

Semantic-Based Development of Service-Oriented Systems*

Martin Wirsing¹, Allan Clark², Stephen Gilmore², Matthias Hölzl¹,
Alexander Knapp¹, Nora Koch^{1,3}, and Andreas Schroeder¹

¹ Ludwig-Maximilians-Universität München, Germany

² University of Edinburgh, United Kingdom

³ F.A.S.T. GmbH, Germany

Abstract. Service-oriented computing is an emerging paradigm where services are understood as autonomous, platform-independent computational entities that can be described, published, categorised, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. The IST-FET Integrated Project SENSORIA aims at developing a novel comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated in a pragmatic software engineering approach. In this paper we present first ideas for the SENSORIA semantic-based development of service-oriented systems. This includes service-oriented extensions to the UML, a mathematical basis formed by a family of process calculi, a language for expressing context-dependent soft constraints and preferences, qualitative and quantitative analysis methods, and model transformations from UML to process calculi. The results are illustrated by a case study in the area of automotive systems.

1 Introduction

Service-oriented computing is an emerging paradigm where services are understood as autonomous, platform-independent computational entities that can be described, published, categorised, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. These characteristics pushed service-oriented computing towards nowadays widespread success, demonstrated by the fact that many large companies invested a lot of efforts and resources to promote service delivery on a variety of computing platforms, mostly through the Internet in the form of Web services. Tomorrow, there will be a plethora of new services as required for e-government, e-business, and e-science, and other areas within the rapidly evolving Information Society. These services will run over “global computers”, i.e., computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms, etc., with particular regard to exploiting their universal scale and the programmability of their services.

The aim of IST-FET Integrated Project SENSORIA is to develop a novel comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated in a pragmatic software

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

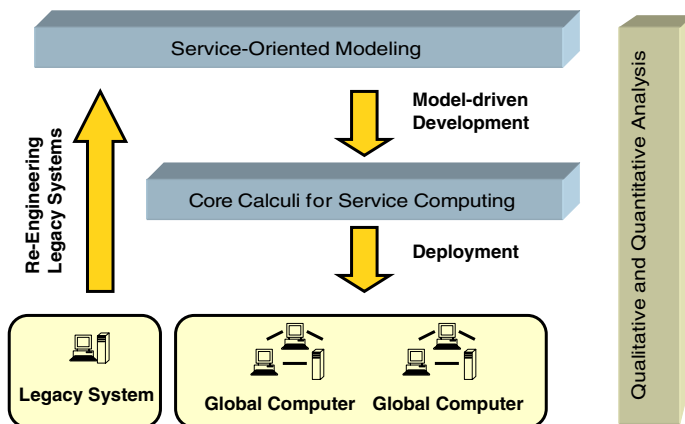


Fig. 1. The SENSORIA approach to service-oriented systems development

engineering approach. This includes a new generalised concept of service, new semantically well-defined modelling and programming primitives for services, new powerful mathematical analysis and verification techniques and tools for system behaviour and quality of service properties, and novel model-based transformation and development techniques.

In the envisaged software development process, services are modelled in a platform-independent architectural design layer; by using model transformations, these models are then transformed and refined to the service computing platform of SENSORIA which, in turn, can be used for generating implementations over different global computing platforms in a (semi-)automated way. On the other hand, legacy code is transformed systematically into service oriented software models (see Fig. 1).

The added value of SENSORIA to this widely used process comes from the deep mathematical foundations and their associated analysis methods. A typical scenario could be as follows: A service engineer will write her design of a service-oriented system for global computing in a precisely defined specialisation of UML for services on global computers. This UML extension will be carefully designed to be automatically connected with well-defined mathematical models of global computing services. These models come with mathematical theories, techniques and tools for analysing their qualitative and quantitative properties such as performance, security, costs, mobility, and distribution. By automatic translation of the analysis results back to UML the service engineer will get direct feedback on her system design from the mathematical models and can revise her models accordingly. Then she can use again mathematically well-founded transformation and refinement techniques for constructing the implementation or she can use the SENSORIA analysis techniques for checking the appropriateness of the offered or discovered services of other parties. She could also adopt the re-engineering techniques of SENSORIA for bringing legacy code in a service compatible format.

In this paper we present first ideas for the SENSORIA semantic-based development of service-oriented systems. This includes service-oriented extensions to the UML, a mathematical basis formed by a family of process calculi, a language for expressing

context-dependent soft constraints and preferences, qualitative and quantitative analysis methods, and model transformations from UML to process calculi. The results are illustrated by a case study in the area of automotive systems.

The paper is organised as follows: In Sect. 2 we present the running example, (an excerpt of) the UML extension for services, and two process calculi PEPA [22] and Sagas [13] which are used for analysing the UML designs and as semantic basis for service transactions with compensation. Moreover, we show how soft constraints and preferences can be used for choosing the best service offer. In Sect. 3 we present the SENSORIA model transformation approach and show how we use the VIATRA2 [37,2] model transformation tool for translating UML diagrams with compensation into the Saga calculus and therefore giving semantics to compensations. In Sect. 4 we present some of the SENSORIA methods for qualitative and quantitative analysis; in particular, we show how the dynamic behaviour of a service orchestration can be model checked, and how the performance aspects of a service level agreement for providing help in an accident scenario can be analysed. We conclude the paper in Sect. 5 with some remarks on further SENSORIA results.

2 Languages for Service-Oriented Systems

Current service description and composition languages such as WSDL [40] and BPEL [9] are tailored to specific technological platforms such as Web Services and the Grid, and address low-level concerns.

The languages that have emerged for composing services into business processes such as WSFL [17], BizTalk [7], WSCI [39] and, most prominently, BPEL and BPEL4WS [10] have limited expressive power. They offer restricted support for concurrency and distribution and are mainly oriented towards programming workflows, making use of interconnection mechanisms that are far too rigid to support modelling business processes at the more abstract architectural layers.

SENSORIA aims at the definition of platform independent linguistic primitives for modelling and programming global service-oriented systems. Language primitives for services and their interactions are developed on two different abstraction levels, at the architectural design level and at the programming abstraction level for service overlay computing. The scientific tools used for the definition of programming-level primitives are category theory, process algebra and calculi as well as logics and constraints. A UML (Unified Modelling Language) [30] extension to service-oriented modelling makes the formal approaches available for practitioners and is the basis for many SENSORIA verification techniques. An additional soft-constraint-based language for service selection allows the declarative specification of orchestrations.

2.1 Automotive Case Study

Today's embedded computers in cars can access communication networks like the Internet and thereby provide a variety of new services for cars and drivers. A set of possible scenarios of the automotive domain are examined within the scope of the SENSORIA project, among which we select a car repair and an accident assistance scenario for illustrating the different techniques presented in this article.

In the car repair scenario, the diagnostic system reports a severe failure in the car engine so that the car is no longer drivable. The car's discovery system identifies garages, car rentals and towing truck services in the car's vicinity. The in-vehicle service platform selects a set of adequate offers taking into account personalised policies and preferences of the driver and tries to order them. We assume that the owner of the car has to deposit a security payment before being able to order services.

In the accident assistance scenario, the car's airbag is deployed after an accident. This causes the safety system to report the car's location to an accident report centre. This centre attempts to determine the severity of the accident and take appropriate actions.

2.2 Language for Service Orchestration

In the car repair scenario, it is necessary to invoke services in a specific order. It is not possible, e.g., to order any service before the security payment was deposited. Similarly, when undoing orders of repair assistance services, the security payment may be returned only after all orders were cancelled. That is to say, compensations of executed sequential forward actions must be performed in reverse order.

