Formal Verification of an Automotive Scenario in Service-Oriented Computing

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

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

Nora Koch[*]
Cirquent GmbH
Zamdorfer Straße 120
81677 München, Germany
nora.koch@cirquent.de

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

ABSTRACT

We report on the successful application of academic experience with formal modelling and verification techniques to an automotive scenario from the service-oriented computing domain. The aim of this industrial case study is to verify *a priori*, thus before implementation, certain design issues. The specific scenario is a simplified version of one of possible new services for car drivers to be provided by the in-vehicle computers.

Categories and Subject Descriptors

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

General Terms

Experimentation,Verification

Keywords

Automotive systems, Service-Oriented Computing, Model checking

1. INTRODUCTION

When a new vehicle is developed in an automotive company, 80% of the innovation cost is due to software systems. The total cost of a finished vehicle consists for 25% of software costs and the vehicle contains more than 70 Electronic Control Units, which each require their own specific software [29]. Naturally, these units have to interplay in different settings and, consequently, there is great potential for Service-Oriented Computing (SOC) in automotive systems (think, *e.g.* of orchestration of services for driver assistance and of infotainment). In fact, there is a lot of activity in this direction.[1] Moreover, the highly advanced mobile technology available nowadays allows car drivers telephone and Internet access within their vehicles, giving rise to a variety of new services for the automotive domain. In addition, the main reason for introducing service-oriented software engineering approaches is the complex and flexible networking of a continuously increasing number of car functionalities.

In this paper, we consider a scenario that offers a car driver one such a service, *viz.* on road assistance to contact a garage, a tow truck and a rental car when stranded with a malfunctioning vehicle. We report on the lessons that we have learned from applying formal modelling and verification techniques during the requirements analysis phase of this on road assistance scenario. The goal of this case study is to verify *a priori*, *i.e.* before any implementation, certain design issues related to functional requirements. To this aim, the requirements model of this scenario [22]—a high-level UML [35] specification that makes use of domain-specific extensions [23] like stereotypes to deal with compensations—is formally defined as a set of communicating UML state machines. The on-the-fly model checker UMC [34] is subsequently used to verify a set of correctness properties formalized in the action- and state-based temporal logic UCTL [3].

Model checking deals with the automatic analysis of correctness properties of system designs [6]. Such verifications are exhaustive, *i.e.* all possible input combinations and states are taken into account, and a counterexample is usually generated in case a certain property does not hold. Correctness properties reflect typical (un)desired behaviour of the system under scrutiny. Model checking allows the evaluation of design alternatives before implementation and validation—and not just afterwards. In this way, design errors—which

[*]Nora Koch is also affiliated with Ludwig-Maximilians-Universität, Oettingenstr. 67, 80538 München, Germany. `kochn@pst.ifi.lmu.de`.

[1]"Mitsubishi Electric Corporation, IBM and ILS Technology LLC [...] are delivering a service oriented architecture (SOA) solution that is specifically designed for the automotive manufacturing industry." (Press release, 31 Jan. 2008)

constitute up to 40% of software errors and are among the most expensive ones to resolve if discovered after implementation [24]—can be detected in the design phase, leading to considerable reductions in cost and to improved quality.

We begin this paper with brief descriptions of the methods and tools used for our case study, which is itself introduced in the subsequent section. We then discuss the formal model of our case study and the properties we verified, followed by the lessons we learned from this case study. Finally, we conclude with a discussion of future work.

2. METHODS AND TOOLS

In this section, we briefly describe the methods and tools that we use for our case study, *viz.* a UML Profile for SOC, the temporal logic UCTL and the model checker UMC.

2.1 A UML Profile for SOC

We assume some familiarity with the basics of the Unified Modelling Language UML [35], as it is the *de facto* industrial standard for modelling and documenting software systems. The reader is referred to [37] and the references therein for details on SOC.

In order to provide a domain-specific language in the context of SOC, the specification language UML 2.0 has been extended by using the available extension mechanisms. The result is a so-called UML Profile. The extension consists of a set of stereotypes and constraints specified in the Object Constraint Language OCL [35]. Stereotypes enrich models with service-oriented concepts, including structural and behavioural aspects of SOC as well as non-functional notions. Constraints allow for a more precise semantics of the newly introduced model elements.

The structure of a service-oriented architecture can be visualized by UML deployment and structure diagrams. Deployment diagrams are used to represent the "usually nested" nodes of the architecture, *i.e.* hardware devices and the software execution environment. Nodes are connected through communication paths that show the type of communication: permanent, temporary or on the fly. The last type characterizes service-oriented systems that require service discovery and provide loose binding of software. The stereotypes ≪permanent≫, ≪temporary≫ and ≪on the fly≫ are defined for these three types of connections between nodes.

Structure diagrams, on the other hand, show the interplay of components, their ports and their interfaces. The UML Profile for SOC introduces services implemented as ports of components. Each service may contain a required and a provided interface. In turn, interfaces contain one or more operations, which may contain an arbitrary number of parameters. A service has a service description and a service provider, and may have one or more service requesters. These concepts, and the relationships among them, are represented by a metamodel [23], which provides the basis for the definition of the UML Profile. For each class of the metamodel, a stereotype is defined and relationships are expressed by constraints. The stereotypes that are used to build structure diagrams of service-oriented systems are as follows: ≪service≫, ≪service interface≫ and ≪service description≫. An example of a structure diagram is shown in Section 4.

