CMC–UMC: A Framework for the Verification of Abstract Service-Oriented Properties[*]

Maurice H. ter Beek
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
m.terbeek@isti.cnr.it

Franco Mazzanti
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
f.mazzanti@isti.cnr.it

Stefania Gnesi
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
s.gnesi@isti.cnr.it

ABSTRACT

CMC and UMC are two prototypical instantiations of a common logical verification framework for the analysis of functional properties of service-oriented systems. The service-oriented SocL logic is used to describe the required system properties. Computational models of the system can be built either using the COWS specification language or designing the system as a collection of interacting UML state machines, and an on-the-fly model checker can be used to verify the satisfaction of the requirements and possibly to generate counterexamples or witnesses for them. An automotive case study is used to illustrate the overall framework.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Experimentation,Verification

Keywords

Service-Oriented Computing, Model checking

1. INTRODUCTION

The experimentation at ISTI–CNR with on-the-fly model-checking techniques has been initiated in 1999 with the design and development of a model checker [11] for the action-based ACTL logic extended with fixpoint operators. The computational model in that case was constituted by networks of automata built from terms of a process algebra derived from the regular value-passing CCS calculus.

At a later stage, the same model-checking algorithms have been applied to a different computational model, viz. UML statecharts. At that point the usefulness of not just an action-based logic, but rather an action- and state-based logic, had become evident: This allows one to express in a natural way not only properties of evolution steps (i.e. related to the executed actions) but also internal properties of states (e.g. the values of object attributes). This has resulted in the UMC prototype [12, 16] which has been used for the analysis of several case studies in the context of a number of research projects [2, 4, 5].

In 2007, the original on-the-fly model-checking approach has been applied to the process algebra COWS, developed in the context of SENSORIA. The IST-FET Integrated Project SENSORIA addresses problems of Service-Oriented Computing (SOC) by building, from first-principles, novel theories, methods and tools supporting the engineering of software systems for service-oriented overlay computers [20]. The results of SENSORIA include a comprehensive service ontology [6], modelling languages for service-oriented systems based on UML, and a number of process calculi for SOC, like COWS. COWS is a specification language specifically oriented to the design of service-oriented systems, and in that context the need became evident of a mechanism in the logic to express the correlation between dynamically generated values as appearing inside actions at different times. These values represent the correlation values which allow, e.g., to relate the responses of a service to their specific request, or to handle the concept of a session involving a long sequence of interactions among the interacting partners. This has led to the development of CMC as a new companion of the UMC prototype. At this point both CMC (v0.3) and UMC (v3.4) shared the same roots (and part of the actual code) but differed in the details of the supported logic and clearly in the underlying computational model.

The next evolution step has consisted of recognizing the need on the one hand of a logic which allows one to express and reason in a natural way about service-oriented properties, and on the other hand of a mechanism that allows one to hide all the computational-model-dependent details inside the on-the-fly model generator so that the logic (and model checker) could remain at an abstract computational-model-independent level. This led to development of the SocL logic and to the restructuring of CMC with the introduction of an explicit abstraction mechanism of the ground COWS-dependent model, an approach which has consequently been tested in [10] by applying it to the SENSORIA Automotive case study.

[*]The research carried out for this paper was partially funded by the EU project SENSORIA (IST-2005-016004) and by the MIUR–PRIN 2007 D-ASAP.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

Being satisfied with the preliminary results, our successive step has been to adopt the same SocL logic, and ground-computational-model abstraction approach also inside the UMC framework, achieving in this way a uniform logical verification framework, of which CMC v0.5 and UMC v3.6 are two instantiations, for the analysis of functional properties of service-oriented systems. In particular, this approach has allowed us to model our original Automotive case study using two different specification languages, and still reason about its properties and check the model's conformance to them in an abstract way. In this paper we illustrate in more detail this final result.

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 accepting requests, delivering corresponding responses, and, on-demand, cancelling requests. We are interested in expressing and verifying properties about these systems, like for example checking whether a service is

1. *available*: if it is capable to accept a request;
2. *responsive*: if it always guarantees a response to each accepted request;
3. *sequential*: if, after accepting a request, no other requests may be accepted before giving a response;
4. *one-shot*: if, after accepting a request, no other requests can be accepted.