To address such compensation scenarios within SENSORIA, Bruni et al. [13] defined a calculus to provide a first semantic basis for service orchestration (a more elaborated calculus is under development). This calculus builds upon sagas [18], which is a formalism for long running transactions (LRT) with compensations initially developed for database systems. LRTs are transactions that require a very long time to complete (e.g. hours or days). In such cases, traditional database techniques to guarantee the ACID property, such as locking resources, are not suitable. Instead, sagas allow the specification of compensation actions which are installed dynamically when forward actions succeed.

As service orchestrations must be able to handle LRTs, most orchestration languages allow the specification of compensation actions. Hence, the saga calculus is a suitable semantic basis for service orchestrations. Within the calculus, one can specify compensation (%), sequential (;) and parallel (|) composition of actions, as well as sub-transactions ([]). The semantics of sub-transactions is elegant for the specification of orchestrations: if a sub-transaction fails and compensates successfully, the sub-transaction is considered successful. If the enclosing transaction fails however, the sub-transaction is required to compensate as well. This allows the specification of dependencies between, e.g., ordering of services, or the continuation of processes although other parallel processes fail.

The compensation strategy defined by the calculus is interruption based, that is to say, if a parallel branch fails all other branches are interrupted and compensate immediately. They do not need to execute forward actions until their control flow joins. In sequential composition, the compensation order is required to be the inverse of the forward action ordering. For the car repair scenario for example, we assume that the garage appointment is ordered next to the security payment deposit. This leads to the specification

$$\begin{aligned} & \textit{ChargeCreditCard} \% \textit{RevokeCharge} ; \\ & \textit{OrderGarageAppointment} \% \textit{CancelGarageAppointment} \end{aligned}$$

If the forward flow *ChargeCreditCard* ; *OrderGarageAppointment* fails, it is compensated by executing *CancelGarageAppointment* ; *RevokeCharge* in exactly that ordering, which is the behaviour required by the scenario specification.

The saga calculus supports further constructs such as nondeterministic choice, exception and failure handling, and by this covers the semantics of common orchestration language constructs (see [13] for more details).

2.3 Language for Quantitative Analysis

Performance Evaluation Process Algebra (PEPA) [22] is a high-level language for quantitative analysis of systems. PEPA is a *stochastic process algebra* which extends classical process algebras by associating a duration with each activity used in a PEPA model. Thus where classical process algebras such as CCS and CSP deal with instantaneous actions which abstract away from time PEPA has instead continuous-time activities whose durations are quantified by exponentially-distributed random variables. PEPA models describe finite-state systems and via the operational semantics of the language a PEPA model gives rise to a continuous-time finite-state stochastic process called a Markov chain.

Continuous-time Markov Chains (CTMCs) are amenable to solution using standard procedures of numerical linear algebra such as Gaussian elimination or conjugate gradient methods. These can be applied to find the compute the *steady-state* or *equilibrium* probability distribution over the model. From this it is straightforward to compute conventional performance measures such as utilisation or throughput.

More advanced tools [27,11] can perform *transient* analysis of the CTMC where one considers the probability distribution at a chosen instant of time. It is possible to use these results to perform more complex quantitative analysis such as computing *response time* measures and *first passage time quantiles* as used in service-level agreements.

The PEPA process algebra is a compact formal language with a small number of combinators. Components perform activities. Activities have a type and rate specified using *prefix* (.). Alternative behaviours can be composed in a *choice* (+). Parallel composition of components uses CSP-style synchronisation over a set of activity types (\bowtie). Private behaviour can be hidden (*/*).

2.4 UML Extension for Service Oriented Architectures

Within the SENSORIA approach services are modelled with UML. For the static aspects of service-oriented software systems, this representation effort ranges from rather simple, stereotyped language extensions for introducing services to more complicated structures like dependency relations between services and their contextual relationships to resources and legacy systems. The dynamic parts of service-oriented software, in particular orchestration and choreography of services are supported by developing primitives for interaction and activity modelling that take into account possible failures and quality-of-service aspects. The extensions will incorporate structural and behavioural as well as functional and non-functional notions.

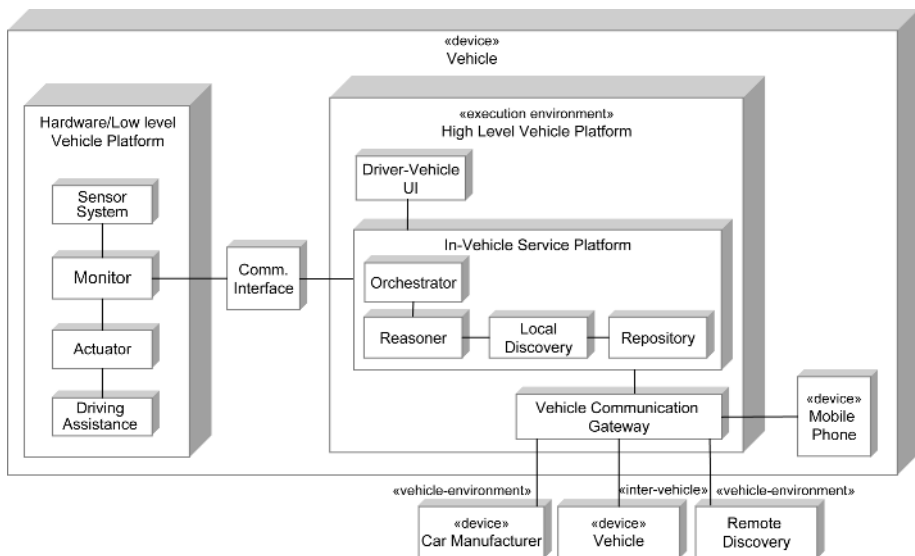


Fig. 2. Simplified architecture of car and car environment

The structure of a service oriented architecture can be visualised by UML deployment and composite structure diagrams. A deployment diagram is used to represent the—usually nested—nodes of the architecture, i.e. hardware devices or software execution environments. Fig. 2 shows a UML deployment diagram of the car and its environment as first approximation to an architecture model. The nodes are connected through communication paths that show the three types of communication that characterise the automotive domain: intra-vehicle communication, inter-vehicle communication, and communication among vehicle and environment such as communication with the car manufacturer or a remote discovery server. Note that the architecture comprises a local as well as a remote discovery service in order to find services in the local repository.

Modelling Structural Aspects. Service oriented architectures are highly dynamic because services are only loosely coupled, i.e., a service often needs to be discovered before it is connected and can be disconnected at run-time as well. Hence, different modelling features are required to express evolving connections. In addition to UML deployment diagrams, which give a static view of the architecture, a representation showing the evolution of an architecture is required. Baresi, Heckel, Thöne and Varró propose the construction of models visualising the functional aspects encapsulated in business-related components [3]. We use these UML structure diagrams to represent the evolving connections within the service oriented architecture of the vehicle and its environment. Fig. 3 shows the car internal components, a temporary connection to the discovery service of the car manufacturer, and a remote service (car rental) which knows the remote service discovery and will (later) publish its description to the remote service discovery. Other remote services such as tow truck and garage and their relationship to the discovery service can be modelled analogously. After publishing, the

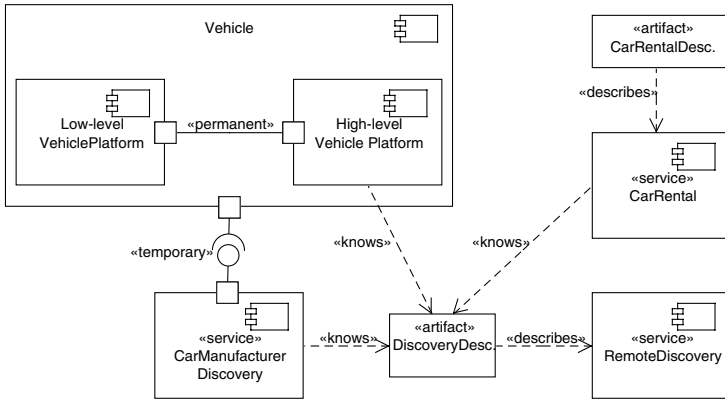


Fig. 3. Car components modelled with UML extension for SOA before service publishing

discovery service knows the description of the published services, so that it can pass these descriptions to service requesters at service discovery time.