Behavioural aspects in SOC comprise the implementation of composite components. Most of the behaviour of such components results from the orchestration of simpler services (see *e.g.* the UML activity diagram in Figure 1). The focus is on service interactions, long-running transactions, and their compensation and exception handling. To model orchestration, we use a UML activity diagram containing service-aware elements, such as ≪scope≫, ≪send≫, ≪receive≫, and ≪compensate≫ actions. A scope is a structured activity that groups actions, which may have associated compensation and exception handlers. Scopes and the corresponding handlers are linked by specific compensation and exception edges (the stereotypes ≪compensation edge≫ and ≪exception edge≫). Scopes include stereotyped actions like send, receive, send and receive, compensate and compensate all. The stereotype ≪send≫ is used to model the sending of messages; ≪received≫ models the reception of a message blocking until the message is received; ≪send and receive≫ is a shorthand for a sequential order of a send action and a receive action. Container for data to be send or received are modelled by ≪send pin≫s and ≪receive pin≫s. Long-running transactions, like those provided by services, require the management of compensation. Therefore, the UML Profile contains ≪compensate≫ and ≪compensate all≫ actions. The former triggers the execution of the compensation defined for a scope or activity, the latter for the current scope. Compensation is called on all subscopes in the reverse order of their completion.

For details about the UML extension for service-oriented systems, the reader is referred to [23].

2.2 The Temporal Logic UCTL

To use the full potential of specification languages that—like UML—allow action as well as state changes to be modelled, one needs associated verification techniques that allow the validation of behavioural properties over one's model. Therefore, various temporal logics that allow one to express both action-based and state-based properties have been introduced recently [3, 5, 10, 18, 19, 28]. The advantage of all these logics lies in the ease of expressiveness of properties that in pure action-based or pure state-based logics can be quite cumbersome to write down. Moreover, their use often leads to a reduced state space, smaller memory needs and less time spent for verification. Obviously, the effective gain depends—as always—on the specific system under scrutiny.

In this paper, we will use the action- and state-based temporal logic UCTL [3].[2] UCTL includes both the branching-time action-based logic ACTL [8] and the branching-time state-based logic CTL [6]. Its syntax allows one to specify the properties that a state should satisfy and to combine these basic predicates with advanced temporal operators dealing with the actions performed:

$$\begin{array}{lll} \phi & ::= & \mathit{true} \mid p \mid \phi \wedge \phi' \mid \neg\phi \mid E\pi \mid A\pi \\ \pi & ::= & X_\chi\phi \mid \phi\,{}_\chi U\,\phi' \mid \phi\,{}_\chi U_{\chi'}\,\phi' \mid \phi\,{}_\chi W\,\phi' \mid \phi\,{}_\chi W_{\chi'}\,\phi' \end{array}$$

Predicates are ranged over by p, *state formulae* are ranged over by ϕ, *path formulae* are ranged over by π and *actions* are ranged over by χ. E and A are the *path quantifiers* "exists" and "for all", resp., while X, U and W are the indexed "next", "until" and "weak until" *temporal operators*, resp.

In UCTL, the "weak until" cannot be derived from the "until" because disjunction or conjunction of path formulae is not expressible according to the UCTL syntax. The same holds for any pure branching-time temporal logic [27].

[2]In [2, 13, 14], a less restricted logic (μ-UCTL) was defined and used in combination with previous versions of UMC.

Starting from these basic UCTL operators, it is straightforward to derive the standard logical operators $\vee$ and $\Rightarrow$, the well-known temporal logical operators EF ("possibly"), AF ("eventually") and AG ("always") and the diamond and box modalities $<>$ ("possibly") and $[]$ ("necessarily"), resp., of the Hennessy-Milner logic [16].

The semantic domain of UCTL is a doubly labelled transition system (L^2TS for short) [9]. An L^2TS (a.k.a. a Kripke transition system [28]) is a labelled transition system whose states are labelled by atomic propositions and whose transitions are labelled by sets of actions.

2.3 The Model Checker UMC

We have developed an on-the-fly model checker for UCTL, called UMC [26], which allows the efficient verification of UCTL formulae (*i.e.* specifying action- and/or state-based properties) over a set of communicating UML state machines [35]. The possible system evolutions are formally represented as an L^2TS, whose states represent the various system configurations and whose transitions represent the possible evolutions of a system configuration. More concretely, the states of this L^2TS are labelled with the observed structural properties of the system configurations (like the active substates of objects, the values of object attributes, *etc.*), while its transitions are labelled with the observed properties of the system evolutions (like which is the evolving object, which are the executed acions, *etc.*).

The big advantage of an 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 [4, 11]. The basic idea underlying UMC is that, given a state of an L^2TS, the validity of a UCTL formula on that state can be evaluated by analyzing the transitions allowed in that state and the validity of a certain subformula in only some of the next reachable states, all this in a recursive way. The current version of UMC uses an on-the-fly model-checking algorithm which has a linear complexity.

Another interesting feature offered by UMC is the possibility to select a desired subset of system events or object attributes, and to show the minimized graph of all the possible system evolutions (traces) in which only the relevant labels are shown. This allows one to obtain abstract slices of the system behaviour, for which only certain kinds of interactions are considered. These abstract slices are very useful for achieving confidence in the overall correctness of the design. Since abstracted full-trace minimization of an L^2TS requires a full traversal of this L^2TS, and moreover has a high complexity, this functionality is—unfortunately—only possible for finite and reasonably sized systems.

The current UMC prototype can be experimented via a web interface [34].

3. AUTOMOTIVE 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, *etc.*). 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. Some of these scenarios, like the one described in this section, are used to validate the engineering approaches developed in the EU project SENSORIA [31].

3.1 The On Road Assistance Scenario

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 being able to order any service. Finally, each service can be denied or cancelled, causing an appropriate compensation activity.