This list contains just some of the common properties that express desirable attributes of services and service-oriented applications (see, e.g., [1, 6]). They however give a rough idea of the goal of our experimentation.

We are moreover interested in expressing and verifying orchestration properties involving relations among the sub-services invoked during the excution of one service session, like for example compensation properties.

2. THE SOCL LOGIC

To formalize properties like those expressed above, we use SocL, a variant of the logic UCTL [3] originally introduced to express properties of UML statecharts. We give only a short description of SocL, referring to [9, 10] for a more detailed explanation. UCTL and SocL have many commonalities: they share the same temporal logic operators, they are both state- and event-based branching-time logics, they are both interpreted on Doubly-Labelled Transition Systems (L^2TSs [8]) by exploiting the same on-the-fly model-checking engine. An L^2TS is a labelled transition system whose states are labelled by atomic propositions and whose transitions are labelled by sets of actions. The two logics mainly differ for the syntax and semantics of state-predicates and action-formulae, and for the fact that SocL also permits the specification of parametric formulae allowing to reason about dynamic data values exchanged among entities and hence on the correlations between interactions.

The syntax of SocL formulae is defined as follows:

$(state\ formulae)$

$$\phi ::= true \ \mid\ \pi \ \mid\ \neg\phi \ \mid\ \phi \wedge \phi' \ \mid\ E\Psi \ \mid\ A\Psi$$

$(path\ formulae)$

$$\Psi ::= X_\gamma \phi' \ \mid\ \phi\ {}_\chi U_\gamma\ \phi' \ \mid\ \phi\ {}_\chi W_\gamma\ \phi'$$

$\pi \in AP$ are abstract atomic propositions, A and E are *path quantifiers*, U and W are the doubly-indexed *until* and *weak until* operators drawn from those firstly introduced in [8] and subsequently elaborated in [18]. χ and γ are action formulas: γ may contain free variables which are then used as binders for the variables in ϕ', while χ cannot contain free variables. Finally, τ denotes the empty action formula.

The interpretation domain of SocL is L^2TSs in which transitions are labelled by sets of abstract observable actions, each one representing a relevant service interaction occurring during a single evolution step. Observable actions have the form $t(i, c)$, where t denotes the kind of an interaction (i.e. whether it is the acceptance of a request, a reply, and so on), i indicates the name of an interaction (i.e. the name of an operation exposed by a service) and c denotes a tuple of data values (correlation values) which identify a specific activation of a service operation (often denoting a *session*).

Examples *request(cardcharge, req1)*, *response(cardcharge, req1)*, and *cancel(cardcharge, req1)* correspond, respectively, to the initial request of the *cardcharge* interaction using the correlation data $req1$, and to a response, and to a failure notification inside the same session. The states of the L^2TS are instead labelled by a set of atomic propositions which have the form $p(i, c)$, where p denotes an abstract state predicate possibly parameterized by an interaction name and a correlation value (e.g. *accepting_request(cardcharge)*).

A state formula π is satisfied in a state S if at least one of the abstract propositions of S satisfies π. An action formula χ is satisfied by an L^2TS transition r if at least one of the abstract actions e of r satisfies χ. The action formula τ is satisfied by an L^2TS transition r if the set of abstract actions of r is empty. If a transition r satisfies an action formula χ, then it also defines some bindings for the free variables of χ which are used to instantiate the parametric subformula ϕ.

The next operator X says that in the next state of the path, reached by an action satisfying γ, the formula ϕ' holds. The intuitive meaning of the doubly-indexed until operator U on a path is that ϕ' holds at some future state of the path reached by a last action satisfying γ, while ϕ has to hold from the current state until that state is reached and all the actions executed in the meanwhile along the path either satisfy χ or τ. Finally, the weak until operator W holds on a path either if the corresponding strong until operator holds or if for all states of the path the formula ϕ holds and all the actions of the path either satisfy χ or τ.

