An Experience on Formal Analysis of a high-level graphical SOA Design

Maurice H. ter Beek Franco Mazzanti Aldi Sulova
ISTI–CNR, Via G. Moruzzi 1, 56124 Pisa, Italy
{terbeek,mazzanti,sulova}@isti.cnr.it

Abstract:
In this paper, we present the experience gained with the participation in a case study in which a novel high-level design language (UML4SOA) was used to produce a service-oriented system design, to be model checked with respect to the intended requirements and automatically translated into executable BPEL code.

This experience, beyond revealing several uncertainties in the language definition, and several flaws in the designed model, has been useful to better understand the hidden risks of apparently intuitive graphical designs, when these are not backed up by a precise and rigorous semantics.

The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow the full exploration of designs which otherwise risk to become simple to draw and update, but difficult to really understand in all their hidden ramifications. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using appropriate model transformation techniques. This is particularly valuable in the context of agile development approaches which are based on rapid and continuous updates of the system designs.

1 Introduction

Service-Oriented Computing (SOC) has emerged during the last decade as an evolutionary new paradigm for distributed and object-oriented computing [Pa07, SH05]. Services are autonomous, distributed, and platform-independent computational elements capable of solving specific tasks, ranging from answers to simple requests to complex business processes. Services need to be described, published, categorized, discovered, and then dynamically and loosely coupled in novel ways (composed, orchestrated) so as to create largely distributed, interoperable, and dynamic applications and business processes which span organizational boundaries as well as computing platforms. Their underlying infrastructures are called Service-Oriented Architectures (SOAs). Unlike any earlier computing paradigm, SOC is destined to exert a continuous influence on modern day domains like e-Commerce, e-Government, e-Health, and e-Learning.

We have actively participated in the IST-FET Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers), funded by the EU under the 6th framework programme's Global Computing initiative. SENSORIA has successfully developed a novel comprehensive approach to the engineering of service-oriented software

systems, in which foundational theories, techniques, and methods are fully integrated into a pragmatic software engineering process. We refer to [WH10] and the references therein for details on SENSORIA.

Within the context of SENSORIA the novel high-level graphical design language UML4SOA [Fo10, UML] has been defined with the aim of facilitating service-oriented system designs. The same notation has been used inside the project to specify the credit request scenario from the SENSORIA's Finance case study (see Sect. 2).

High-level graphical design notations are very intuitive and very efficient in describing the current structure and status of an ongoing software project. Especially if they are associated with automatic code generation / design verification features, they might play an interesting role inside an agile software development process as they can help in reducing the effort of evolution cycles, and they can help in maintaining the focus of the development at a level which is also understandable by the client. This might help the cooperation between the clients and the developers, promoting the rapid delivery of software and its regular adaption to the evolving requirements, in the spirit of the agile approach to software development [FH01, Mar02, SW07] .

The author's role in this case study was centered around the formal analysis of the original UML4SOA design. This goal required some kind of formalization of the UML4SOA semantics and the translation of the system design into a formal model suitable to undergo a model-checking phase. To this aim, we first translated the UML4SOA design of the credit request scenario by hand into a formal operational UMC model [BM10]. This provided us with many useful insights into an automatization of translating UML4SOA diagrams into UML statecharts, which finally resulted in a prototype of a translator (based on the standard Eclipse EMF format and its transformation capabilities). UMC [Be10, GM10, Ma09, UMC] is an on-the-fly model checker that allows the efficient verification of SocL formulae over a set of communicating UML state machines. The state machines are defined using the UML statecharts notation which has a standard presentation and semantics defined by the Object Management Group (OMG). SocL [GM10] is an event- and state-based, branching-time, efficiently verifiable, parametric temporal logic that was specifically designed to capture peculiar aspects of services.

Since the purpose of the case study was the just the experimentation with a novel design notation and with novel model transformation and verification approaches, in our case the software development process has been very informal and it did not have the rigorous structure of any kind of industrial software development method (whether agile or not).

Our experience, beyond revealing several uncertainties in the language definition, and several flaws in the designed model, has been useful to better understand the hidden risks of apparently intuitive graphical design notations, when these are not backed up by a precise and rigorous semantics; in this case, in fact, abstract designs risk to become simple to draw and update, but difficult to really understand in all their hidden ramifications. The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow to overcome the problem. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using appropriate model transformation techniques, and this is particularly valuable in the

context of agile development approaches which are based on rapid and continuous updates of the system designs.