3.2 A UML Specification

In [22], the automotive architecture as defined within SENSORIA is described and the requirements model of the on road assistance scenario is specified in UML 2.0 using a UML Profile [23]. This scenario consists of the components:[3]

- Engine: causes low oil level alert
- Discovery engine: discovers services needed
- Reasoner: selects best services
- Orchestrator: composes services to achieve goal
- Driver: calls garage, tow truck, rental car and bank
- GPS: sends vehicle's current coordinates to services
- Garage: receives diagnostic data about vehicle
- Tow truck: receives GPS coordinates of vehicle
- Rental car: receives GPS coordinates of driver
- Bank: receives deposit from driver

Since the workflow describing the service orchestration is one of the most interesting aspects of a service-oriented system, we depict it in Figure 1 as a UML 2.0 activity diagram.

The orchestrator is triggered by an engine failure (in our case due to a low oil level) and consequently contacts the other components to compose the various services to reach its goal (in our case sending the driver a rental car and a tow truck to tow the stranded vehicle to a garage). Note that we use stereotyped UML 2.0 actions to indicate the type of interactions (≪send≫, ≪receive≫ and ≪sendAndReceive≫) as well as to model compensations of long-running transactions (≪compensate≫ and ≪compensate all≫). The actions match operations of required and provided interfaces of the services, defined as ports of UML 2.0 components.

[3]Some components of [22] are renamed and (de)composed.

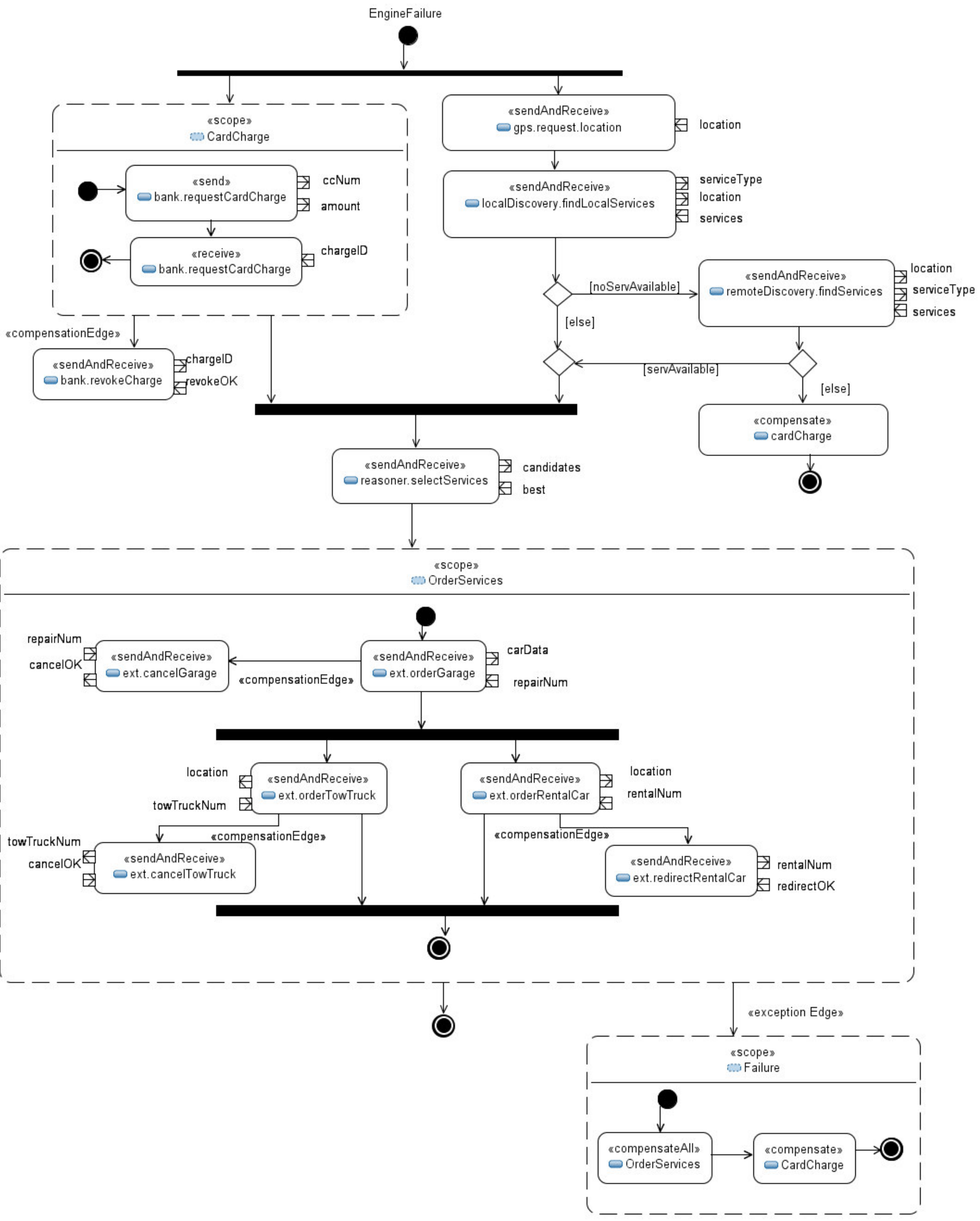
EngineFailure
«scope» CardCharge
«send» bank.requestCardCharge
ccNum
amount
«receive» bank.requestCardCharge
chargeID
«compensationEdge»
«sendAndReceive» bank.revokeCharge
chargeID
revokeOK
«sendAndReceive» gps.request.location
location
«sendAndReceive» localDiscovery.findLocalServices
serviceType
location
services
[noServAvailable]
«sendAndReceive» remoteDiscovery.findServices
location
serviceType
services
[else]
[servAvailable]
[else]
«compensate» cardCharge
«sendAndReceive» reasoner.selectServices
candidates
best
«scope» OrderServices
repairNum
cancelOK
«sendAndReceive» ext.cancelGarage
«compensationEdge»
«sendAndReceive» ext.orderGarage
carData
repairNum
location
towTruckNum
«sendAndReceive» ext.orderTowTruck
«sendAndReceive» ext.orderRentalCar
location
rentalNum
towTruckNum
cancelOK
«compensationEdge»
«sendAndReceive» ext.cancelTowTruck
«compensationEdge»
«sendAndReceive» ext.redirectRentalCar
rentalNum
redirectOK
«exception Edge»
«scope» Failure
«compensateAll» OrderServices
«compensate» CardCharge

Figure 1: Orchestration of services.