Three different types of connections are identified: discovery connection, permanent connection (as in modelling of non service oriented architectures) and temporary connections. For more details about the last two types the reader is referred to [3]. The discovery connection is based on the information provided by a discovery service.

We can observe these three types of connections in the service oriented vehicle architecture. In order to be invoked services need to be discovered before the binding to the car’s on-board system takes place. This type of connection using a discovery process is visualised with a «knows» stereotyped dependency, see Fig. 3. A temporary connection from the car on-board system to the car manufacturer discovery service is graphically represented by a UML connector with interfaces. For a permanent service we select a UML connector without interfaces as shown between components within the vehicle in Fig. 3.

Three components are involved in the execution of service orderings: a service discovery which may be local or external to the car, a reasoner for service selection and a service orchestrator, see Fig. 4.

Modelling Behavioural Aspects. The most interesting aspect when modelling the behaviour of a service oriented system is the workflow describing the orchestration of services. Modelling orchestration of services includes specifying non-functional properties of services such as performance and resource consumption, and also modelling transactional business processes that may require a very long period of time in order to complete. As discussed above, the key technique to handle long running transactions is to install compensations which are not directly available in UML.

We start with the modelling of the accident assistance scenario. The accident assistance scenario is concerned with road traffic accidents and dispatch of medical assistance to crash victims. Drivers wishing to use the service must have in-car GPS location tracking devices with communication capabilities and have pre-registered their mobile phone information with the service. If a road traffic accident occurs, the deployment of

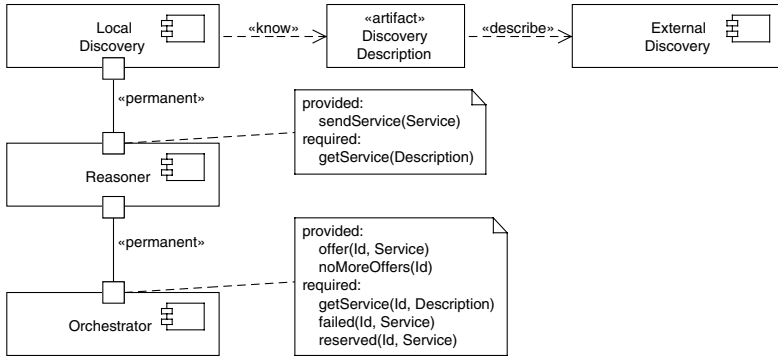


Fig. 4. Components for executing service orderings and their ports

the car airbag causes the on-board safety system to report the car’s current location (obtained by GPS) to a pre-established accident report endpoint which in turn attempts to call the registered driver’s mobile phone. If there is no answer to the call then medical assistance is dispatched to the reported location of the car (presuming that the driver has been incapacitated by injuries sustained in the accident).

We model this scenario in UML as state machine; to represent quantitative aspects (e.g., answer time) we use stereotypes to attach rates to transitions, see Fig. 5. Fig 6 explains the meaning of the rates.

Regarding the modelling of the second scenario, “car repair”, the main focus lies on the specification of an appropriate transactional business process. As discussed in section 2.2, such a business process contains both forward actions and compensations. As UML activity diagrams lack such compensations, we define a set of modelling primitives and corresponding stereotypes for UML activity diagrams.

- Saga is an executable activity node that may have subordinate nodes as an ActivityGroup with the ability to compensate long running transactions.
- CompensableAction specialises UML Action to own exactly one pair of actions (forward action and compensation action).

To provide a more intuitive representation, both forward and compensation actions are drawn separated by a line within CompensableAction instances, although this is not completely UML compliant. The metamodel depicted in Fig. 7 shows how these compensation elements are related to UML elements StructuredActivityNode, ActivityNode and Action.

With these extensions, the orchestration for the car repair scenario can be compactly formulated (Fig. 8). In the modelled business process, the driver’s credit card is charged with the security deposit payment, which will be revoked if ordering the services failed. Then, a garage appointment is searched for. The appointment with the garage will give coordinates to tow the broken down car to, and also a location constraint that restricts the car rental agency that may be ordered. If ordering the car rental fails, the overall process does not fail, as the activity is enclosed in a sub-transaction. However, if ordering a tow truck fails the garage appointment has to be cancelled as well. For this reason, the

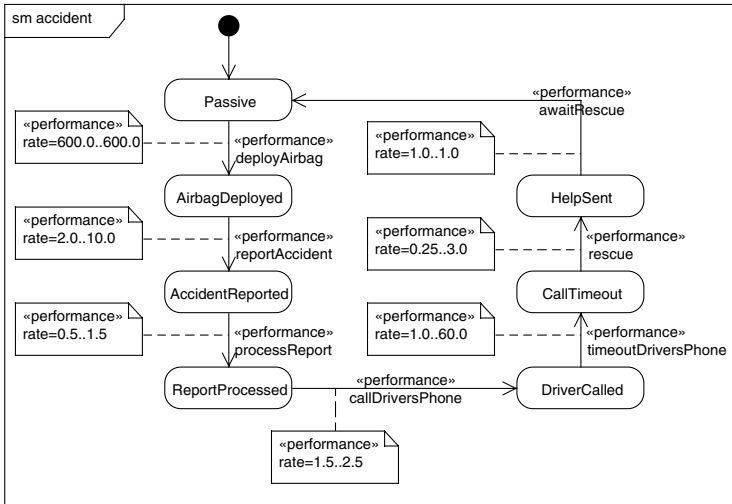


Fig. 5. State machine of the accident assistance scenario

Rate	Value		Meaning
	min	max	
r_1	600.0	600.0	an airbag deploys in 1/10 of a second
r_2	2.0	10.0	the car can transmit location data in 6 to 30 seconds
r_3	0.5	1.5	it takes about one minute to register the incoming data
r_4	1.5	2.5	it takes about thirty seconds to call the driver's phone
r_5	1.0	60.0	give the driver from a second to one minute to answer
r_6	0.25	3.0	vary about one minute to decide to dispatch medical help
r_7	1.0	1.0	arbitrary value — the driver is now awaiting rescue

Fig. 6. Table of minimum and maximum values of the rates from Fig. 5. All times are expressed in minutes. Thus a rate of 1.0 means that something happens once a minute (on average). A rate of 6.0 means that the associated activity happens six times a minute on average, or that its mean or expected duration is ten seconds, which is an equivalent statement.

orchestrator will try to order a tow truck service until either no more service offers are found or the ordering succeeds. If ordering a tow truck fails the rental car delivery will be redirected to the driver's actual location.

