verification facilities also by non technical people. This without losing the possibility to tune and control the verification environment in a more advanced way. In particular:

- CMC/UMC are accessible as web applications to allow their experimentation and use without any kind of local installation, and exploiting the friendliness and flexibility of hypertextual documents to support the interactions with the user.
- CMC/UMC are usable with a simple, platform independent, java-based, graphical interface to achieve offline model exploration and verification.
- CMC/UMC are available as binary, platform-specific, command line oriented applications (for Mac, Windows, Linux and Sun systems) to exploit the simplest, most efficient, and finest level of interaction and control of the system verification and exploration.
- Models can be edited as simple textual documents.

- UML Statechart models can also be edited through a dedicated graphical interface.
- UML Statechart models can be extracted from standard UML XMI documents.

Abstraction Mechanisms. In our context, services are considered as entities which have some kind of abstract internal state and which are capable of supporting abstract interactions with their clients, like for example accepting requests, delivering corresponding responses and, on-demand, canceling requests. Moreover, concrete operational models, with a specific concrete operational semantics, are used to describe the details of the system states and their possible evolution steps. This means that an abstraction mechanism needs to be applied to the system state description and to the system evolution information. This mechanism allows to extract from the operational semantics of the specific computational model the relevant aspects we want to observe. In our tools this abstraction step is achieved via a list of pattern matching rules which allow to specify which state properties and which transition events we want to observe. These rules are presented as structured actions of the form "mainlabel(flag,flag,..)". When this abstraction step is performed, the semantic model of a service-oriented system can be seen as a doubly labeled transitions system (L^2TS), where both the states and the edges are labeled with sets of the above described structured actions. This abstract L^2TS induced by the operational semantics of the system will constitute the reference structure used by the logic as interpretation domain and by the full-trace minimization algorithm to generate and display the abstract minimized views of the system.

CMC is the instantiation of our verification framework with respect to the COWS process calculus. COWS ha been explicitly defined for the specification and orchestration of services and combines in an original way constructs and well known features like asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities. The abstraction rules of CMC allow to "intercept" the communication actions occurring between two COWS processes and present them as request/response events in the context of some client-server interaction. The corresponding abstract labels will therefore appear on the edges of the L^2TS as they represent the abstract events occurred during an evolution step. The CMC abstraction rules moreover allow to observe the willingness of a COWS term to participate to a communication synchronization (e.g., its willingness to perform the input side of the synchronization) and present it as a state property reflecting the willingness of a service to accept operation requests. In this case this abstract property will appear as an abstract label associated to some states of the L^2TS.

UMC is instead the instantiation of our verification framework with respect to UML statecharts. These have a standard presentation and semantics as defined by the OMG (Object Management Group). The communication events which can be observed in this case are based on the notion of message passing. Indeed, we can distinguish the event of sending an operation request (on the client side) from the event of accepting that request (on the server side). Moreover UML

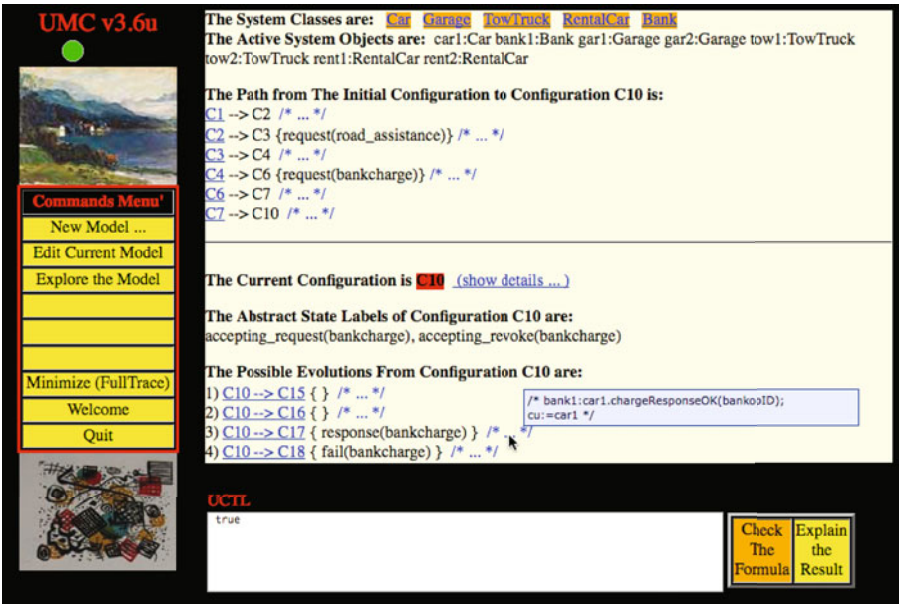

Fig. 1. UMC manual model exploration page

statecharts are built over the concept of local attribute of objects, and during the execution of a system transition (beyond multiple communication actions) several updates of the local object attributes can be executed. The abstraction rules of UMC allow to observe all these events (acceptance of a message, sending of a message, update of a local attribute) as abstract events representing relevant aspects of the service-oriented behavior of the system, and represent them as abstract labels associated to the L^2TS edges. Other abstraction rules of UMC allow instead to observe the specific value of selected object attributes, and whether or not an object in a specific state, and present this information as abstract state predicates labeling the states of the L^2TS.