Other useful operators can be derived, some of the most widely used are:

false stands for $\neg$ *true*;
$\phi \vee \phi'$ stands for $\neg(\neg\phi \wedge \neg\phi')$;
$< \gamma > \phi$ stands for $EX_\gamma\, \phi$;
$[\gamma]\, \phi$ stands for $\neg < \gamma > \neg\, \phi$;
$EF\phi$ stands for $E(true\ {}_{tt}U\phi)$;
$EF_\gamma\ true$ stands for $E(true\,{}_{tt}U_\gamma true)$;
$AF_\gamma\ true$ stands for $A(true\,{}_{tt}U_\gamma true)$;
$AG\,\phi$ stands for $\neg\, EF\, \neg\, \phi$;
$E(\phi_\chi\ U\ \phi')$ stands for $\phi' \vee (\phi \wedge [E(\phi_\chi U_{\chi\vee\tau}\phi')])$;
$A(\phi_\chi\ U\ \phi')$ stands for $\phi' \vee (\phi \wedge [A(\phi_\chi U_{\chi\vee\tau}\phi')])$;
$E(\phi_\chi\ W\ \phi')$ stands for $\phi' \vee (\phi \wedge [E(\phi_\chi W_{\chi\vee\tau}\phi')])$;
$A(\phi_\chi\ W\ \phi')$ stands for $\phi' \vee (\phi \wedge [A(\phi_\chi W_{\chi\vee\tau}\phi')])$.

Using this formal setting the previously cited informal properties can then be formally stated in the following way, where for a given correlation variable var, its binding occurrence will be denoted by $\underline{var}$ and all remaining occurrences, that are called *free*, will be denoted by var:

1. Available service:
   $AG(accepting\_request(i))$

2. Responsive service:
   $AG[request(i, \underline{var})]\, AF_{response(i,var) \vee fail(i,var)}\, true$

3. Sequential service:
   $AG[request(i, \underline{var})]$
   $A(\neg\, accepting\_request(i)\, _{tt}$
   $U_{response(i,var) \vee fail(i,var)}\, true)$

4. One-shot service:
   $AG[request(i, \underline{var})]\, AG\, \neg\, accepting\_request(i)$

To illustrate how system specific orchestation / compensation properties can be expressed and checked we first need to introduce a case study. In Section 4 we will present an automotive scenario studied in SENSORIA [13].

3. THE OVERALL STRUCTURE OF THE MODEL CHECKERS

Two different specification languages have been taken into consideration for the development of the model checkers for the SocL logic. The first is represented by the process algebra COWS [14, 15] (one of the SENSORIA core languages) and the second is based on UML statecharts [19].

The approach adopted by the tools to generate the L^2TS and the verification of the formulae is the "on-the-fly" approach, meaning that the L^2TS corresponding to the model is generated "on-demand", following the needs of the logical verification engine. Given a state of an L^2TS , the validity of a SocL formula on that state is evaluated by analyzing the transitions allowed in that state, and by analyzing the validity of the necessary subformulae possibly in some of the necessary next reachable states, all this in a recursive way. Indeed each tool consists of two separate, but interacting, engines: an L^2TS generator engine and a logical verification engine. The L^2TS generator engine is again structured in two logical components: a ground evolutions generator, strictly based on the operational semantics of the language, and an abstraction mechanism which allows to associate abstract observable events to system evolutions and abstract atomic propositions to the system states. The verification engine is the component which actually tries to evaluate a logic formula following the on-the-fly approach, and is described in more detail in [3, 9].

The L^2TS generator engine maintains an archive of already generated system states in order to avoid unnecesary duplications in the computation of the possibile evolutions of states. The logical verification engine maintains an archive of logical computation fragments; this is not only useful to avoid unnecessary duplications in the evalution of subformulae, but also necessary to deal with the recursion in the evaluation of a formula arising from the presence of loops in the models. These computation fragments have the form $\langle \texttt{subformula}, \texttt{State}\rangle \rightarrow \texttt{ComputationProgress}$ and associate each evalution of a subformula in some state to its computation status which in the end will be a value `True` or `False`.

```
interface L2TS GENERATOR is
  – EVOLUTIONS ENGINE
  Initial_State: → State
  Get_Evolutions: State → set of Evolution
  Evolution_Source: Evolution → State
  Evolution_Target: Evolution → State
  Evolution_Actions: Evolution → Ground_Actions
  – ABSTRACTIONS
  Abstract_Predicates:
    State → Abstract_Propositions
  Abstract_Events:
    Evolution → Abstract_Events
end

interface ON-THE-FLY EVALUATOR is
  Evaluate:
    (State_Formula,State) → Computation_Result
end
```

Table 1: Interfaces of L^2TS generator / evaluator.