It is obviously possible to model the same orchestration with a plain UML activity diagram, and handle compensations as exceptions (Fig. 9). This requires explicit programming of the compensations and the conditions under which they are executed. In addition to actions, activities and control nodes, the specification requires an InterruptibleActivityRegion in order to terminate all active and pending activities of the region in case an interruption occurs. Even in this simple scenario, this approach requires the verification of three conditions. For larger scenarios the diagram's complexity will increase and its usefulness will decrease rapidly. Furthermore, it is difficult to explicitly model the silent failure of the car rental.

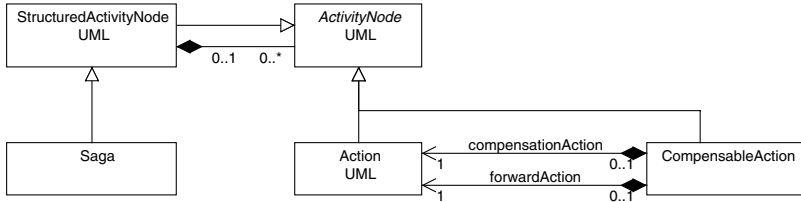


Fig. 7. UML extension for sagas

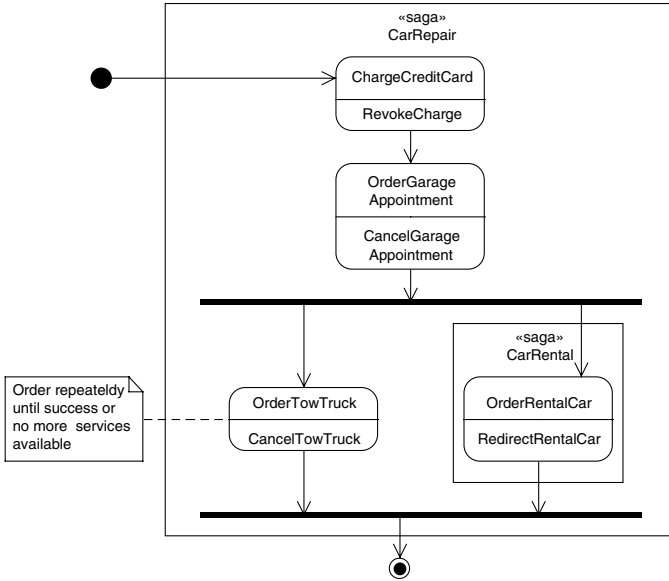


Fig. 8. Modelling car repair workflow with UML extension for sagas

2.5 Soft Constraints for Selecting the Best Service

In many cases service-oriented systems can utilise different combinations of services to achieve their goals. These combinations differ in functional and non-functional aspects like cost, reliability or performance. The reasoning component (cf. Fig. 4) of a service-oriented system decides how the available services are orchestrated so that the best compromise between different goals of the system is achieved. Soft constraints are a promising way to specify and implement reasoning mechanisms. In the case study, soft constraints are used for service selection only.

Soft constraints are an extension of classical constraints to deal with non-functional requirements, over-constrained problems and preferences. Instead of determining just a subset of admissible domain elements, a soft constraint assigns a grade—to be chosen from a set of finite or infinitely many “preference” values—to each element of the application domain. Bistarelli, Montanari and Rossi [6,5] have developed a very elegant semiring based theory of soft constraints where many different kinds of soft constraints can be represented and combined in a uniform way over so-called constraint semirings

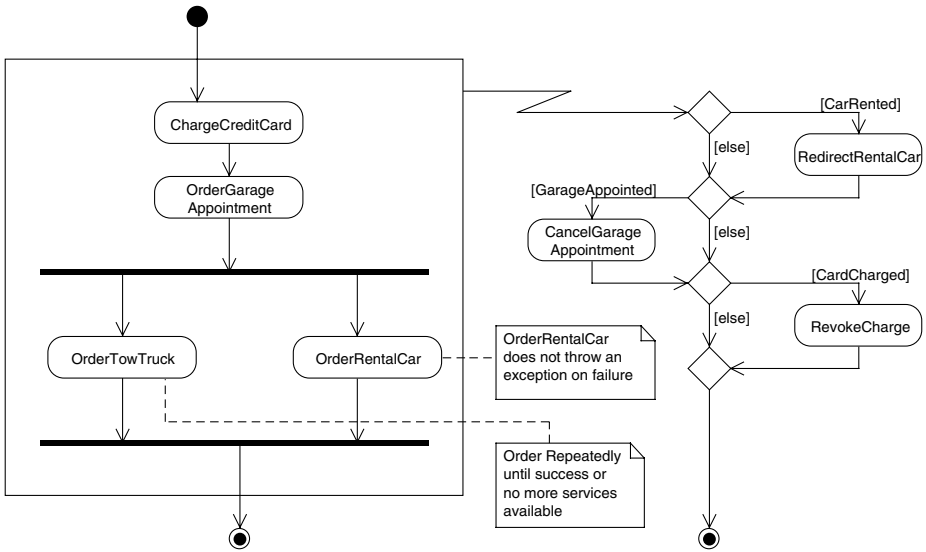


Fig. 9. Modelling car repair workflow with plain UML activity diagrams

(*c-semirings*). Examples for *c-semirings* are the semiring of Boolean values *Bool* or the “fuzzy natural numbers” *FuzzyNat*. The *Bool* semiring can be used to express hard constraints in the *c-semiring* framework; in the *FuzzyNat* semiring the value 0 is used to represent inadmissible solutions, higher values are used to represent increasingly preferred solutions.

In *SENSORIA* we are developing a language which extends the *c-semiring* approach with possibilities to specify preferences between constraints and to vary the constraints according to a *context*. This simplifies the description of behaviours of systems in a dynamically changing environment and with complex trade-offs between conflicting properties.

A context is an expression in a suitable logic (e.g., predicate logic or temporal logic) which can serve as a guard for a constraint. For example, the distance to the destination might determine whether the quick availability of a rental car is important or not. In this case, “*distance < 20km*” is a context that can restrict the applicability of a constraint to situations where we are close to our destination. Variables appearing in contexts are called *context variables*; variables appearing free in constraints but not in contexts are called *controlled variables*. In the car repair scenario the context variables will contain, among others, the distance to our destination, the time remaining until the appointment starts, or whether the journey is work related. The controlled variables represent properties of offers. Each offer is identified by a serial number (*offer-nr*), and other controlled variables are used to specify cost or quality of the offers, see Fig. 10.

A soft constraint is a mapping from (domains of) controlled variables into a *c-semiring*. An expression of the constraint language consists of (1) a set of labelled conditional rules where the constraints contained in the head of the rule depend on the guard, and (2) a set of conditional inequalities between constraint labels which specify preferences between constraints.

Name	Type	Domain	Meaning
<i>distance</i>	<i>context</i>	\mathbb{R}^+	The distance remaining to the target location (in km)
Δ_t	<i>context</i>	\mathbb{N}	The time until the appointment starts (in minutes)
<i>work-related?</i>	<i>context</i>	\mathbb{B}	Is the appointment work-related?
<i>offer-nr</i>	<i>controlled</i>	\mathbb{N}	The unique serial number of an offer
<i>rental-car-cost</i>	<i>controlled</i>	\mathbb{R}^+	The cost of the rental car in Euros
<i>rental-car-availability</i>	<i>controlled</i>	\mathbb{N}	The estimated availability time of the rental car
<i>garage-cost</i>	<i>controlled</i>	\mathbb{R}^+	The cost of the garage (in Euros)
<i>garage-duration</i>	<i>controlled</i>	\mathbb{N}	The estimated duration that the garage needs for the repair (in hours)

Fig. 10. Examples for context variables and controlled variables

In the car repair scenario we maintain hard constraints named *towTruckOffers*, *rental-CarOffers* and *garageOffers* containing disjunctions of the offers that the reasoner obtained from the discovery mechanism. If a new offer is provided to the reasoner the corresponding constraint or constraints are extended by another term.