Step by Step Exploration. The first and simpler way to explore a CMC/UMC model is to manually navigate through its L^2TS structure, observing at each step the set of possible immediate evolutions, the set of abstract events occurring during these evolutions, the set of abstract properties holding in the current state and, if desired, also all the ground details of the underlying computational model with respect to the current state structure and evolutions. The web application interface (shown in Fig. 1), thanks to the use of tooltips, colors and hyperlinks, makes this exploration experience more immediate.

Abstract Minimized Traces. The possibility of selecting a small set of abstract events of interest and, starting from them, compute and observe the minimized full-trace abstract view of the system is an extremely powerful way of

checking whether the system behavior matches the intended requirements. This works even in the case in which the requirements themselves are not fully clear or well formalized. Let us consider, for example, the automotive case study described in [25]. This is formalized as a collection of UMC statecharts, which is a model constituted by several hundreds of states. Suppose we are interested in observing only the "bank" related events and the "garage" related events. Using the appropriate abstractions and using UMC to build the minimized model with respect to them we obtain the L^2TS shown in Fig. 2, which summarizes all the possible system traces with respect to the observed set of events. It is extremely easy to become confident of the correctness of the model just looking at the L^2TS, without being forced to identify a priori a *complete* set of requirements and formalize them in terms of logic formulas for being separately model checked. Unfortunately the abstract minimization approach to system verification has also some drawbacks:

- It is computationally expensive: for very large models it might be too much resource consuming to compute its abstract, minimal full-trace view.
- If the L^2TS is not finite, it is not even a matter of available computing resources. Building the abstraction is not possible.
- The abstract view completely lost the connection with the original "concrete" computational model. If the system behavior is not the expected one, no immediate way is available to reconstruct unexpected computations in the concrete model.
- If the resulting chart is rather complex, relying on just the intuition to assess its correctness is unreliable and lacks of concrete formal evidence.

Socl Model Checking. To overcome the drawbacks of the previous approach, as well as to directly formalize and check *specific* functional / safety / liveness requirements of a system, a verification technique based on *on-the-fly, bounded model checking of Socl formulas* is considered. This approach also permits to reduce the average verification time and, at the same time, performing some verification also in the case of non finite-state systems. Socl [15] is a service-oriented temporal logic derived from UCTL [16, 17, 24] of which we recall here the most important characteristics:

- It is a branching time logic, built over the classical intuitive "eventually" (F), "always" (G), "until" (U), "next" (X) temporal operators. The evaluation of this logic is known [7] to be achievable with a computational complexity which is linear with respect to the size of the formula and the size of the model.
- It is an event and state based logic. Being its interpretation domain our abstract state/event based L^2TS structures, Socl allows to directly express state predicates to be evaluated over the abstract labels associated to the states of the L^2TS, and action expressions to be evaluated over the abstract labels associated to the edges of the L^2TS.

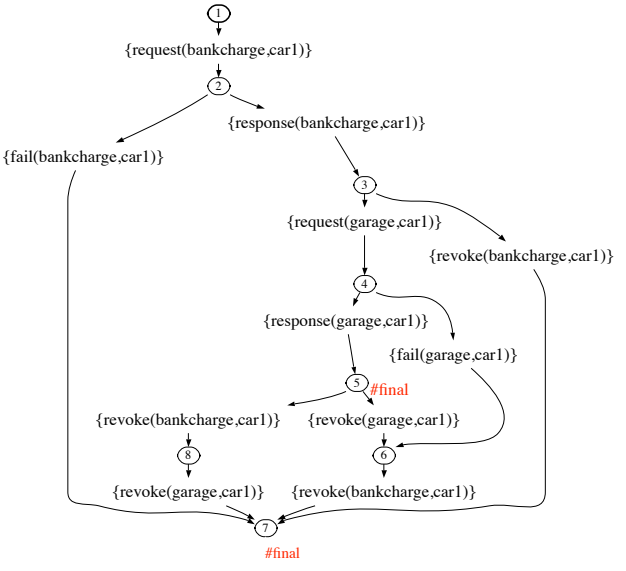

Fig. 2. An abstract view of the automotive case study

– It is a parametric temporal logic, in which the values of the arguments of an abstract event occurring during a transition can be used to dynamically instantiate a parametric subformula to be evaluated in the target state of the transition itself.

Socl is supported in CMC/UMC by an on-the-fly model checking algorithm which generates the model statespace *on demand* according to the flow of the evaluation. The UCTL formula is evaluated adopting a top-down traversal of the structure of the formula itself minimizing the need of the model statespace generation (which is explored in a depth-first way); a *bounded* [8] model checking approach is also used to try to produce an evaluation result also in the case of infinite state models.