From an abstract point of view the engines can be seen as an implementation of the interfaces shown in Table 1.

The big advantage of the on-the-fly approach to model checking is that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result [7]. Another advantage with respect to other very efficient symbolic approaches (e.g. those BDD based) is that it is much easier to generate a clear and detailed explanation of the returned results (i.e. a *counterexample* or a *witness*).

4. A SAMPLE CASE STUDY

A vehicle that leaves the assembly line today is equipped with a multitude of sensors and actuators that provide the driver with services that assist in conducting the vehicle more safely, such as vehicle stabilization systems. Driver assistance systems kick in automatically when the vehicle context renders it necessary, and more and more context is taken into account (e.g. road conditions, vehicle condition, driver condition, weather conditions, traffic conditions, and so on). Naturally, these units have to interplay in different settings and, consequently, there is great potential for SOC in automotive systems (think for example of service orchestration and service composition). In addition, due to the advances in mobile technology, telephone and Internet access in vehicles is possible, giving rise to a variety of new services for the automotive domain, such as handling based on information provided by other vehicles passing nearby or by location-based services in the surrounding. For example, assume that while a driver is on the road with her/his car, the vehicle's diagnostic system reports a low oil level. This triggers the in-vehicle diagnostic system to report a problem with the pressure of the cylinder heads, which results in the car being no longer driveable, and to send this diagnostic data as well as the vehicle's GPS coordinates to the repair server. Based on the driver's preferences, the service discovery system identifies and selects an appropriate set of services (garage, tow truck, and rental car) in the area. When the driver makes an appointment with the garage, the results of the in-vehicle diagnosis are automatically sent along, allowing the garage to identify the spare parts needed to repair the car. Similarly, when the driver orders a tow truck and a rental car, the vehicle's GPS coordinates are sent along. Obviously, the driver is required to deposit a security payment before s/he is able to order any service. Finally, each of these services can be denied or cancelled, causing an appropriate compensation activity.

With respect to the above scenario, some of the properties we might want to check are for example:

1. After the garage has been booked, if the tow service is not available then the garage is revoked;
2. After a successful credit card deposit, in the case that no services are found, the deposit is revoked;
3. If a credit card deposit is denied by the bank, then no services will be booked.

We will see later how properties like these are formalized.

5. THE COWS SPECIFICATION

As a first specification language for our automotive case study we use COWS (*Calculus for Orchestration of Web Services* [15]), a recently proposed process calculus for specifying and combining services, while modelling their dynamic behaviour. We refer the interested reader to [14, 15] for the details of COWS, as well for many examples illustrating COWS' peculiarities and expressiveness, and for comparisons with other process-based and orchestration formalisms.

In the case of COWS the formal operational semantics of the language defined by a set of term rewriting rules defines a ground L^{2}TS modelling all the possible system evolutions. The labels appearing in the edges of this ground L^{2}TS represent the input-output actions between parallel terms occurring during a rewriting step and have the form:

$p_{id1} \bullet o_{id2}?\langle x_{id3}, x_{id4}\rangle$,
$p_{id1} \bullet o_{id2}!\langle v1, v2\rangle$,

where pair $p_{id1}.o_{id2}$ represents a communication endpoint constituted by a partner and operation name, operators ? and ! indicate the action modality, and $\langle ... \rangle$ represents the action parameters. The labels on the nodes of this ground L^{2}TS have the same form as the labels on the edges, except that they do not represent the actions actually executed during a step, but just the potentially active actions of some term in some system state (they will also appear as labels in an outgoing edge from that node, if a matching pair of input/output actions is present in the node).