Other constraints specify the preferences of the users. These constraints are soft constraints, for simplicity we use the fuzzy natural numbers as the domain of all these constraints.

This constraint prefers garages that can repair the car as quickly as possible:

$$\textit{fastRepair} : [\textit{garage-duration} \mid n \mapsto \lfloor 48/n \rfloor]$$

We also may want the repair to be done cheaply, but only if we are paying ourselves. Repairs costing more than 1000 Euros are still acceptable, but only barely.

$$\textit{cheapRepair} : \textbf{in context } \neg \textit{work-related}?$$

$$\textbf{assert } [\textit{garage-cost} \mid n \mapsto \lceil 1000/n \rceil] \textbf{end}$$

We are content to use any kind of car for short distances as long as it is cheap. In this case a cost of more than 100 Euros per day is unacceptable as the constraint evaluates to 0.

$$\textit{shortDistance1} : \textbf{in context } \textit{distance} < 20\text{km}$$

$$\textbf{assert } [\textit{rental-car-cost} \mid n \mapsto \lfloor 100/n \rfloor] \textbf{end}$$

The following constraint means that we want to obtain a car as quickly as possible if the appointment is work-related, the distance is short and we have limited time to go to the appointment. If the rental car takes longer than Δ_t minutes we regard the offer as unacceptable. In this constraint the preference value depends on the context as Δ_t appears in the computation of a value.

$$\textit{shortDistance2} : \textbf{in context } \textit{work-related}?$$

$$\wedge \textit{distance} < 20\text{km} \wedge \Delta_t < 60\text{min}$$

$$\textbf{assert } [\textit{rental-car-availability} \mid (n \mapsto \lfloor \Delta_t/n \rfloor)] \textbf{end}$$

When determining the configuration of a system we might not consider all constraints to be equally important. For example, it might be most important that the car

is repaired both quickly and cheaply, and that we consider the other constraints only if we have several offers that are equal in that respect. This can be expressed by taking the product of the grades computed by both constraints. On the other hand, we consider *shortDistance1* and *shortDistance2* to be incomparable, i.e., we compare the grades of these constraints individually and do not compute a combined value.

$$\begin{aligned} & \textit{fastRepair} * \textit{cheapRepair} > \textit{shortDistance1}, \textit{shortDistance2} \\ & \textit{shortDistance1}, \textit{shortDistance2} > \textit{fastRepair} \\ & \textit{fastRepair} > \textit{cheapRepair} \end{aligned}$$

From a set of constraints and preferences the reasoner can compute either the best solutions, or a set of all solutions that are better than a certain threshold. Two techniques that are used for solving soft constraint systems are branch and bound search [38] and dynamic programming [5].

3 Model Transformation

3.1 Use of Model Transformation in SENSORIA

In the field of service oriented computing, there is a large gain from using model transformation languages, since data formats from different services can be easily mapped to each other with the help of model transformation languages.

Model transformation is used in SENSORIA for several tasks such as e.g. model refinement, deployment, and model analysis. “Refinement” uses model transformations to add additional details to the model, and remove degrees of freedom left in the initial model, trading abstraction for determinism. In SENSORIA we see “deployment” as a special refinement mapping models from the SENSORIA platform level to platform specific software artifacts. This use of model transformations is similar to the transformation from PIMs to PSMs in the MDA approach [28]. Finally, “model analysis” uses model transformations to translate models to logics or languages tailor-made for model analysis, such as process algebras. With the help of these specific models, different crucial qualitative and quantitative properties of the modelled service oriented software system can be verified. Furthermore, model transformations can be used to back-annotate the initial model with analysis results provided by analysis tools. Thus, the fact that analysis is performed on an internal representation can be made transparent to the modeller.

There are several existing model transformation tools and frameworks, some of which are MOF QVT [31], ATLAS ATL [25] and IBM MTF [29]. Unfortunately, all of them have at best a partially defined formal semantics, which makes the use in semantic based engineering hard. Furthermore, MOF QVT still lacks an implementation.

The SENSORIA project comprises two model transformation languages, one with a pragmatic and easy usable programming model (VIATRA2) and one based on a declarative model with strong mathematical foundations (AGG). By having both languages, SENSORIA covers both pragmatism and formal mathematical approaches to model transformations.

VIATRA2 [37,2] (Visual Automated Model Transformations) is a model transformation language supporting graph transformation rules and imperative abstract state

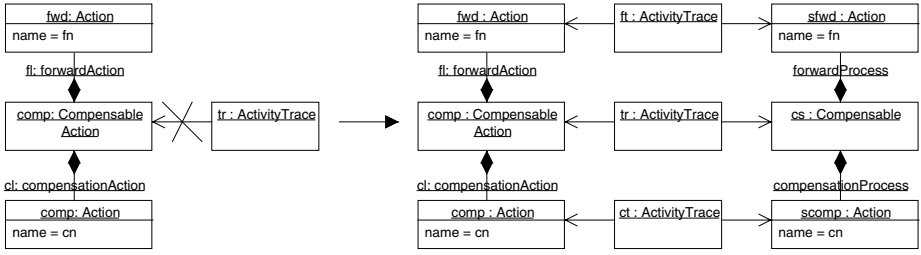


Fig. 11. Graph transformation rule for compensable actions

machine (ASM)-like code. VIATRA2 is a fully hybrid language since it allows to call graph transformation rules from imperative code through rule application calls, and also to call ASM code from declarative graph transformation rules. Nevertheless, VIATRA2 delivers a formal semantics of transformations along with the language. A big advantage of VIATRA2 over other solutions is that it offers a good combination of both imperative code with declarative graph patterns, as imperative constructs ease programming complex transformations considerably [19]. A recursive processing of the source model can be easily intertwined with graph transformation steps. In this way, the advantages from both the declarative and the imperative approach can be brought together.

AGG [1] (Attributed Graph Grammar System) uses an algebraic model transformation approach based on attributed graphs grammars. AGG has a strong mathematical basis in category theory, and follows a more declarative approach to transformation, which allows for easy mathematical analysis of transformation properties.

A graph transformation rule in both VIATRA2 and AGG consists of a left hand side and right hand side. The left hand side specifies the conditions under which the rule applies. The right hand side specifies which changes are made to the model if the rule applies. These changes reuse variables bound in the precondition and manipulates or remove the model elements to which the variables are bound (see, e.g., Fig. 11).

3.2 Model Transformation Examples

We discuss two examples of model transformation use in SENSORIA. The first, which was implemented in VIATRA2, shows how UML activity diagrams can be transformed to saga expressions. The activity diagram from the car repair scenario, Fig. 8, is transformed into the following corresponding sagas expression:

```
[ChargeCreditCard % RevokeCharge ;
OrderGarageAppointment % CancelGarageAppointment ;
(OrderTowTruck % CancelTowTruck | [OrderRentalCar%RedirectRentalCar])]
```

The abstract syntax tree of this expression is created via a VIATRA2 model transformation. We will now give a brief overview of the transformation implementation.