The Socl verification engine is exactly the same in both CMC and UMC since it is based on the abstract L^2TS computed from the models, and not on the specific concrete computational models defined by input model specification languages. In the following we show some examples of Socl formulas, just to give an intuition of its structure, referring to [15] for the details of its definition and for its formal presentation. These examples are written with respect to the same abstraction rules used to generate the abstract minimized traces shown in Fig. 2. With respect to the above scenario we can check, for example, that: "*It is always true (AG) that an unsuccessful response from the garage to a client is always eventually (AF) followed by a revoke operation to the bank, on behalf of the*

same client". This property can be formalized in Socl (following the CMC/UMC syntax) as:

```
AG [fail(garage, $client)] AF{revoke(bankcharge,%client)} true.
```

Another general property that we can check with respect to the same scenario is that: "*It is always true (AG) that a request for an operation is always followed (AF) either by a successful response to that operation or by a failure notification*". In this case the property can be formalized as:

```
AG [request($operation,$client)]
      AF {response(%operation,%client)
            or fail (%operation,%client)}.
```

Proofs and Counterexamples. It is well known that providing a counterexample for a given temporal logic formula is quite easy in the case of linear time logics and quite complex in the case of branching time logics. The problems to be solved for the generation of useful proofs/counterexamples are essentially three:

- The proof/counterexample is not (in general) based on a single execution path of the system, but may be based on a subgraph of the L^2TS modeling the system.
- Not all the L^2TS states needed by the proof/counterexample are in general "useful" from the point of view of the user.
- The information on the set of L^2TS states needed by the proof/counterexample is sometimes not sufficient to produce usable feedback to the user. We might need to provide feedback also on which subformula was being evaluated when the L^2TS states have been explored.

Let us consider, for example, a simple formula of the kind: "*(AG predicate1) or (AG predicate2)*". If this formula does not hold, its counterexample has the form of a pair of paths, one leading to a state in which *predicate1* does not hold, and another leading to a state in which *predicate2* does not hold.

Let us consider, moreover, the formula: "*EF predicate*". If this formula does not hold, its counterexample would coincide with the full system statespace, however it would be completely pointless to provide the user with an exhaustive list of all the states for which the *predicate* does not hold. On the contrary, if the formula holds the user might be interested in the sequence of steps which would prove it.

Let us consider, as third example, the formula "*EF AG predicate*". If the formula holds for a certain system, the user might be interested in the proof for the first part of the formula, showing an execution path which, starting from the initial state would lead to an intermediate state for which the subformula "*AG predicate*" holds; once identified that intermediate state all the other states reachable from it belonging to the proof of the subformula "*AG predicate*" would probably add only irrelevant noise and complexity to the original information. The "useful" part of the proof, would be constituted by a fragment of the full proof. CMC/UMC tries to convey to the user what is supposed to be the "useful" part of a proof or counterexample, but only more experience might consolidate the identification of the "best" reasonable behavior.

Application to the Case Studies. The design and development of the prototypes has greatly taken advantage from the early experiences gained through their application to the SENSORIA case studies. The first of these applications has been the use of UMC for the analysis of communication protocols for service-oriented applications [24, 26]. Subsequently the SENSORIA automotive case study has been the stimulus for the first experimentations with the Socl logic and the COWS language [15]. The same case study has also been specified in terms of UML statecharts and verified with UMC [25], thus experimenting with the SENSORIA UML profile for SOA [19, 22]. Finally both COWS/CMC and UML/UMC have been applied for the formalization and verification of the SENSORIA Credit Portal case study (see Chapter 0-3).

3 Model-Checking Service Conversations with ChorSLMC

A service-based system is a decentralized coordinated distributed system, where independent partners interact by message passing. It is then useful to consider the extension of automated verification techniques, based on model-checking, to service-oriented models, able to certify the general "standard properties" of concurrent distributed systems, such as reachability, termination, liveness, race-freedom, just to refer a few. We may also be interested in domain specific invariants. This class of properties is easily expressible in some kind of temporal logic. Adding to these, it is well known that to describe interactions among partners in a service relationship two viewpoints are considered particularly useful: orchestration and choreography.

"Orchestration" focuses on the coordination of several partners from the local viewpoint of a single participant, for the purpose of providing a new functionality or service to the external world, "choreography" describes the global behavior of a system that emerges from the interaction of several independent participants. An orchestration can be seen as the description of a workflow process, with its own control flow graph, while a choreography, just like a message sequence chart, describes the message exchanges between a group of partners involved in a complex transaction. Orchestration specification languages are programming languages, with a definite operational semantics (cf. WS-BPEL [1] and various service-oriented calculi described in Chapter 2-1), while choreography languages (cf. WS-CDL [33] and the calculus of [13]) define global behaviors of composite systems "without a single point of control", and are not intended to be "executable". Therefore, in addition to common behavioral-temporal properties, an important analysis problem in service-oriented computing is to check conformance of local descriptions (orchestrations) with respect to choreographies (cf. [9, 14]). Specifying (and checking) conformance of localized process interactions against choreographies requires a specification language able to talk about the internal spatial structure of a concurrent system, and its dynamic evolution. Such expressiveness falls out of the scope of extensional behavioral logics such as Hennessy-Milner logic and variants (and supporting tools).

We have developed a fairly simple, yet powerful, technique, building on dynamic-spatial logics and model-checking [10, 11], particularly appropriate for this class of analysis problems. We have also implemented a supporting tool ChorSLMC, which is an extension of SLMC, a dynamic-spatial logic model-checker. The tool may be used to check not only choreography conformance, but many other key properties of service-oriented systems, such as race-freedom and deadlock absence, and system invariants, that may be easily expressed in the underlying logical framework.