This ground L^{2}TS is then abstracted by applying a set of abstraction rules which allow one to associate abstract observable events to the actual ground events of the computational model. The above scenario (fully described in [9]) can be modelled by the following top level COWS term:

$$\begin{array}{l}[p_{car}](SensorsMonitor \mid \mathit{Orchestrator} \mid \mathit{GpsSystem} \mid \\ \quad \mathit{LocalDiscovery} \mid \mathit{Reasoner})\\ \mid \mathit{Bank}\\ \mid \mathit{Garage_1} \mid \mathit{Garage_2} \mid \mathit{TowTruck_1} \mid \mathit{TowTruck_2}\end{array}$$

where, e.g., the garage behaviour is described by the term:

$$\begin{array}{l} *\,[x_{cust}, x_{loc}]\\ \quad p_{garage\_i} \bullet o_{order}?\langle x_{cust}, x_{loc}\rangle.\\ \qquad x_{cust} \bullet o_{garageFail}!\langle \mathit{failData}\rangle\\ \quad + \; p_{garage\_i} \bullet o_{ordergar}?\langle x_{cust}, x_{loc}\rangle.\\ \qquad (x_{cust} \bullet o_{garageOk}!\langle \mathit{garageData}\rangle\\ \qquad \mid p_{garage\_i} \bullet o_{cancel}?\langle x_{cust}\rangle)\end{array}$$

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

While space prohibits the presentation of the full ground level COWS specification, we show instead some of the abstraction rules used to define the abstract L^{2}TS, which define the abstract events we are interested to observe:

$$\begin{array}{ll}
\mathit{State}: & o_{engfail}?\langle\rangle \rightarrow \mathit{accepting\_request}(\mathit{road\_assistence})\\
\mathit{Action}: & o_{engfail}!\langle\rangle \rightarrow \mathit{request}(\mathit{road\_assistence})\\
\ldots & \\
\mathit{State}: & o_{charge}?\langle *\rangle \rightarrow \mathit{accepting\_request}(\mathit{bankcharge})\\
\mathit{Action}: & o_{charge}!\langle \$1\rangle \rightarrow \mathit{request}(\mathit{bankcharge}, \$1)\\
\mathit{Action}: & \$1 \bullet o_{chargeFail}!\langle *\rangle \rightarrow \mathit{fail}(\mathit{bankcharge}, \$1)\\
\mathit{Action}: & \$1 \bullet o_{chargeOk}!\langle *\rangle \rightarrow \mathit{response}(\mathit{bankcharge}, \$1)\\
\mathit{State}: & o_{bankrevoke}?\langle \$1\rangle \rightarrow\\
& \qquad \mathit{accepting\_revoke}(\mathit{bankcharge}, \$1)\\
\mathit{Action}: & o_{bankrevoke}!\langle \$1\rangle \rightarrow \mathit{revoke}(\mathit{bankcharge}, \$1)\\
\ldots & \\
\mathit{Action}: & o_{ordergar}!\langle \$1\rangle \rightarrow \mathit{request}(\mathit{garage}, \$1)\\
\mathit{Action}: & \$1 \bullet o_{garageFail}! \rightarrow \mathit{fail}(\mathit{garage}, \$1)\\
\mathit{Action}: & \$1 \bullet o_{garageOk}!\langle \$1\rangle \rightarrow \mathit{response}(\mathit{garage}, \$1)\\
\mathit{Action}: & o_{cancel}!\langle \$1\rangle \rightarrow \mathit{revoke}(\mathit{garage}, \$1)\\
\ldots & \\
\mathit{Action}: & \$1 \bullet o_{rentalcarOk}! \rightarrow \mathit{response}(\mathit{rentalcar}, \$1)\\
\ldots & \\
\mathit{Action}: & \$1 \bullet o_{towtruckOk}! \rightarrow \mathit{response}(\mathit{towtruck}, \$1)\\
\mathit{Action}: & \$1 \bullet o_{towtruckFail}! \rightarrow \mathit{fail}(\mathit{towtruck}, \$1)\\
\ldots &
\end{array}$$

These rules associate to the ground communication actions of the model (described in the left side of each rule) the L^{2}TS labels (described in the right side of each rule) with respect to which the model checker will verify the SocL formulae.