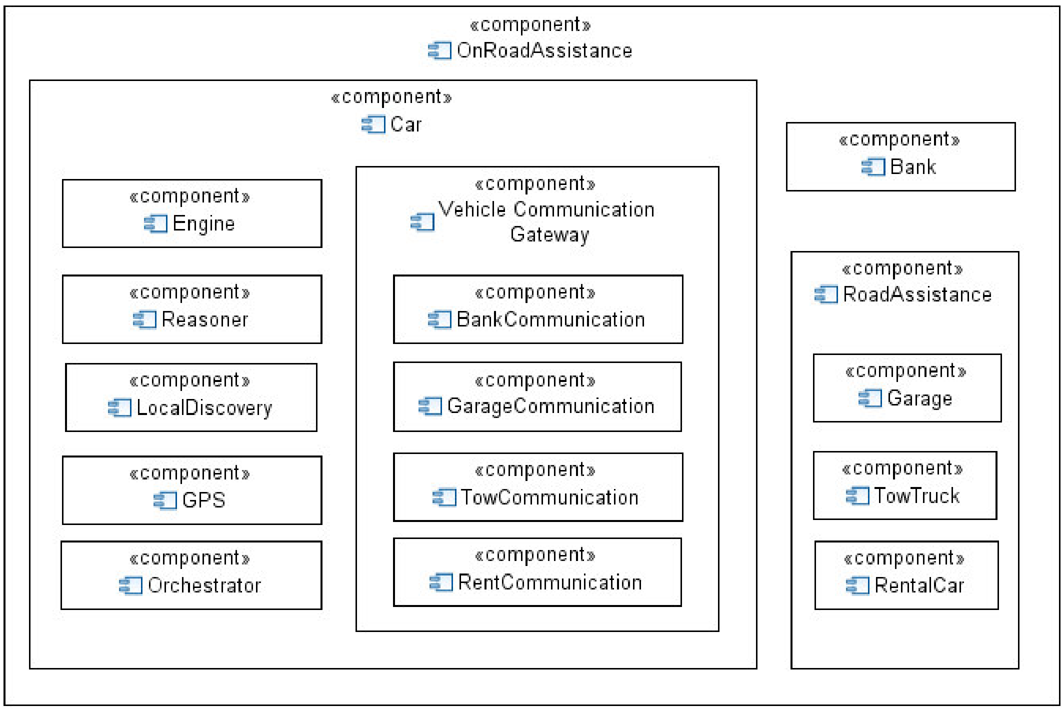

Figure 2: Interplay of state machines.

4. FORMAL VERIFICATION

In this section, we describe the application and results of formal modelling and verification techniques to the on road assistance scenario. First, we discuss a specification of this scenario as a set of communicating UML state machines. Second, the results of using the model checker UMC to verify several properties expressed in the temporal logic UCTL are presented and interpreted.

In the previous section, we presented the on road assistance scenario by means of UML activity diagrams using a UML Profile for SOC (*cf.* Figure 1). To carry out a formal analysis of the system, we need a description of it in terms of standard UML state machines. It would of course be highly desirable if the transformation from the activity diagrams into the corresponding statechart diagrams were performed automatically, *e.g.* by applying model-driven transformation techniques based on graph transformations as described in [15]. The SENSORIA roadmap already points to the development of tools going in this direction [21]. For the current paper, however, we manually performed the required translation, using the original activity diagrams merely as a notation to guide the UMC statechart-based design of the system.

4.1 A Formal Specification

In order to verify behavioural properties over the on road assistance scenario, its requirements model must be accompanied by an operational model formalizing its behaviour. Since our goal was to use UMC to verify properties expressed in UCTL, we formally defined the scenario as a set of communicating UML state machines. The reader can consult our specification via [34]. Figure 2 depicts the UML structure diagram of the set of UML state machines and the subcomponents `BankCommunication` and `Orchestrator` are depicted in Figures 3 and 4, resp.

It is important to note that we have implemented one of the many possible operational models that can be defined in accordance with the scenario's requirements model of [22]. In fact, we made the following assumptions w.r.t. this model:

- We do not define a separate state machine for each component, but rather structure some of them as subcomponents of others (*cf.* Figure 2).
- We abstract altogether from a remote discovery state machine (to search for services in a remote repository).
- `LocalDiscovery` returns at most one choice of services for on road assistance.
- Compensations are explicitly modelled as requests to cancel operations (*viz.* `bankrevoke` and `garagerevoke`).
- All communications between `Car`'s subcomponents, as well as those between these subcomponents and `Bank` and `RoadAssistance` (*i.e.* all service invocations), are modelled as pairs of request/response signals. Whereas this is necessary for the former (synchronous operation calls would deadlock), the latter might equally well be modelled as synchronous operation calls.