Approach. Our approach to the choreographic analysis problem relies on language translations, and on the reuse of previously developed model-checking techniques for spatial logic and related tools. More concretely, we have developed provably correct encodings, allowing local descriptions of partner sites, expressed in a service-oriented calculus, to be adequately translated into a lower level analysis language (a dialect of the π-calculus), and global descriptions (choreographies), to be adequately translated into dynamic-spatial logic formulas. The correctness of our translation ensures that a system *System*, expressed in the core Conversation Calculus [29] (described in Chapter 2-1 and referred below by CC) or, alternatively, in a simple dialect of WS-BPEL, conforms to a choreography *Choreography*, expressed in a WS-CDL dialect, if and only if its π-calculus translation satisfies the corresponding dynamic-spatial logic formula.

$$[\![System]\!] \models [\![Choreography]\!]$$

The correctness of the translation between the source language (either CC or WS-BPEL) is obtained by observing that for model-checking purposes, we don't really need full abstraction but just some suitable operational correspondence. The encoding of choreographies in the logic is supported by the structural observational power of spatial logics, that allow observation of internal message exchanges, unobservable by purely behavioral logics such as those supported by other existing model checking tools. Choreography conformance of service-oriented systems is then reduced to a model-checking problem that may be easily handled by existing tools, namely the Spatial Logic Model Checker (SLMC) [31, 32] (started to be developed in Global Computing 1 Project Profundis, and extended during Global Computing 2 Project SENSORIA). The structural observation power of spatial logics turns out to be essential in this application to choreographic verification, since, e.g., the message exchanges mentioned in a choreographic description are not observable by traditional process logics invariant under behavioral equivalences. Thus, general process logics and tools that cannot observe internal message exchanges in a system would not be appropriate for the service verification problem we consider here. Both local descriptions of services, expressed in suitable orchestration languages, and the global choreographic descriptions, expressed in a WS-CDL dialect, are translated by ChorSLMC into π-calculus / dynamic-spatial logic specifications, respectively, which are directly fed to the SLMC verification engine.

Input Specification Languages. The ChorSLMC tool supports two modeling languages for defining the behavior of partners in a service collaboration: a core fragment of the Conversation Calculus, obtained by removing exception handling primitives, and a fragment of WS-BPEL. The specification syntax is depicted below, and includes the basic constructors presented in Chapter 2-1. Both the CC model and the WS-BPEL model are detailed in [30].

$$
\begin{array}{rll}
\alpha ::= & LABEL!(\tilde{o}) & \text{(send here)} \\
| & LABEL?(\tilde{i}) & \text{(receive here)} \\
| & LABEL\hat{}!(\tilde{o}) & \text{(send up)} \\
| & LABEL\hat{}?(\tilde{i}) & \text{(receive up)} \\
P ::= & \mathsf{end} & \text{(inaction)} \\
| & \mathsf{context}\ n\ \{P\} & \text{(site)} \\
| & \alpha.P & \text{(action)} \\
| & \mathsf{switch}\ \{\alpha_1.P_1; \ldots ; \alpha_k.P_k\} & \text{(select)} \\
| & \mathsf{def}\ LABEL \Rightarrow P & \text{(service definition)} \\
| & \mathsf{new}\ n.LABEL \Leftarrow P & \text{(service instantiation)} \\
| & \mathsf{join}\ n.LABEL \Leftarrow P & \text{(conversation join)} \\
| & P_1 \mid P_2 & \text{(parallel)} \\
| & Id & \text{(process identifier)} \\
| & \mathsf{if}\ (\mathit{bool\ expr})\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 & \text{(conditional)}
\end{array}
$$

To describe choreographies, a fairly simplified version of the WS-CDL language is also considered, defined as an extension of the dynamic-spatial logic available in SLMC with specialized choreography operators as shown below. In such a way, it is possible to freely mix choreography operators with propositional and first order name quantification, spatial operators and fixpoint operators. The choreography fragment is close to the languages of global types introduced by [13, 18], and is also processed directly by the ChorSLMC tool.

$$
\begin{array}{rll}
A ::= & \mathsf{end} & \text{(no action)} \\
| & \mathsf{exchange}(n, LABEL, A) & \text{(may interaction in conversation } n\text{)} \\
| & \mathsf{exchanges}(n, LABEL, \mathit{arg}, A) & \text{(may interaction in conversation } n\text{)} \\
| & \mathsf{aexchange}(n, LABEL, A) & \text{(all interaction in conversation } n\text{)} \\
| & \mathsf{aexchanges}(n, LABEL, \mathit{arg}, A) & \text{(all interaction in conversation } n\text{)} \\
| & \mathsf{parallel}(A', A'') & \text{(parallel activities)} \\
| & \mathsf{choice}(A', A'') & \text{(choice)} \\
| & F & \text{(spatial logic formulae)}
\end{array}
$$

The language contains constructs to express parallel / choice flow and primitives to express message exchanges: $\mathsf{exchange}(n, LABEL, A)$ asserts that there is a message interaction on label $LABEL$ between two partners in conversation n and A specifies the behavior of the continuation; $\mathsf{exchanges}(n, LABEL, \mathit{arg}, A)$ specifies an extra argument arg which captures the conversation name exchanged in the communication; $\mathsf{aexchange}(n, LABEL, A)$ asserts that after all interactions on label $LABEL$ in conversation n the continuation satisfies behavior A. We