This paper is organized as follows. In Sect. 2, we sketch the credit request scenario from SENSORIA's Finance case study. In Sect. 3, we briefly introduce the UML4SOA profile as developed in SENSORIA. Subsequently, we outline our translations from UML4SOA diagrams to UMC statecharts in Sect. 4 and in Sect. 5 . In Sect. 6 we present the lessons we learned from this formalization of UML4SOA designs into executable UMC code. Finally, we draw some conclusions in Sect. 7.

2 Finance Case Study: The Credit Request Scenario

The credit request scenario from SENSORIA's Finance case study was provided by one of SENSORIA's industrial partners, S&N [S&N], which is a leading IT company of the financial services industry. The scenario is specified in UML 2.0 by using the profiles SoaML [OMG], which defines a metamodel for designing the structural aspects of SOA, and UML4SOA [Fo10, UML], which defines a high-level domain-specific modeling language for behavioral service specifications (see Sect. 3).

The scenario's main services are CreditRequest and—to a lesser degree—Rating. These are complemented with web services CustomerManagement, SecurityAnalysis, Balance-Analysis, and RatingCalculator that serve to interact with clients, save the information they provide, calculate ratings, and the like. The CreditRequest service describes a bank service that offers clients the possibility to ask for a loan and subsequently orchestrates the steps needed to process this request, which may involve an evaluation by a clerk or a supervisor before a contract proposal is sent to the client. The Credit Request service is depicted in Fig. 1. The scopes **Initialize** and **Finalize** handle a client's login and log off.

The loan workflow is represented in the scope **Main**, whose initial activity is to service the request. The **Creation** scope starts with a call from the Portal. The data involved is stored in the CreditManagement service and a confirmation is sent to the Portal. In case of a fault, the compensation handler removes the request from the CreditManagement service. The retrieval of the client's balances and securities is handled by the **Handle-Balance&SecurityData** scope and these are stored in the Balance and Security services. Upon completion of the request, the Rating service calculates the rating of the client's request after being asked to do so through the **RatingCalculation** scope.

The Rating service calculates a rating, which implies whether the request can be accepted automatically or a clerk or a supervisor needs to decide this. The **Decision** scope takes care of this. Rating AAA automatically leads to an offer to the client. In any other case, the **Approval** scope is used for a decision by an employee, based on which an offer or a decline is generated, according to the **Accept** and **Decline** scopes. Rating BBB means that the request has some risk, but can be decided by a clerk, while CCC indicates a much more risky request that needs a supervisor to decide. The decision is sent to the Portal. An offer is saved in the CreditManagement service and sent to the Portal, allowing the client to see it and decide to accept or decline it through interaction with the Portal. In case of a