While the single state machines are not very complex, their interplay is. In fact, the system has 535 states and 814 transitions and, consequently, validation by hand is not feasible. Note that their potential interplay is much more complex still: It suffices to imagine a system composed of several `Cars`, `Banks`, `Garages`, *etc.*

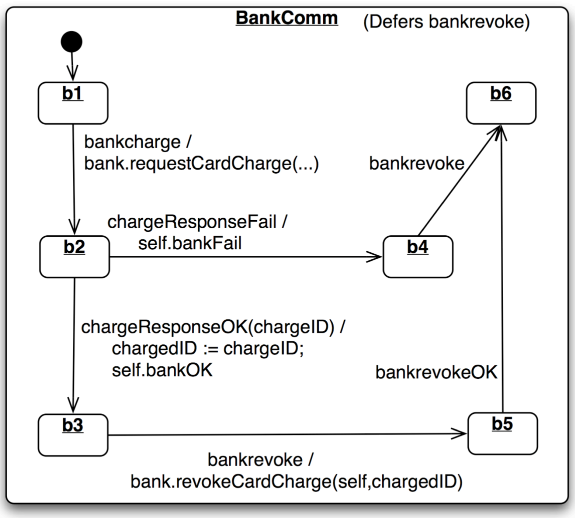

Figure 3: Subcomponent BankCommunication.

4.2 Validation with UMC

We have used UMC v3.4 on an ordinary PC to verify a number of behavioural properties expressed in UCTL over our implementation of the on road assistance scenario.

Service responsiveness

A service is *responsive* if it guarantees a response to each received request. An example of service responsiveness is expressed by the UCTL formula

$$AG\ [\,requestCardCharge\,]\ A\,[\,true\ {}_{true}U_{chargeResponseOK \vee chargeResponseFail}\ true\,], \quad \text{(F1)}$$

which states that each time action `requestCardCharge` takes place, always at a certain moment action `chargeResponseOK` or `chargeResponseFail` takes place. More intuitively: If the `Car` requests the `Bank` to charge a credit card, then the `Bank` will surely reply with a notification of either a successful or a failed attempt to charge the credit card.

We verified formula F1 with UMC and it is true.

Service coordination

A service is *coordinated* if its confirmation is always preceded by a request (for this service). An example of service coordination is expressed by the UCTL formula

$$\neg\, E\ [\,true\ {}_{\neg requestCardCharge}U_{chargeResponseOK}\ true\,], \quad \text{(F2)}$$

which states that it may never happen that action `charge-ResponseOK` takes place if action `requestCardCharge` has not taken place before. More intuitively: The `Car` cannot receive a notification of the fact that the credit card has been charged, if it did not previously request the `Bank` to do so.

We verified formula F2 with UMC and it is true.

Service reliability

A service is *reliable* if it guarantees a successful response whenever it accepts a request (for this service). An example of service reliability is expressed by the UCTL formula

$$AG\ [\,requestGarage\,]\ A\,[\,true\ {}_{true}U_{garageResponseOK}\ true\,], \quad \text{(F3)}$$

which states that each time action `requestGarage` takes place, always at a certain moment action `garageResponseOK` takes place. More intuitively: Reservation requests from `Car` to `Garage` are always followed by a notification of success.

We verified formula F3 with UMC and it is false. Note that this is not surprising: The `Garage` service might be temporarily unable to provide the requested service (so it sends the unsuccessful response `garageResponseFail`). Note that the `Garage` service is responsive, *i.e.* a formula similar to formula F1 does hold also for the `Garage` service.

Uniqueness of response

A service guarantees *uniqueness of response* if it guarantees a single successful response whenever it accepts a request (for this service). An example of the uniqueness of a response is expressed by the UCTL formula

$$AG\ [\,requestGarage\,]\ AF\ <garageResponse>\ \neg\, EF\ <garageResponse>\ true, \quad \text{(F4)}$$

in which `garageResponse` stands for `garageResponseOK` $\vee$ `garageResponseFail`. Formula F4 states that each time action `requestGarage` takes place, always eventually action `garageResponseOK` or `garageResponseFail` takes place and afterwards it may not happen that eventually one of the latter two actions takes place again. More intuitively: After a reservation request from `Car` to `Garage`, it cannot happen that the `Car` receives more than one notification.

We verified formula F4 with UMC and it is true.

Functional system behaviour

Often one wants to check specific relations between the effects of different service invocations. An example of such a relation is that the success of a service request is always followed by either its cancellation or by the success of another service request, as is expressed by the UCTL formula

$$AG\ [\,chargeResponseOK\,]\ A\,[\,true\ {}_{true}U_{garageResponseOK \vee bankrevokeOK}\ true\,], \quad \text{(F5)}$$

which states that each time action `chargeResponseOK` takes place, always at a certain moment action `garageResponseOK` or `bankrevokeOK` takes place. More intuitively: If the `Bank` sends the `Car` a notification of the fact that the credit card was successfully charged, then in the future either this operation of charging the credit card will be revoked or the `Car` will receive a notification of the fact that the `Garage` has been reserved.

We verified formula F5 with UMC and it is true.

The results of the verifications are summarized in Table 1.

formula	property	validity
F1	service responsiveness	true
F2	service coordination	true
F3	service reliability	false
F4	uniqueness of response	true
F5	functional system behaviour	true

Table 1: Validation results.