6. THE UML STATECHARTS-BASED SPECIFICATION

State machines are another widely used specification language to describe, often in an object-oriented way, the components of a concurrent system. UML statecharts are quite a standard formalism for designing state machines, and many tools exist to support this notation. In UMC a statechart (modelling a state machine) is associated to the notion of "class", while the system configuration (deployment) is defined by a set of objects (active instances of classes). A class defines the dynamic behaviour of its objects (through a statechart), toghether with their interfaces towards the other components (i.e. the set of signals and operations they can receive) and the structure of the local status of the object instances (i.e. the local object attributes).

In this case the ground L^{2}TS of the system is defined by the informal operational semantics of a UML system constituted by a set of concurrent communicating state machines, where the parallelism among objects (according to the UMC choices) is modelled through interleaving, the communications are modelled as immediate and failure-free, and the events queue are plain FIFO queues. The labels of the edges in this ground L^{2}TS are essentially constituted by the events of sending / dispatching of signal or operation events among state machines, and have the form:

sourceobj : *targetobj.eventname*(*params*).

Other special cases of ground labels appearing in the L^2TS edges are:

$sourceobj : accept(eventname, params)$ and
$sourceobj : lostevent(eventname)$,

which are used to denote, respectively, the successfull dispatching of an event from the object events queue and its use for triggering a transition and the removal and discarding of an event from the queue because it does not trigger any transition. The ground labels associated to the nodes of the ground L^2TS instead have the form:

$inState(objname.statename)$ or
$objname.attribname = value$

and denote, respectively, the fact that a certain state of an object is currently active and that a certain attribute of an object has a certain value.

With respect to our case study, the UMC specification has the following structure:

Class Car **is** ... **end** Car;
Class Garage **is** ... **end** Garage;
Class TowTruck **is** ... **end** TowTruck;
Class Rental **is** ... **end** Rental;
Objects:
car1: Car;
bank1: Bank;
garage1, garage2: Garage;
towtruck1,towtruck2: TowTruck;
rental1,rental2: Rental;

where Car objects have an internal parallel structure, constituted by the various components:

Class Car **is**
...
State Top =
Engine / Orchestrator / LocalDiscovery /
GPS / Reasoner / VehicleCommunicationGateway
...
end Car;

Again, we omit the details of the structure of the UML description of the system, referring to [5] for a more detailed presentation, while we show how the ground L^2TS resulting from the UML computational model is abstracted by applying a set of abstraction rules which allow to associate abstract observable events to the actual ground events of the model. An example of such rules is shown below:

$State : inState(car1.Orchestrator.o1) \rightarrow$
$\quad accepting\_request(road\_assistance)$
$Action :\ accept(engineFailure) \rightarrow$
$\quad request(road\_assistence)$
$\ldots$
$State : inState(bank1.s1) \rightarrow$
$\quad accepting\_request(bankcharge)$
$Action :\ \$1 : requestCardCharge \rightarrow$
$\quad request(bankcharge, \$1)$
$Action :\ \$1.chargeResponseFail \rightarrow\ fail(bankcharge, \$1)$
$Action :\ \$1.chargeResponseOK \rightarrow$
$\quad response(bankcharge, \$1)$
$State : inState(bank1.s1) \wedge\ bank1.cu = \$1 \rightarrow$
$\quad accepting\_revoke(bankcharge, \$1)$
$Action :\ \$1.revokeCardCharge \rightarrow$
$\quad revoke(bankcharge, \$1)$
$\ldots$
$Action :\ \$1 : requestGarage \rightarrow\ request(garage, \$1)$
$Action :\ \$1.garageResponseFail \rightarrow\ fail(garage, \$1)$
$Action :\ \$1.garageResponseOK \rightarrow\ response(garage, \$1)$
$Action :\ \$1 : revokeGarage \rightarrow\ revoke(garage, \$1)$
$\ldots$
$Action :\ \$1.rentResponseOK \rightarrow\ response(rentalcar, \$1)$
$\ldots$
$Action :\ \$1.towResponseOK \rightarrow\ response(towtruck, \$1)$
$Action :\ \$1.towResponseFail \rightarrow\ fail(towtruck, \$1)$
$\ldots$