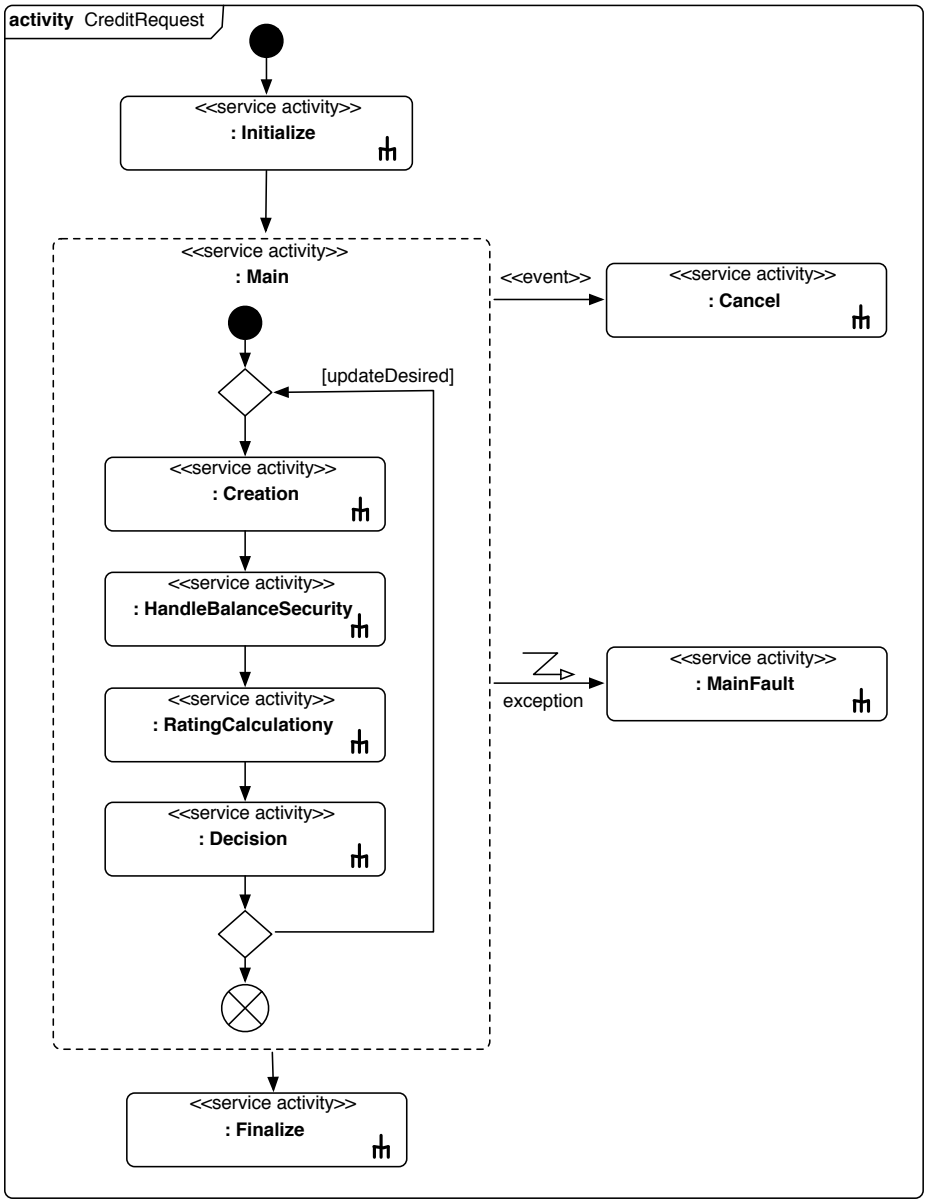

Figure 1: Service Credit Request.

decline, the client is allowed to update the data and restart the request.

At any moment, the client may want to abort the request, in which case the request data needs to be deleted. This requires the execution of compensation activities to semanti-

cally rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests.

3 UML4SOA: A UML Profile for Service Behavior

The *de facto* industrial standard for specifying and documenting software systems, UML, is not expressive enough for modeling structural and behavioral aspects of SOA. SOC introduces key concepts (e.g. partner services, message passing among service requesters and service providers, compensation of long-running transactions) that require specific support in the modeling language to avoid diagrams to become overloaded with technical constructs that reduce their readability.

There are two ways to extend UML to provide a domain-specific high-level design language: One can either change the UML metamodel to create new UML metaclasses or apply a user-defined UML profile to the model. In SENSORIA, the latter approach was chosen, mainly because profiles are easy to use and constitute a lightweight extension. Hence a UML 2.0 profile for modeling the *behavioral* aspects of SOA, called UML4SOA [Fo10], was designed. UML4SOA complements the SoaML profile [OMG], which defines a metamodel for designing the *structural* aspects of SOA. UML4SOA is a minimal extension built on top of the Meta Object Family (MOF) metamodel (i.e. new modeling elements are defined only for specific service-oriented features).

A UML profile consists of a set of stereotypes and constraints (specified in OCL). A stereotype is a limited kind of metaclass that cannot be used by itself, but only in conjunction with one of the metaclasses it extends. A metaclass is a class whose instances are classes. A metaclass defines the behavior of classes and their instances, much like a class in object-oriented programming defines the behavior of certain objects. Each stereotype may extend one or more classes through extensions as part of a profile. All UML modeling elements may be extended by stereotypes (e.g. states, transitions, activities, use cases, components). Stereotypes of the UML4SOA profile can thus be used to enrich UML models with service-oriented concepts, including structural and behavioral aspects of SOA as well as non-functional notions. Constraints allow for a more precise semantics of the newly introduced modeling elements.

The structure of a SOA can be visualized by UML structure diagrams, showing the interplay between components, their ports, and their interfaces. UML4SOA 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, which provides the basis for the definition of UML4SOA. For each class of the metamodel, a stereotype is defined and relationships are expressed by constraints.