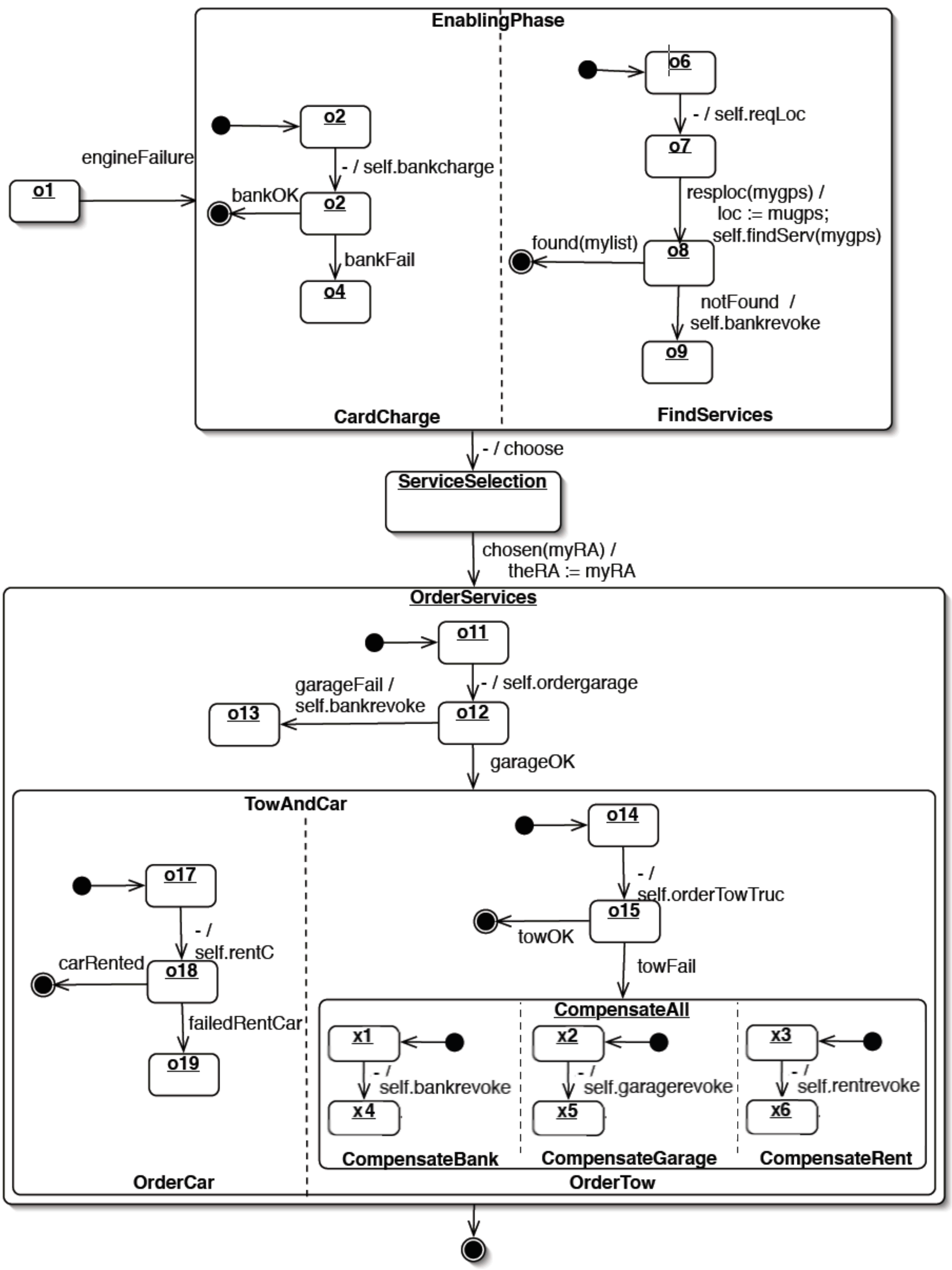

Figure 4: Subcomponent `Orchestrator`.

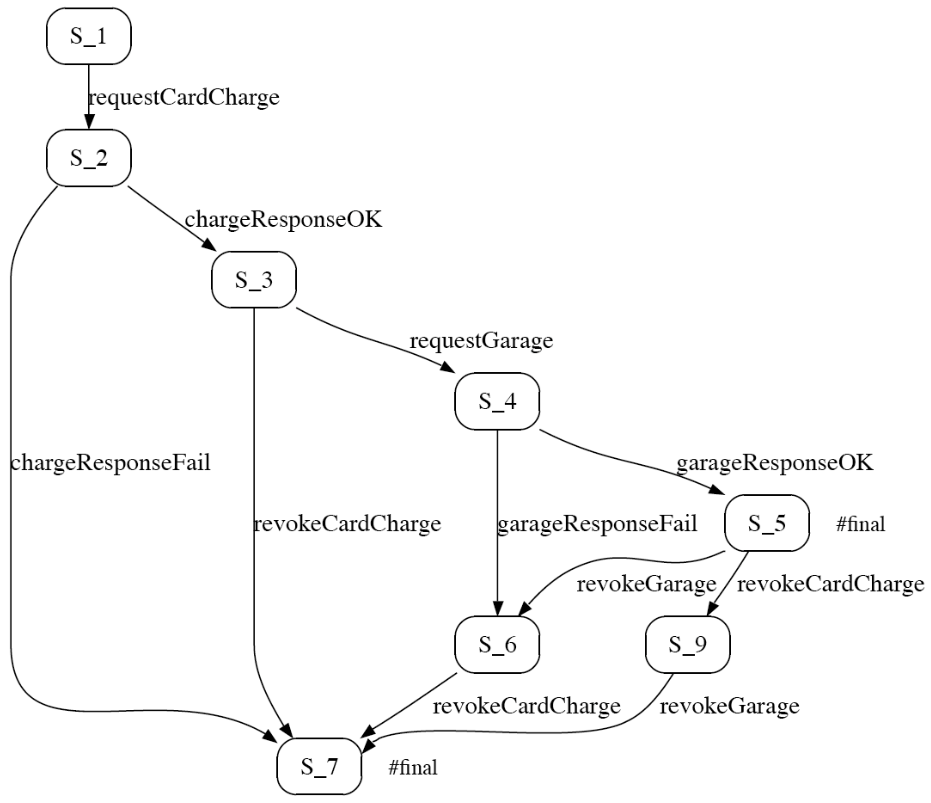

Figure 5: Abstract behavioural slice.

The verifications show that the requirements model of the on road assistance scenario has been well designed in [22].