In the UML activity to sagas transformation example, all compensable actions are translated to simple sagas expressions with graph transformation rules (see Fig. 11).

```

rule nodeToSagas(in StartNode, out EndNode, out Sagas) =
  try choose with find p_activityNode(StartNode) do
    try choose NextNode
      with find p_nextNode(StartNode, NextNode) do
        try choose with find p_parseableNode(NextNode) do
          call handleSequence(StartNode, NextNode,
                               EndNode, Sagas);
        else call handleActivity(StartNode, NextNode,
                                 EndNode, Sagas);
      else fail;
    else try choose with find p_sagaNode(StartNode) do
      call handleSagaNode(StartNode, EndNode, Sagas);
    else try choose with find p_parallelStartNode(StartNode) do
      call handleParallelNode(StartNode, EndNode, Sagas);
    else try choose with find p_finalNode(StartNode) do
      call handleFinalNode(StartNode, EndNode, Sagas);
    else
      call handleNOP(StartNode, EndNode, Sagas);

```

Fig. 12. Pattern Matching Dispatch. Called rules call `nodeToSagas` recursively

Non-compensable actions are translated with a similar graph transformation rule. These rules are called as long as applicable. Note that the creation of a trace prevents the repeated application of the rule (A second rule creates actions for activity nodes without compensations).

In a second step, the parallel nodes, nested sagas as well as sequential edges between activity nodes are translated into parallel, saga and sequence expressions respectively. This is done with a recursive imperative rule, since the transformation is complex in nature: saga's parallel expressions may be nested arbitrarily. Starting from the initial node, a node is transformed into a prefix of a saga expression and a recursive rule call is performed on successor nodes to transform the remaining subgraph. Different cases are distinguished by pattern matching (see Fig. 12). All names starting with `p_` denote patterns to be matched, while names starting with `handle` denote imperative rules which call `nodeToSagas` recursively. After this second step terminates successfully, the transformation is complete.

Another example for the use of model transformations is the transformation from UML state and communication diagrams to PEPA. PEPA is, as elaborated above, an algebra for performance analysis. It is possible to extract PEPA models by model transformation from UML state and communication diagrams.

4 Qualitative and Quantitative Analysis

Qualitative and quantitative analysis methods for software systems aim at providing transparent support for the designer throughout the software construction phases based right on those notations used in development. However, proving the correctness of a design or measuring the performance, in general, relies on mathematical models and tool support that are not offered on the level of general software development notations, let

alone using specialised extensions for particular domains. In *SENSORIA*, model transformations are employed to lift methods and tools from the well-founded, abstract, mathematical level to the concrete UML-based design level for service-oriented architectures. Furthermore, backward transformations project analysis results, delivered in terms of the underlying mathematical model, back to modelling notation (cf. section 3). We demonstrate how model checking (qualitative analysis) and performance evaluation (quantitative analysis) are applied to the automotive case study.

4.1 Model Checking Orchestration

The orchestration of services, like in the extended UML activity diagram description in Fig. 8, has to be implemented in an *orchestrator*. We transform the saga-based model into a conventional UML state machine model which details the handling of service allocation and compensation. Using model checking we can prove that the implementation model indeed preserves the compensation properties of the original UML model.

The state machine in Fig. 13 describes an implementation of the car repair workflow as depicted in Fig. 8. It relies on a *reasoner* for choosing services as in Fig. 4. The general idea of the implementation is that every service needed is first requested from the reasoner (`getService`). If the reasoner delivers an offer (`offer`) in a certain amount of time (`after(T)`), the service first is reserved and ordered afterwards. If the reasoner misses the deadline, or the reservation or the ordering fail the orchestrator terminates. In the latter cases the reasoner is informed of the failure (`fail`) such that it can avoid offering a failing service again. During the reservation phase the reasoner may send better offers for the requested service (`offer`). If a service can be reserved eventually, the reasoner is notified (`reserved`) and the offer is ordered. The compensation handling is done using a global compensation handler, like suggested by the UML activity diagram description in Fig. 9.

For the services “charge credit card”, “order tow truck”, and “order car rental”, however, different implementation details are to be realised: On the one hand, the credit card charge need not be reserved; this is an implementation decision. Ordering the tow truck has been marked as to be done repeatedly until no more offers are available. Thus every possible offer has to be checked, that is, previous reservations may have to be compensated in the orchestration process (`compensateReservation`). Finally, the car rental may or may not fail without calling for overall compensation, as it is marked as a nested `«saga»`.

In *SENSORIA* different tools are available for verifying the correct behaviour of the orchestrator implementation. The UML model checker UMC [35] (developed by ISTI) offers the verification of temporal properties described in the μ -calculus directly on UML state machine. The UML model checking tool Hugo/RT [23] (developed by LMU) translates UML state machines and collaborations into different off-the-shelf model checkers like Spin [34] and UPPAAL [36] and also supports Java and SystemC code generation. Here, we used the UPPAAL option of Hugo/RT. By instrumenting the model, which we will not detail here, we checked, e.g., that whenever the global final state is reached all services indeed have been ordered; if an order fails all orders and reservations which have been done up to the point of failure of getting a service—i.e.,