A key aspect of service orientation is the ability to compose existing services, i.e. creating a description of the interaction of several (simpler) services, which is known as

orchestrating. An orchestration is a behavioral specification of a service component, or ≪participant≫ in SoaML. UML4SOA proposes the use of UML activity diagrams for modeling service behavior, in particular for orchestrating (see, e.g., the orchestration in Fig. 1 and the UML4SOA diagrams in Sect. 6). A UML4SOA stereotype ≪serviceActivity≫ can be directly attached as the behavior of a ≪participant≫. The focus is on service interactions, long-running transactions, and their compensation and exception handling.

A scope is a structured activity that groups actions, which may have associated compensation and exception handlers (see, e.g., Fig. 10). Scopes and the corresponding handlers are linked by specific compensation and exception edges (UML4SOA stereotypes ≪compensation≫ and ≪event≫).

Scopes include stereotyped actions like ≪send≫, ≪receive≫, ≪send&receive≫, ≪reply≫, ≪compensate≫, and ≪compensateAll≫. The stereotype ≪send≫ is used to model the sending of messages; ≪receive≫ models the reception of a message blocking until the message is received; ≪send&receive≫ is a shorthand for a sequential order of a send action and a receive action, and ≪reply≫ accepts a return value produced by a previous receive action. Container for data to be send or received are modeled by ≪snd≫ and ≪rcv≫ pins, while ≪lnk≫ pins refer to the partner service in interactions. Long-running transactions, like those provided by services, require the management of compensation. Therefore, UML4SOA contains ≪compensate≫ and ≪compensateAll≫ 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. The graphical notation of the UML4SOA stereotyped elements is shown in Fig. 2.

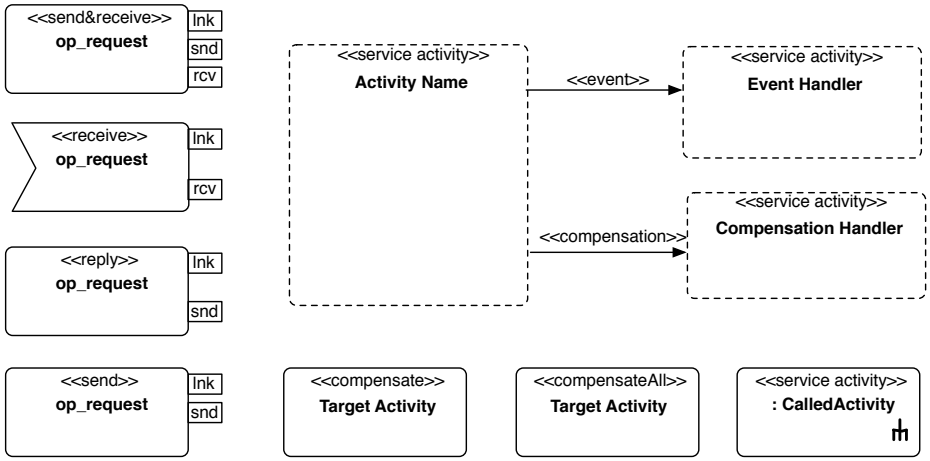

Figure 2: UML4SOA: new graphical elements.

4 Translating UML4SOA Diagrams into UMC Statecharts

Appended to [BM10], we provided a UMC model of the credit request scenario described in Sect. 2. This model was obtained by translating the specification provided by S&N. In the specification discussed in Sect. 2, only the CreditRequest and Rating services are described as UML4SOA activity diagrams, while the other components used by these services (the Portal, Credit Management, Customer Management, Security Analysis, Balance Analysis, and Rating Calculator services) are described by UML4SOA protocol state machines. Since UMC needs a closed model, we moreover modeled a single Client (interacting with a single instance of the Portal) that can make up to two credit requests.

The translation of the main CreditRequest and Rating services was performed in a rather general way, in order to be able to serve as a blueprint for an automatic translation tool from UML4SOA specifications into UMC code (see Sect. 5). This explains the at times long-winded organizational structure. The UMC encoding of the other components, originally specified by protocol state machines, was done by hand.