As shown in Figure 5, UMC can also be used to generate, *e.g.*, all abstract traces of the system obtained by observing only the interactions of `Bank` and `Garage` with `Car`. Such graphs give a precise and complete understanding of the relation between the activities of charging a credit card and of reserving a garage. In fact, formula F5 appears to be just one of the many logical properties that would need to be verified w.r.t. these two activities to obtain the same understanding. The advantages of verification by means of a set of formulae are that these can usually be evaluated in an efficient way also on large or even infinite systems, and that in case of unexpected behaviour a precise explanation can be obtained from the model checker by simply illustrating the sequence of computation steps (and a fragment of the traversed L^2TS) that has led to the verification result.

5. RELATED WORK

Formal methods are intensively used in the transportation field to rigorously design and develop mission and safety-critical real-time subsystems. In this sector, the adoption of formal methods is indeed being encouraged by the need for certifiable code generators (*e.g.* according to the DO-178B, EN 50128 and IEC 61508 standards) and products that can easily be validated (*cf.* [30, 32]). Our aim is to carry on experimental research on temporal logics and other techniques useful for the formal analysis of service-oriented systems, rather than to support in a direct way the development of safety-critical software by developing commercial automatic code generators or validation tools.

The approach that is probably nearest to ours is that of the HUGO/RT project [20], in which model checking a UML system is achieved through model translation into the input language for the SPIN or UPPAAL model checkers [33, 36]. A recent related approach [12] describes how UML diagrams can be translated into the Maude language and subsequently be model checked. While HUGO/RT, SPIN and Maude's model checker rely on a state-based linear-time temporal logic (LTL) to express a system's properties, our prototype adopts the action- and state-based branching-time temporal logic UCTL. Moreover, having an in-house model checker (UMC) for our logics allows us to easily experiment with the best logical features to specify the desirable properties of service-oriented systems, exploiting the advantages of both the on-the-fly approach of the model checker and the linear complexity of the evaluation algorithm.

6. LESSONS LEARNED

In the context of the EU project SENSORIA [31], we have used our longstanding experience with formal modelling and verification techniques, to validate the requirements model of an automotive scenario from the SOC domain. The specific on road assistance scenario that we have validated was one of the outcomes of discussions with automotive experts on possible new services for drivers to be provided by the in-vehicle computers. The validation has shown that the requirements model of the on road assistance scenario has been well designed.

The case study described in the paper has moreover shown the usefulness and feasibility of a formal approach to specifying and rigorously analyzing a system design, also in industrial contexts. So far, we have not yet tried to use the model checker UMC for a particularly big and complex system, since we have been mostly interested in experimenting with its functionalities from a more qualitative point of view, rather than from a quantitative point of view. This is also the reason for which the case study in this paper has been kept relatively simple and for which we have not provided any information in Table 1 on the resource usage and computation time spent.

There is definitely much room for improvement of UMC still, *e.g.* regarding the UML support of the tool, regarding the temporal logic UCTL, regarding further optimizations of the on-the-fly model-checking algorithm and regarding the overall usability and user-interface issues.

7. CONCLUSIONS AND FUTURE WORK

This paper describes ongoing work on applying academic experience with formal modelling and verification (model checking, to be precise) to an industrial case study on automotive systems taken from the SOC domain. This work is performed in the context of the EU project Sensoria [31], whose aim is to develop a comprehensive and pragmatic—but theoretically well-founded—approach to software engineering for service-oriented systems. The reader is referred to [37] for an exemplary overview of some of the service-oriented techniques and methods that are currently being developed in Sensoria.

We defined the on road assistance scenario as a set of communicating UML state machines, after which UMC was used to verify a number of properties expressed in UCTL. The outcome showed that the requirements model of the on road assistance scenario has been well designed in [22]. The reader is referred to [22] for an overview of related approaches to the specification and analysis of the on road assistance scenario (and other scenarios from the automotive case study) that have been developed for the engineering of service-oriented systems within the scope of Sensoria.

We have several ideas to extend the work presented in this paper in the future. To begin with, we would like to relax some of the assumptions we made w.r.t. the requirements model (*cf.* Section 4.1): One can think of the addition of a remote service discovery engine or of more advanced compensation handling mechanisms. We would also like to validate more complex scenarios: One can think of a scenario in which several drivers, each with a stranded car, compete for a limited number of tow trucks. Moreover, we would like to extend our current implementation to be able to use the full potential of the action- and state-based temporal logic UCTL: One can think of properties that require state-based formulae. We also would like to evaluate performance issues of the scenario of this paper: One can think of using a suitable specification language and model checker to verify also quantitative properties, e.g. by using action- and state-based stochastic logics [1, 7, 17]. Finally, we would like to see whether the scenario of this paper is suitable to validate truly specific service-oriented features: One can think of properties described by means of the service-oriented logic SocL [10], which is a very recent specialization of UCTL meant to capture peculiar aspects of services.

8. ACKNOWLEDGEMENTS

The research reported in this paper has been partially funded by the EU project Sensoria (IST-2005-016004) [31].

9. REFERENCES

[1] C. Baier, L. Cloth, B. Haverkort, M. Kuntz and M. Siegle. Model checking action- and state-labelled Markov chains. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04), Florence, Italy*, pages 701–710. IEEE Computer Society, Los Alamitos, CA, 2004.

[2] M.H. ter Beek, S. Gnesi, F. Mazzanti and C. Moiso. Formal Modelling and Verification of an Asynchronous Extension of SOAP. In A. Bernstein, T. Gschwind and W. Zimmermann, editors, *Proceedings of the 4th IEEE European Conference on Web Services (ECOWS'06), Zurich, Switzerland*, pages 287–296. IEEE Computer Society, Los Alamitos, CA, 2006.

[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. To appear in *Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'07), Berlin, Germany*, volume 4916 of *Lecture Notes in Computer Science*. Springer, Berlin, 2008.