Notice that the abstract observable events constituting the abstract L^2TS (as they appear on the right side of the abstractions rules) are the same abstract events used in the case of the COWS specification. Indeed the abstract properties we want to verify are independent from the implementation details of a specific computation model, and the abstraction mechanism which is part of the CMC / UMC model generation engines allows us to raise the reasoning / verification level from that of the ground computational model to that of a common service-oriented, computational-model-independent abstract level.

7. ON ROAD ASSISTANCE PROPERTIES

With respect to our on road assistance scenario, SocL allows us to express several relevant abstract properties for the various services composing the system, and CMC / UMC allow to verify them with respect to our two different computational models. Some of the properties which hold for our two models are shown below:

- The road assistance service is always available until requested:
$A(accepting\_request(road\_assistance)_{tt}$
$\quad U_{request(road\_assistance)}\ true)$
- Road assistance is a *one-shot* service (it cannot be triggered twice), and it is always activated in this scenario:
$AF_{request(road\_assistance)}$
$\quad AG\ \neg\ accepting\_request(road\_assistance)$
- The service Bank is always *available*:
$AG\, accepting\_request(bankcharge)$
- The service Bank after accepting a request always provides a unique Ok or Fail response (*single-response*):
$AG\ [request(bankcharge, \underline{customer})]$
$\quad AF_{response(bankcharge,customer) \vee fail(bankcharge,customer)}$
$\quad\quad \neg EF_{response(bankcharge,customer) \vee fail(bankcharge,customer)}\ true$
- After a successful response to a creditcard charge request the Bank accepts revoke requests for the succeeded transaction (strongly revokable):
$AG\ [response(bankcharge, \underline{customer})]$
$\quad A(accepting\_revoke(bankcharge, customer)_{tt}$
$\quad\quad W_{revoke(bankcharge,customer)}\, true)$

By putting into relation several events belonging to different subinteractions we can check also orchestration / compensation properties on the internal architecture of a service design. Some examples are shown below:

- After the Garage has been booked, if the Tow service is not available then the Garage is revoked:

  $AG\,[response(garage, \underline{customer})]$
  $\quad AF([fail(towtruck, customer]$
  $\qquad AF_{revoke(garage,customer)}\ true)$

- After a successful deposit, either some services are booked, or the deposit is revoked:

  $AG\,[response(bankcharge, \underline{customer})]$
  $\quad AF_{\ revoke(bankcharge,customer)\vee response(towtruck,customer)\vee response(rentalcar,customer)}\ true$

- If the deposit is denied by the Bank, then no services will be booked:

  $AG\,[fail(bankcharge, \underline{customer})]$
  $\quad \neg\ EF_{\ response(garage,customer)\vee response(rentalcar,customer)}\ true$

The models used in this case study are quite simple (the COWS model has 202 states, while the UML model has 501 states). The big difference is due to the fact that while in COWS communications are modelled as synchronous pairs of send-receive actions, in UML they are modelled as two separate actions (sending and storing into a queue, and removing and dispatching from the queue). The model can easily be made more complex by adding elements to the system configurations. For example, by adding a second car (allowing the checking of potential interferences among concurrent service invocations) the COWS model grows to more than $40,000$ states. We are still in the process of checking the system under these more complex configurations.

Another feature provided by of our formal framework is the possibility to compute minimized versions of models which preserve the equivalence with original systems with respect to the set of full traces of observable events generated by the systems. This is achieved by translating the L^2TS models into plain LTSs and then use standard minimization tools to compute minimal versions. This would allow to compare the two models with respect to their possible abstract traces.