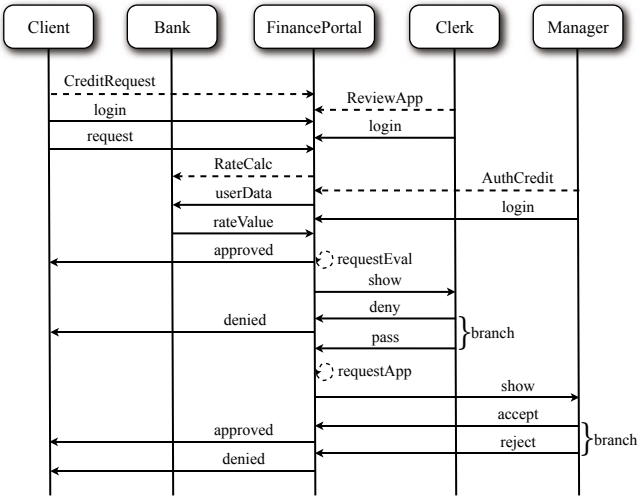

Fig. 3. Credit request message sequence chart

refer to [30] for a detailed explanation of our orchestration and choreography description language semantics, and the formal specification of their translation into the π-calculus and logic understood by the SLMC framework.

Simple Examples. We now illustrate the usage of the specification languages and of our tool. Consider the credit request scenario from the SENSORIA Financial Case Study described in Chapter 0-3, whose choreographic specification may be graphically depicted by the message sequence chart in Fig. 3. We specify the part of the choreography related to the interaction between the client, the finance portal and the bank as follows, using the basic choreographic language for CC systems (actually the input syntax for ChorSLMC).

```
defprop clientInteraction =
   maxfix Loop.
   hidden clientConv.
     exchanges(financePortal, creditRequest, clientConv,
     exchange(clientConv, login,
     exchange(clientConv, request,
     exchanges(bank, rateCalc, clientConv,
     exchange(clientConv, userData,
     exchange(clientConv, rateValue,
     choice(
        exchange(clientConv, approved, exchange(client,approved,Loop)),
        may_tau(clerkInteraction(clientConv,Loop)))))))))));
```

Notice that the exchange specification may be used not only to specify "regular" message exchanges, but also conversation initiation (`creditRequest`) and

conversation join (`rateCalc`) messages (see [12]). The behavior of each partner / role is then specified using the appropriate modeling language. We show the code for the *creditRequest* service definition:

```
defproc cc FinancePortalSpec1 =
  context financePortal {
    def creditRequest => (
      login?(uid).request?(data).
        join bank.rateCalc <= (
          userData!(data).rateValue?(rate).
          if (rate=aaa) then approved!().end
          else this(clientChat).
               requestEval^!(clientChat,uid,data).end))};
```

We specify the whole system as the composition of the roles of the finance portal, bank, client, clerk and manager. Also we specify that *clientInteraction* is the entry point of the global choreography.

```
defproc cc System = FinancePortalSpec1 | BankSpec | BankSpec2
       | ClientSpec | FinancePortalSpec2 | ClerkSpec
       | FinancePortalSpec3 | ManagerSpec;
defprop chor = clientInteraction;
```

After all definitions have been loaded into the ChorSLMC tool we may verify that the CC credit request system conforms to the prescribed choreography.

```
check System(up,here) |= chor;
Processing...
* Process System(up,here) satisfies the formula chor *
```

Notice that the tool may be used to automatically verify (for finite state models) not only choreographic conformance of composite service systems, but also common safety and liveness properties, such as invariant satisfaction, race and deadlock absence. For example:

```
check System(up,here) |= eventually(exchange(bank,rateCalc,true));
Processing...
* Process System(up,here) satisfies the formula
eventually (exchange(bank,rateCalc,true)) *
```

To conclude, the ChorSLMC tool provides a very flexible and powerful instrument to analyze general structural safety and liveness properties of service-oriented systems, expressed in languages which are familiar to software engineers, while building in solid process calculi and specification logic based foundations.

4 The LocUsT Tool

A fundamental concern of service-oriented applications is to ensure that resources are used correctly. Devising expressive, flexible and efficient mechanisms

to control resource usages is therefore a major issue in the design and implementation of languages for services. In [6], a comprehensive framework has been proposed for safely protecting code with usage policies, within a linguistic setting. Resource usage control is made feasible by suitably extending and integrating techniques from type theory and model-checking.

The LocUsT tool is the verification core of our framework. It takes as input a usage policy and a program abstraction (called a *usage*), and statically checks whether the abstraction complies with the policy. More precisely, LocUsT decides in polynomial time whether a trace of the given usage exists that violates the policy [5].

Usage Policies. Usage policies define safety properties on sequences of resource accesses and creations. We will define below our usage policies, and the compliance of a trace with a policy. First, we introduce some basic notions.