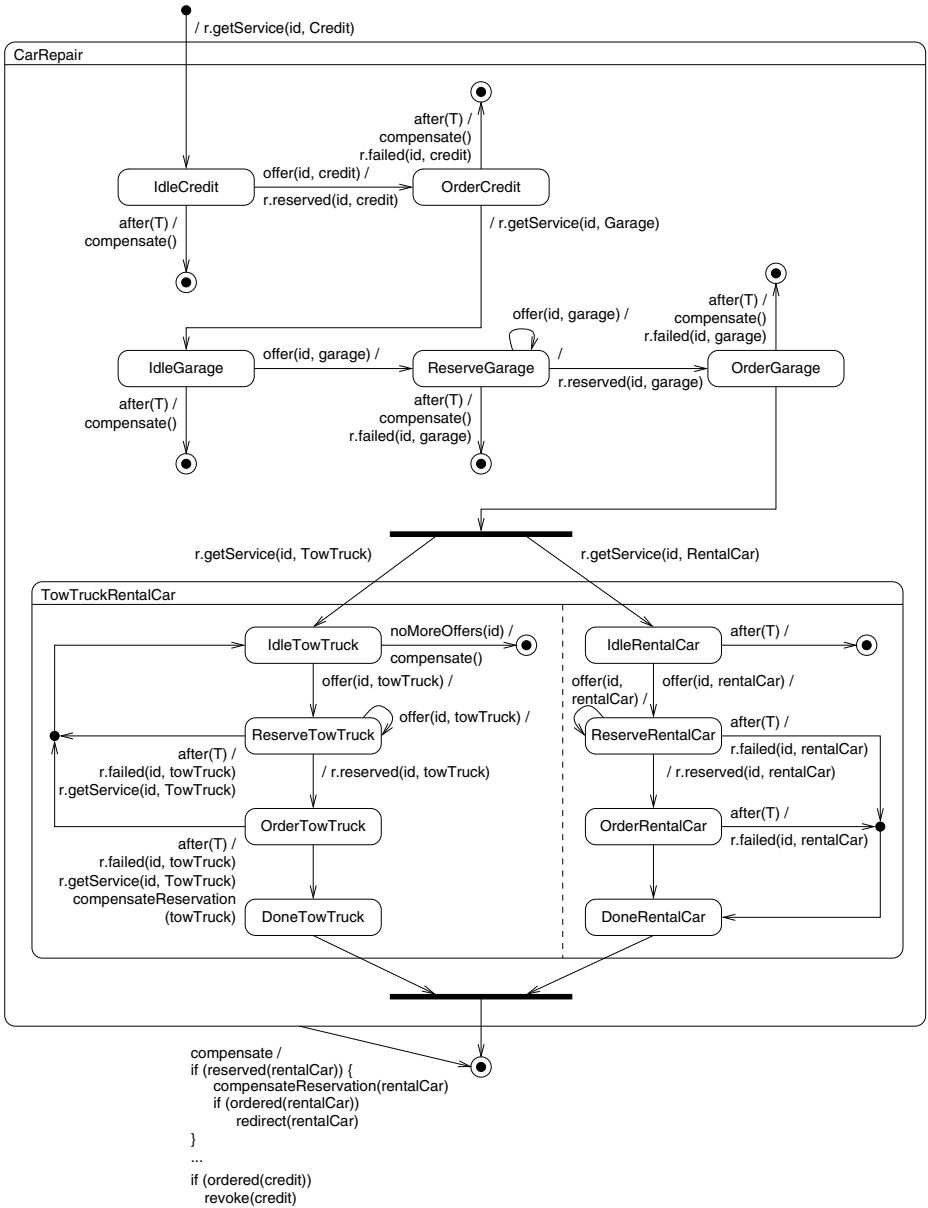


Fig. 13. Implementation state machine for car repair workflow

when one of the final states inside CarRepair is entered—are compensated. After behaviour verification, we have used Hugo/RT for generating Java code from the very same model that has been verified by UPPAAL.

4.2 Quantitative Analysis of the Accident Scenario

In this section we consider the assessment of a service level agreement offered by an automotive collision support service. The scenario with which these systems are concerned is road traffic accidents and dispatch of medical assistance to crash victims.

The Choreographer design platform [14] (developed by the DEGAS project [24]) can perform quantitative analysis via PEPA which starts and ends with UML models. A PEPA model is extracted from a UML model decorated with rate information such as the one shown in Fig. 5. This model is compiled into a CTMC and solved for its steady-state probability distribution. The results from this analysis are reflected back into a modified version of the input UML model with the quantitative analysis results recorded on the state diagrams in the model. Here we use the more computationally expensive but more informative method of transient analysis of the underlying CTMC and focus on the analysis of the PEPA model.

We represent in the model the sequence of events which begins with the deployment of the airbag after the crash and finishes with the dispatch of the medical response team. The first phase of the sequence is concerned with relaying the information to the remote service, reporting the accident. When the diagnostic report from the car is received the service processes the report and matches it to the driver information stored on their database.

$$\begin{aligned} Car_1 &\stackrel{def}{=} (airbag, r_1).Car_2 \\ Car_2 &\stackrel{def}{=} (reportToService, r_2).Car_3 \\ Car_3 &\stackrel{def}{=} (processReport, r_3).Car_4 \end{aligned}$$

The second phase of this passage through the system focuses on the attempted dialogue between the service and the registered driver of the car. We consider the case where the driver does not answer the incoming call because this is the case which leads to the medical response team being sent.

$$\begin{aligned} Car_4 &\stackrel{def}{=} (callDriversPhone, r_4).Car_5 \\ Car_5 &\stackrel{def}{=} (timeoutDriversPhone, r_5).Car_6 \end{aligned}$$

The service makes a final check on the execution of the procedure before the decision is taken to send medical help. At this stage the driver is awaiting rescue.

$$\begin{aligned} Car_6 &\stackrel{def}{=} (rescue, r_6).Car_7 \\ Car_7 &\stackrel{def}{=} (awaitRescue, r_7).Car_1 \end{aligned}$$

We assess the service against the following compound service level agreement(SLA):

At least 40% of airbag deployments lead to medical help being sent within five minutes and at least 80% of airbag deployments lead to medical help being sent within ten minutes.

We assess this SLA using the passage-time quantile computation capabilities provided by the ipc/Hydra tool chain [11]. We vary rates r_2 to r_6 across five or six possible

values leading to $5 \times 5 \times 5 \times 5 \times 6 = 3750$ experiments to be performed. The graphs of computed probability against experiment number for time bounds of five minutes and ten minutes for all 3750 experiments are shown in Fig. 14. Using both of these graphs we determine that the SLA is met across the values of the rates of the model.

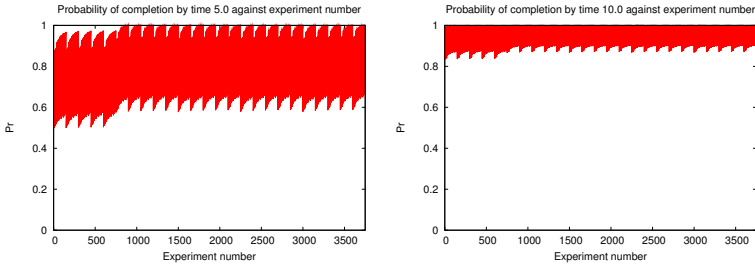


Fig. 14. Graph of probability of completing the passage from airbag deployment to medical assistance dispatch within five minutes and ten minutes plotted against experiment number over all 3750 experiments

We now consider how the cumulative distribution function for the passage from airbag deployment to dispatch of medical assistance is affected as the values of the rates r_2 to r_6 are varied as specified in the table in Fig. 6. The results for r_2 and r_6 are presented in Fig. 15.

These results show that variations in upstream rates (near the start of the passage of interest) such as r_2 , r_3 and r_4 have less impact overall than variations in downstream rates (near the end of the passage of interest) such as r_5 and r_6 . This is true even when the scale over which the upstream rates are varied is much more than the scale over which the downstream rates are varied (contrast variation in r_2 against variation in r_6).

The conclusion to be drawn from such an observation is that, if failing to meet a desired quality of service specified in an SLA then it is better to expend effort in making a faster decision to dispatch medical help (governed by rate r_6) than to expend effort in trying to transmit location data faster (governed by rate r_2), over the range of variability in the rates considered in the present study.

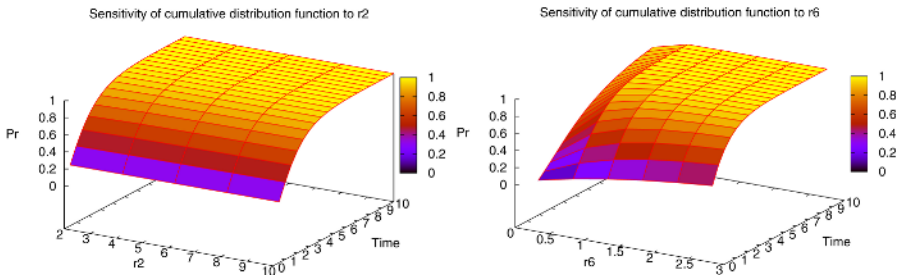


Fig. 15. Graphs of cumulative distribution function sensitivity to changes in rates for the passage from airbag deployment to dispatch of medical assistance

5 Concluding Remarks

In this paper we have presented some of the first results of the SENSORIA semantic-based development of service-oriented systems. We have shown service-oriented extensions to the UML, a first mathematical basis formed by process calculi such as PEPA and the saga calculus, a language for expressing soft constraints and preferences of services, qualitative and quantitative methods for analysing service orchestrations and service level agreements, and model transformations from UML to process calculi.