[4] G. Bhat, R. Cleaveland and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proceedings of the 10th IEEE Symposium on Logics in Computer Science (LICS'95), San Diego, CA, USA*, pages 388–397. IEEE Computer Society, Los Alamitos, CA, 1995.

[5] S. Chaki, E.M. Clarke, J. Ouaknine, N. Sharygina and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing* 17(4): 461–483, 2005.

[6] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8(2): 244–263, 1986.

[7] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti and M. Massink. Model checking mobile stochastic logic. *Theoretical Computer Science* 382(1): 42–70, 2007.

[8] R. De Nicola and F.W. Vaandrager. Actions versus State based Logics for Transition Systems. In I. Guessarian, editor, *Proceedings Spring School on Semantics of Systems of Concurrent Processes, La Roche Posay, France*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, Berlin, 1990.

[9] R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM* 42(2): 458–487, 1995.

[10] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese and F. Tiezzi. A model checking approach for verifying COWS specifications. To appear in *Proceedings of the 11th Conference on Fundamental Approaches to Software Engineering (FASE'08), Budapest, Hungary, Lecture Notes in Computer Science*. Springer, Berlin, 2008.

[11] J.-C. Fernandez, C. Jard, T. Jéron and C. Viho. Using On-The-Fly Verification Techniques for the Generation of test Suites. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96), New Brunswick, NJ, USA*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer, Berlin, 1996.

[12] P. Gagnon, F. Mokhati and M. Badri. Applying Model Checking to Concurrent UML Models. *Journal of Object Technology* 7(1):59–84, 2008.

[13] S. Gnesi and F. Mazzanti. On the fly model checking of communicating UML State Machines. In *Proceedings of the 2nd International Conference on Software Engineering Research, Management & Applications (SERA'04), Los Angeles, CA, USA*, pages 331–338, 2004.

[14] S. Gnesi and F. Mazzanti. A Model Checking Verification Environment for UML Statecharts. Presented at the XLIII Annual Italian Conference AICA, Udine, 2005.

[15] L. Gönczy and D. Varró. Model-Driven Transformations for Deployment—Prototype. Sensoria Deliverable 6.4a, September 2007. Available via [31].

[16] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM* 32(1): 137–161, 1985.

[17] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. Towards model checking stochastic process algebra. In W. Grieskamp, T. Santen and B. Stoddart, editors, *Proceedings of the 2nd International Conference on Integrated Formal Methods (IFM'00), Dagstuhl, Wadern, Germany*, volume 1945 of *Lecture Notes in Computer Science*, pages 420–439. Springer, Berlin, 2000.

[18] M. Huth, R. Jagadeesan and D.A. Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In D. Sands, editor, *Programming Languages and Systems—Proceedings of the 10th European Symposium on Programming (ESOP'01), Genova, Italy*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer, Berlin, 2001.

[19] E. Kindler and T. Vesper. ESTL: A Temporal Logic for Events and States. In J. Desel and M. Silva, editors, *Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN'98), Lisbon, Portugal*, volume 1420 of *Lecture Notes in Computer Science*, pages 365–384. Springer, Berlin, 1998.

[20] A. Knapp, S. Merz and Ch. Rauh. Model Checking—Timed UML State Machines and Collaborations. In W. Damm and E.-R. Olderog, editors, *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02), Oldenburg, Germany*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414. Springer, Berlin, 2002. See also http://www.pst.ifi.lmu.de/projekte/hugo/

[21] A. Knapp and G. Zhang. Model Transformations for Integrating and Validating Web Application Models. In H.C. Mayr and R. Breu, editors, *Proceedings of Modellierung 2006 (MOD'06), Innsbruck, Austria*, volume 82 of *Lecture Notes in Informatics*, pages 115–128. Gesellschaft für Informatik, Bonn, 2006.

[22] N. Koch and D. Berndl. Requirements Modelling and Analysis of Selected Scenarios of the Automative Case Study. Sensoria Deliverable 8.2a, September 2007. Available via [31].

[23] N. Koch, P. Mayer, R. Heckel, L. Gönczy and C. Montangero. UML for Service-Oriented Systems. Sensoria Deliverable 1.4a, September 2007. Available via [31].

[24] P. Liggesmeyer, M. Rothfelder, M. Rettelbach and T. Ackermann. Qualitätssicherung Software-basierter Technischer Systeme—Problembereiche und Lösungsansätze. *Informatik Spektrum*, 21(5):249–258, 1998.

[25] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification.* Springer, Berlin, 1992.

[26] F. Mazzanti. 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

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

[28] M. Müller-Olm, D.A. Schmidt and B. Steffen. Model-Checking—A Tutorial Introduction. In A. Cortesi and G. Filé, editors, *Proceedings of the 6th International Symposium on Static Analysis (SAS'99), Venice, Italy*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, Berlin, 1999.

[29] A. Saad. Java-based Functionality and Data Management in the automobile—Prototyping at BMW Car IT GmbH. JavaSPEKTRUM. SIGS Datacom, March 2003.

[30] SCADE suite. http://www.esterel-technologies.com/products/scade-suite/

[31] EU project Sensoria (IST-2005-016004). http://www.sensoria-ist.eu/

[32] SPARK toolset. http://www.praxis-his.com/sparkada/

[33] Spin model checker. http://www.spinroot.com

[34] UMC model checker. http://fmt.isti.cnr.it/umc/

[35] Unified Modeling Language. http://www.uml.org/

[36] Uppaal tool. http://www.uppaal.com

[37] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch and A. Schroeder. Semantic-Based Development of Service-Oriented Systems. In E. Najm, J.-F. Pradat-Peyre and V. Donzeau-Gouge, editors, *Proceedings of the 26th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'06), Paris, France*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer, Berlin, 2006.