Resources are denoted with $r, r', \ldots \in \mathsf{Res}$, and they can be accessed through *actions* $\alpha, \alpha', \ldots \in \mathsf{Act}$. An *event* $\alpha(r_1, \ldots, r_k) \in \mathsf{Ev}$ models the action α (with arity $|\alpha| = k$) being fired on the target resources $r_1, \ldots, r_k$. The special action `new` represents the creation of a resource. *Traces* $\eta, \eta', \ldots \in \mathsf{Ev}^*$ are finite sequences of events.

Usage policies are an extension of finite state automata. Their edges have the form $\alpha(\boldsymbol{\rho})$, where $\boldsymbol{\rho} \in (\mathsf{Res} \cup \mathsf{Var})^{|\alpha|}$. We use final states to represent policy violations: a trace leading to a final state suffices to produce a violation. Two examples of usage policies are in Fig. 4.

Formally, a usage policy φ is a 5-tuple $\langle S, Q, q_0, F, E \rangle$, where:

- $S \subseteq \mathsf{Act} \times (\mathsf{Res} \cup \mathsf{Var})^*$ is the input alphabet,
- Q is a finite set of states,
- $q_0 \in Q \setminus F$ is the start state,
- $F \subset Q$ is the set of final "offending" states,
- $E \subseteq Q \times S \times Q$ is a finite set of edges, written $q \xrightarrow{\alpha(\boldsymbol{\rho})} q'$

Each usage policy φ denotes a set of traces, i.e. the traces that obey φ. The semantics of φ considers all the possible instantiations of its variables to actual resources: a trace η respects φ when η leads no instantiations of φ (on the resources in η) to an offending state.

Usage policies were first introduced in [3], where a block of code B could be sandboxed by a policy φ, so to require that φ must hold through the execution of B. The definition of policies has since then been revised several times, so to make them more expressive. In the original formulation, policies could only inspect sequences of actions, neglecting resources. In [4] policies can be parametrized over a single resource, and resources can be dynamically created; [5] deals with the general case of an arbitrary number of parameters.

Examples. Consider a Web application that allows for editing documents, storing them on a remote site, and sharing them with other users. The editor is

implemented as an applet run by a local browser. The user can tag any of her documents as *private*. To avoid direct information flows, the policy requires that private files cannot be sent to the server in plain text, yet they can be sent encrypted. This policy is modeled by $\varphi_{\mathsf{IF}}(x)$ below. After having tagged the file x as private (edge from q_0 to q_1), if x was to be sent to the server (edge from q_1 to q_2), then the policy would be violated: the double circle around q_2 marks it as an offending state. Instead, if x is encrypted (edge from q_1 to q_3), then x can be freely transmitted: indeed, the absence of paths from q_3 to an offending state indicates that once state q_3 is reached, the policy will not be violated on file x. A further policy is applied to our editor, to avoid information flow due to covert channels. It requires that, after reading a private file, any other file must be encrypted before it can be transmitted. This is modeled by $\varphi_{\mathsf{CC}}(x, y)$ below. A violation occurs if after some private file x is read (path from q'_0 to q'_2), then some other file y is sent (edge from q'_2 to the offending state q'_4).

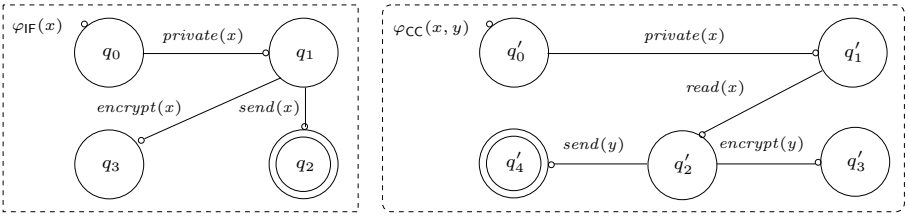

Fig. 4. The information flow policy $\varphi_{\mathsf{IF}}(x)$ and the covert channels policy $\varphi_{\mathsf{CC}}(x, y)$

Here is how the policies $\varphi_{\mathsf{IF}}(x)$ and $\varphi_{\mathsf{CC}}(x, y)$ are expressed in the LocUsT syntax. The field tagged `name` defines the name of the policy. The remaining fields describe the logic of the automaton. The tag `states` is for the set of states, `start` is for the initial state, and `final` is for the list of the final (offending) states. The tag `trans` preludes to the transition relation of the automaton.

```
name: phi_IF
states: q0 q1 q2 q3
start: q0
final: q2
trans:
q0 -- private(x) --> q1
q1 -- encrypt(x) --> q3
q1 -- send(x)    --> q2
```

```
name: phi_CC
states: q0 q1 q2 q3 q4
start: q0
final: q4
trans:
q0 -- private(x) --> q1
q1 -- read(x)    --> q2
q2 -- send(y)    --> q4
q2 -- encrypt(y) --> q3
```

Usages. Usages are program abstractions, expressed in a simple process calculus. They over-approximate all the patterns of resource accesses and creations of the service itself. Formally, usages have the following syntax:

$\mathtt{U}, \mathtt{U}' ::= \mathtt{0}$		empty
	$\alpha(\boldsymbol{\rho})$	event ($\boldsymbol{\rho} \in \mathsf{Res}^{\lvert\alpha\rvert}$)
	`nu n.U`	resource creation
	`U . U'`	sequence
	`U + U'`	choice
	φ`[U]`	policy framing
	`mu h.U`	recursion
	`h`	recursion variable

The usage `0` represents a computation not affecting resources. The usage $\alpha(\boldsymbol{\rho})$ is for a computation that executes the action α on the resources mentioned in $\boldsymbol{\rho}$. The usage `nu n.U` represents the creation of a resource `n`, which can then be used in `U` with the requirement that the first action on n must be a `new(n)` event. The operators `.` and `+` denote sequentialization and non-deterministic choice of usages, respectively. The usage φ`[U]` represents the fact that the policy φ has to be enforced on the usage `U`. The usage `mu h.U` stands for a recursion; the recursion variable `h` may occur in `U`.