But these results represent only a small part of the SENSORIA project. In addition, the SENSORIA project is developing a comprehensive service ontology and a (SENSORIA) Reference Modelling Language (SRML) [16] for supporting service-oriented modelling at high levels of abstraction of "business" or "domain" architectures (similar to the aims of the service component architecture SCA [33]). To provide semantic foundations to the dynamic behaviour of services a new process calculus SCC [8] has been designed which features explicit notions of service definition, service invocation and session handling. Other research strands of SENSORIA comprise a probabilistic extension of a Linda-like language for service-oriented computing [12] and stochastic extensions of KLAIM [32] and beta-binders [15]. SENSORIA addresses security issues ranging from sandboxing for KLAIM [21], trust management for autonomic grid services [26], and security of service composition [4] to a formal framework for security and trust in the requirements phase of system development [20].

Moreover, SENSORIA is developing a model-driven approach for service-oriented software engineering and a suite of tools and techniques for deploying service-oriented systems and for re-engineering of legacy software into services. By integrating and further developing these results SENSORIA will achieve its overall aim: a comprehensive and pragmatic but theoretically well founded approach to software engineering for service-oriented systems.

References

1. The Attributed Graph Grammar System (AGG). tfs.cs.tu-berlin.de/agg. Last visited: June 2006.
2. András Balogh and Dániel Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proc. ACM Symp. Applied Computing (SAC 2006) — Model Transformation Track*, 2006. To appear.
3. Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Style-based Modelling and Refinement of Service-oriented Architectures. *Softw. Sys. Model.*, 2006. To appear.
4. M. Bartoletti, P. Degano, and Ferrari G.L. Security Issues in Service Composition. In *Proc. 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, LNCS, 2006.
5. Stefano Bistarelli. *Semirings for Soft Constraint Solving and Programming*. LNCS 2962. Springer, Berlin, 2004.
6. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
7. Microsoft BizTalk Server. www.microsoft.com/biztalk. Last visited: June 2006.
8. Michele Boreale, Roberto Bruni, Rocco DeNicola, Ivan Lanese, Michele Loreti, Ugo Montanari, Davide Sangiorgi, and Gianluigi Zavattaro. SCC: a Service Centered Calculus. Deliverable 2.2, SENSORIA, 2006.

9. Business Process Execution Language (BPEL). www.oasis-open.org. Last visited: June 2006.
10. BPEL for Web Services. www6.software.ibm.com/software/developer/library/ws-bpel.pdf. Last visited: June 2006.
11. Jeremy T. Bradley and William J. Knottenbelt. The ipc/HYDRA tool chain for the analysis of PEPA models. In *Proc. 1st Int. Conf. on the Quantitative Evaluation of Systems (QEST 2004)*, pages 334–335, Enschede, Netherlands, September 2004.
12. Mario Bravetti and Gianluigi Zavattaro. Service Oriented Computing from a Process Algebraic Perspective. *Journal of Logic and Algebraic Programming*, 2005. To appear.
13. Roberto Bruni, Hernan Melgratti, and Ugo Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'05)*, pages 209–220. ACM, 2004.
14. Mikael Buchholtz, Stephen Gilmore, Valentin Haenel, and Carlo Montangero. End-to-end integrated security and performance analysis on the DEGAS Choreographer platform. In I.J. Hayes J.S. Fitzgerald and A. Tarlecki, editors, *Proc. of the Int. Symposium of Formal Methods Europe (FM 2005)*, LNCS 3582, pages 286–301. Springer-Verlag, June 2005.
15. P. Degano, D. Prandi, C. Priami, and P. Quaglia. Beta-binders for biological quantitative experiments. *Proc. 4th Workshop on Quantitative Aspects of Programming Languages, QAPL '06, Electronic Notes in Theoretical Computer Science*, 2006. To appear.
16. José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. In *In Proc. 3rd International Workshop on Web Services and Formal Methods (WS-FM 06). 8-9 September 2006, Vienna, Austria, 2006*. To appear.
17. Frank Leymann. Web Services Flow Language, version 1.0. Specification, IBM, 2001. www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
18. Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 249–259, New York, 1987. ACM Press.
19. Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the Final Standard. In *Proc. Wsh. MetaModelling for MDA Workshop*, York, 2003. www.omg.org/docs/ad/03-08-02.pdf.
20. Paolo Giorgini, Fabio Massacci, and Nicola Zannone. Security and Trust Requirements Engineering. In *Foundations of Security Analysis and Design III - Tutorial Lectures*, volume 3655 of LNCS, pages 237–272. Springer-Verlag GmbH, 2005.
21. René Rydhof Hansen, Christian W. Probst, and Flemming Nielson. Sandboxing in myKlaim. In *The First International Conference on Availability, Reliability and Security, ARES 2006*, 2006.
22. Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
23. UML Model Translator for Model Checking (Hugo/RT). www.pst.ifi.lmu.de/projekte/hugo. Last visited: June 2006.
24. IST-FET Global Computing I Initiative Project DEGAS. www.omnys.it/degas/. Last visited: June 2006.
25. Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MODELS Satellite Events*, LNCS, pages 128–138. Springer, Berlin, 2005.
26. H. Koshutanski, F. Martinelli, P. Mori, and A. Vaccarelli. Fine-grained and history-based access control with trust management for autonomic grid services. In *Proc. of Internat. Conf. on Autonomic and Autonomous Systems (ICAS06), IEEE Computer Society*, 2006.
27. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In A.J. Field and P.G. Harrison, editors, *Proc. of the 12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, LNCS 2324, pages 200–204, London, UK, April 2002. Springer-Verlag.

28. Model Driven Architecture (OMG). www.omg.org/mda/. Last visited: June 2006.
29. Model Transformation Framework. www.alphaworks.ibm.com/tech/mtf. Last visited: June 2006.
30. Object Management Group (OMG). Unified Modeling Language: Superstructure, version 2.0. Specification, OMG, 2005. www.omg.org/cgi-bin/doc?formal/05-07-04.
31. Query/View/Transformation Specification Final Adopted Specification. www.omg.org/cgi-bin/doc?ptc/2005-11-01. Last visited: June 2006.
32. De Nicola R., Katoen J.P., Latella D., and Massink M. STOKLAIM: A Stochastic Extension of KLAIM. TR 2006-TR-01, ISTI, 2006.
33. SCA Consortium. Service Component Architecture, version 0.9. download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-sca/SCA_White_Paper1_09.pdf. Specification, 2005. Last visited: June 2006.
34. SPIN Model Checker. www.spinroot.com. Last visited: June 2006.
35. Model Checker for UML Statechart Diagrams. fmt.isti.cnr.it/umc/. Last visited: June 2006.
36. UPPAAL Tool Environment. www.uppaal.com. Last visited: June 2006.
37. Dániel Varró and András Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In Thomas Baar et al., editor, *Proc. 7th Int. Conf. Unified Modeling Language (UML'04)*, LNCS 3273, pages 290–304. Springer, Berlin, 2004.
38. Martin Wirsing, Grit Denker, Carolyn Talcott, Andy Poggio, and Linda Briesemeister. A rewriting logic framework for soft constraints. In *WRLA 2006, 6th International Workshop on Rewriting Logic and its Applications*, April 2006. To appear in ENTCS, 2006.
39. Web Services Choreography Interface (WSCI). www.w3.org/TR/wsci. Last visited: June 2006.
40. Web Service Description Language (WSDL). www.w3.org/TR/wsd1. Last visited: June 2006.