8. PROTOTYPES DEPLOYMENT

The cores of both prototypes are constituted by command-line-oriented versions of the model checkers (i.e. the native CMC / UMC executables) which are stand-alone programs written in Ada and compiled for the Windows / Linux / Solaris / MacOSX platforms. These core executables once wrapped with a set of CGI scripts handled by a web server, become accessible through the web as online web applications [17]. In this way a graphical HTML-oriented GUI can easily be built, and the integration with other tools like LTS minimizers can easily be achieved. Moreover, once wrapped into an appropriate set of Java classes, the two native core applications can be transformed into plugins for the Eclipse environment, and into plugins for the SENSORIA SDE Environment. Finally, the Eclipse plugins can optionally be compiled as stand-alone applications, thus enriching the original command-line-oriented core tools with a graphical GUI even without the need of installing a full Eclipse environment. The development of CMC / UMC is still in progress even if the tools are being routinely used for academic and experimental purposes. Until now, the focus of the development has been on the design of qualitative features one would desire for a verification tool, thus experimenting with various logics, system modelling languages, and user interfaces. It is currently outside our purposes to transform the prototypes into real-world development tools. For this kind of evolution we should take into consideration issues like strong code optimizations, scalability over massively large systems, exhaustive testing, and validation issues, all issues outside our short-term plans.

9. CONCLUSIONS

The research being conducted at ISTI and presented in this paper involves two main aspects. From one side we are experimenting with logics allowing to express (as far as possible) in a natural and implementation-independent way properties of service-oriented systems. From another side we are experimenting an on-the-fly architecture for a verification environment which allows us to keep separate model exploration aspects from formula verification aspects, by introducing an abstraction layer over the ground implementation-dependent L^2TS modelling the system evolutions. This allows us to easily instantiate our framework with respect to two very different specification approaches (the process algebra COWS and UML statecharts), achieving the result of describing the same system at different, but related, levels of abstractions. This activity of comparison among models described in different ways has just been started and is precisely our ongoing line of research.

10. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2004.

[2] L.F. Andrade *et alii*. AGILE: Software Architectures for Mobility In *WADT'02*, volume 2755 of *LNCS*, pages 1–33. Springer, 2003.

[3] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *FMICS'07*, volume 4916 of *LNCS*, pages 133–148. Springer, 2008.

[4] M.H. ter Beek, S. Gnesi, F. Mazzanti, and C. Moiso. Formal Modelling and Verification of an Asynchronous Extension of SOAP. In *ECOWS'07*, pages 287–296. IEEE Computer Society, 2006.

[5] M.H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti. Formal verification of an automotive scenario in Service-Oriented Computing. In *ICSE'08*, pages 613–622. ACM Press, 2008.

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

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, 1999.

[8] R. De Nicola and F.W. Vaandrager. Three logics for branching bisimulation. *JACM*, 42(2):458–487, 1995.

[9] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. Tech. Report, Univ. of Florence, 2007. `http://rap.dsi.unifi.it/cows`.

[10] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach

for verifying COWS specifications. In *FASE'08*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[11] S. Gnesi and F. Mazzanti. On the Fly Verification of Networks of Automata. In *PDPTA'99* (Special session on Current limits to automated verification for distributed systems). CSREA Press, 1999.

[12] S. Gnesi and F. Mazzanti. On the fly model checking of communicating UML State Machines In *SERA'04*, pages 331–338, 2004.

[13] N. Koch and D. Berndl. Requirements modelling and analysis of selected scenarios: Automotive case study. SENSORIA deliverable D8.2a, 2006.

[14] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Tech. Report, Univ. of Florence, 2007. `http://rap.dsi.unifi.it/cows`.

[15] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[16] F. Mazzanti. UMC User Guide (Version 3.3). ISTI Tech. Report 2006-TR-33, 2006.

[17] CMC–UMC online. `http://fmt.isti.cnr.it/cmc` `http://fmt.isti.cnr.it/umc`

[18] R. Meolic, T. Kapus, and Z. Brezocnik. ACTLW: An action-based computation tree logic with unless operator. *Information Sciences*, 178(6):1542–1557, 2008.

[19] Object Management Group. *UML 2.0 Superstructure Specification*, ptc/03-08-02, OMG, 2003.

[20] EU project SENSORIA (IST-2005-016004). `http://www.sensoria-ist.eu/`.