For instance, consider the following usage:

```
phi_IF[nu n. new(n).private(n).(send(n)+encrypt(n))]
```

This usage will be rejected by the LocUsT model-checker, because a `send(n)` may occur in a trace after a `private(n)`, so violating the policy φ_{IF}.

The following usage will instead pass the model-checking, because the action `send` is not fired on a private document.

```
phi_IF[nu n. nu f. new(n).new(f).private(n).read(n).send(f)]
```

The following usage is rejected by the model-checker, because it violates the policy φ_{CC}. Note in fact that a file `f` is sent unencrypted after the private file `n` has been read.

```
phi_CC[nu n. nu f. new(n).new(f).private(n).read(n).send(f)]
```

The following trace is detected to attempt a violation of the policy φ_{CC}.

```
nu n. new(n).private(n).nu f. new(f).
    (mu h. phi_CC[ send(f) ] + read(n) . h)
```

After having read the private file `n` an arbitrary number of times, it may activate the policy φ_{CC}, within which sending the unencrypted file `f` is no longer permitted.

Finally, the following usage passes the model-checking, since the file `f` can only be sent after it has been encrypted:

```
nu n. new(n).private(n).nu f. new(f).
    (mu h. phi_CC[ send(f) . h ] + read(n) . encrypt(f) . h)
```

Service Call-by-Contract. So far, we have shown how LocUsT can verify that an abstraction of the service behavior does not violate a given policy. This

technique can serve as a foundation for a service composition framework, where services are orchestrated according to their behavioral properties.

In our framework, each service publishes the abstraction of its behavior (i.e. its usage) in a repository. Then, a client can ask for a service that respects a given property (expressed as a usage policy). This is done by querying the repository with that usage policy. Upon such request, the repository matches the given policy against the usages of the registered services. This task can be accomplished by the LocUsT tool. When LocUsT finds that the property requested by the client matches the usage of a service, the name of that service is forwarded to the client, which can then invoke the service using standard mechanisms.

Summing up, our technique allows for defining a call-by-contract invocation mechanism, which allows clients to abstract from the actual service names, and just consider the properties these services have to offer.

The theory underlying our call-by-contract invocation mechanism was originally introduced in [2]. There, a type and effect system and a model-checker were exploited to define a call-by-contract orchestrator. Call-by-contract is described in detail in Chapter 2-4.

Verification Technique. We now briefly recap the verification technique described in detail in [5], which is the one implemented in the LocUsT tool. Our algorithm is composed of several phases, summarized below.

1. **Regularization.** First, the usage is *regularized*, i.e. transformed so that in no trace a policy framing $\varphi[-]$ is entered twice: for instance $\varphi[\mathtt{U} \,.\, \varphi[\mathtt{U}']]$ becomes $\varphi[\mathtt{U} \,.\, \mathtt{U}']$. Particular care must be exercised when handling recursive usages such as $\mathtt{mu\ h}.\ \varphi[\mathtt{h} + \mathtt{U}]$.
2. **Conversion into BPA.** The usage is transformed in a process of Basic Process Algebras. Dynamic creation caused by `nu n` is handled by instantiating `n` with a finite number of static witnesses. Note that this transformation restricts the resources to be considered by the model-checker from an infinite to a finite set. Yet, this phase is correct, as shown in [5]. Some spurious traces might however be introduced by this transformation, so invalidating completeness. For instance, in some trace of the BPA associated to `nu n.U.(nu m.U')` the same witness might be chosen for both `n` and `m`. This would cause the model-checker to report a violation, so over-approximating the predicted behavior. The "Weak Until" phase described below will allow for recovering completeness.
3. **Framing the Policy.** The policy is duplicated in two layers, so that the first layer handles the transitions made by the usage when *outside* the policy framing, and the second handles them when *inside* the policy framing. Thanks to the regularization phase, this phase only needs to consider two layers.
4. **Instantiating the Policy.** The usage policies are instantiated, by non-deterministically assigning to each variable some known resource, including the witnesses generated in the "Conversion into BPA" phase.

5. **Weak Until.** Policies are adapted so to correctly handle traces where the same witness # happens to be generated twice, i.e. those having a double `new(#)` event. As noticed above, these traces do not correspond to any trace of the original usage, so they must never trigger a policy violation. In [5] this is proved enough to guarantee the completeness of model checking, while preserving its correctness.
6. **Model-Checking.** Finally, the traces of the BPA generated at phase 2 are matched against all the policies obtained after phase 5. Our model-checking algorithm decides whether there exists a policy violated by some BPA trace. Our model-checking procedure is complete, and it always terminates even though the BPA may have an infinite number of traces, possibly of infinite length (for instance, `mu h. c + h.h + a.h.b`).

The complexity of our model-checking algorithm is polynomial in the size of the usage and on the size of the policy. There is an exponential factor in the number of policy parameters, only. From a pragmatic point of view, we expect the number of parameters to be very small in practice. This exponential factor is mainly due to the policy instantiation step above, which is non-deterministic.

5 Conclusion

We have reported on four tools that have been developed within the SENSORIA project, providing practical support for the application of SENSORIA techniques to service-oriented systems, including the SENSORIA case studies. The tools tackle, in particular, the problems of model checking service-oriented systems, including multiparty systems, of checking conformance of orchestrations with respect to choreographic descriptions, and of ensuring that systems access resources according to specified policies.

The tools are at different stages of development. The CMC/UMC framework is more mature, and has been integrated in the SENSORIA Development Environment (see Chapter 6-5), thus allowing to use it in an integrated way inside the software development process. The other tools are less mature, and their integration is part of our future plans. However all the tools are publicly available: CMC and UMC at `http://fmt.isti.cnr.it/cmc` and `http://fmt.isti.cnr.it/umc` respectively, ChorSLMC at `http://ctp.di.fct.unl.pt/SLMC/` and LocUsT at `http://www.di.unipi.it/~zunino/software/locust`.

References

1. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (2006)
2. Bartoletti, M., Degano, P., Ferrari, G.L.: Enforcing secure service composition. In: Proc. of CSFW-18 2005, pp. 211–223. IEEE Computer Society, Los Alamitos (2005)
3. Bartoletti, M., Degano, P., Ferrari, G.L.: History-based access control with local policies. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 316–332. Springer, Heidelberg (2005)

4. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 32–47. Springer, Heidelberg (2007)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Model checking usage policies. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 19–35. Springer, Heidelberg (2009)
6. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6) (2009)
7. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL*. In: Proc. of LICS 1995, pp. 388–397. IEEE Computer Society, Los Alamitos (1995)
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
9. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
10. Caires, L.: Behavioral and spatial observations in a logic for the pi-calculus. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 72–89. Springer, Heidelberg (2004)
11. Caires, L., Cardelli, L.: A Spatial Logic for Concurrency (Part I). Information and Computation 186(2), 194–235 (2003)
12. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
13. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
14. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication–centred concurrent programming. Technical report, W3C (2006)
15. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 230–245. Springer, Heidelberg (2008)
16. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML state machines. In: Proc. of SERA 2004, pp. 331–338. ACIS (2004)
17. Gnesi, S., Mazzanti, F.: A model checking verification environment for UML statecharts. In: Proc. of XLIII Annual Italian Conference AICA. AICA (2005)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of POPL 2008, pp. 273–284. ACM, New York (2008)
19. Koch, N., Mayer, P., Heckel, R., Gönczy, L., Montangero, C.: UML for Service-Oriented Systems. SensoriaDeliverable 1.4a (September 2007)
20. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
21. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze (2007), `http://rap.dsi.unifi.it/cows`
22. Mayer, P., Schroeder, A., Koch, N.: Mdd4soa: Model-driven service orchestration. In: Proc. of EDOC 2008, pp. 203–212. IEEE Computer Society, Los Alamitos (2008)

23. Mazzanti, F.: UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo". CNR (2006), http://fmt.isti.cnr.it/WEBPAPER/UMC-UG33.pdf
24. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008)
25. ter Beek, M.H., Gnesi, S., Koch, N., Mazzanti, F.: Formal verification of an automotive scenario in service-oriented computing. In: Proc. of ICSE 2008, pp. 613–622. ACM Press, New York (2008)
26. ter Beek, M.H., Gnesi, S., Mazzanti, F., Moiso, C.: Formal modelling and verification of an asynchronous extension of soap. In: Proc. of ECOWS 2006, pp. 287–296. IEEE Computer Society, Los Alamitos (2006)
27. ter Beek, M.H., Mazzanti, F., Gnesi, S.: CMC-UMC: A framework for the verification of abstract service-oriented properties. In: Proc. of SAC 2009, pp. 2111–2117. ACM Press, New York (2009)
28. Unified Modeling Language, http://www.uml.org/
29. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service oriented computation. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
30. Vieira, H.T., Caires, L., Sousa, D.: Checking Services Conformance Based on Spatial Logic Model-Checking (revised). Technical Report TR-DI/FCT/UNL-04/2009, Departamento de Informática, Universidade Nova de Lisboa (2009)
31. Vieira, H.T., Caires, L., Viegas, R.: The Spatial Logic Model Checker, http://ctp.di.fct.unl.pt/SLMC/
32. Vieira, H.T., Caires, L., Viegas, R.: The Spatial Logic Model Checker User's Manual v1.0. Technical Report TR-DI/FCT/UNL-05/2005, Departamento de Informática, Universidade Nova de Lisboa (2005)
33. Web Services Choreography Working Group WCDL. Web Services Choreography Description Language: Primer (2006), http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/