It is outside the purpose of this paper to try to present all details of the rationale and the rules according to which the UML4SOA activity diagrams were translated into UML statecharts. We just hint here that in general a UML4SOA activity diagram is mapped into a corresponding UML statechart with a single state. Each progression step during the execution of a UML4SOA activity diagram is modeled by one (or sometimes more) transitions in the statechart model, which implement the corresponding semantics. In general, the UML statechart transitions modeling the execution of an activity node have the following form, where tau represents an internal signal of the statechart:

```
s1 -> s1 -- actually no state change
 { tau [enabling conditions for incoming edges] /
       resetting of enabling conditions;
       execution of specific node activity;
       setting of enabling conditions for outgoing edges;
       self.tau;
 }
```

As a specific example, we consider the ≪send≫ node shown in Fig. 3. The corresponding specific UMC transition rule modeling the execution of this node is as follows:

```
s1 -> s1
 { tau [N1_Finalize_DefaultInitialOUT = true] /
       N1_Finalize_DefaultInitialOUT := false;
       LNK_portalService.goodbye([self],
         VAR_CreditRequest_customerData);
       N2_Finalize_Send_goodbyeOUT := true;
       self.tau;
 }
```

Since UML4SOA is a high-level modeling language, some details of the specification

are voluntarily underspecified (oftentimes inherited from UML). This results in the need for specific assumptions to accompany any implementation of a UML4SOA design in a lower-level formalism, so as to make certain details of the model explicit.

UMC offers users the following specific possibilities to customize their system designs:

1. By default, messages that arrive at a time at which they cannot be accepted are never discarded. This specific choice can be reversed by simply avoiding to declare as "Deferred" the corresponding events.

2. By default, the order in which events are removed from an events queue associated with a state machine is "FIFO" (First In First Out). This specific choice can be reversed (even on a *per-object* basis) by simply changing the object queue policy to "RANDOM" (i.e. any queued event is eligible for being dispatched, independently of its position in the queue).

3. By default, all objects are assigned the same priority (thus modeling the maximal degree of nondeterminism in their scheduling). This specific choice can be reversed by simply assigning (also dynamically) to each entity its own priority, thereby modeling more specific schemata.

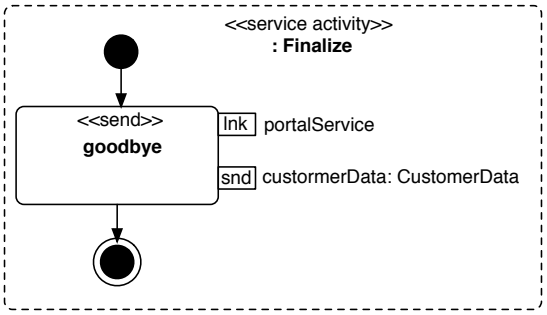

Figure 3: A ≪send≫ node from scope **Finalize**.

5 Tool Support: From MagicDraw to Eclipse to UMC

In this section, we describe the techniques and frameworks we used to support an automatic transformation from UML4SOA designs into UMC executable code. This transformation is based on our translation experience described in the previous section and on OMG's Model Driven Architecture (MDA). MDA is a software development paradigm that focuses the development process more on system design than on its implementation. The main target of this approach is to create models in an efficient and domain-specific way. This is realized by using domain-specific languages and generating the software from these models.

In our context, MDA is used not only to provide a fully automatic approach for transforming UML4SOA service-oriented orchestrations into an implemented system, but also to support qualitative and quantitative verifications by third-party tools. For this purpose we choose the Eclipse Modeling Framework (EMF) and Xpand. EMF is an open source platform that has widely adopted the MDA paradigm and implements most of its core specifications, while Xpand is an EMF-related template language (supported by openArchitectureWare [oAW]) based on domain-specific models for model-to-text transformations. The reason for choosing EMF is threefold:

1. The UML 2.0 core metamodel is already implemented and distributed as part of the Eclipse UML 2.0 plug-in. Hence, there is no need to define a new UML4SOA metamodel, but we only need to consider the stereotypes defined in UML4SOA.
2. MagicDraw can export a UML4SOA design as an Eclipse UML 2.0 instance model.
3. A new graphical interface for UMC, which allows one to directly draw a UMC model in terms of graphical nodes and edges, was already experimented in the context of the Eclipse environment [Su09].

Figure 4 shows the steps performed to obtain UMC code from a UML4SOA system design.

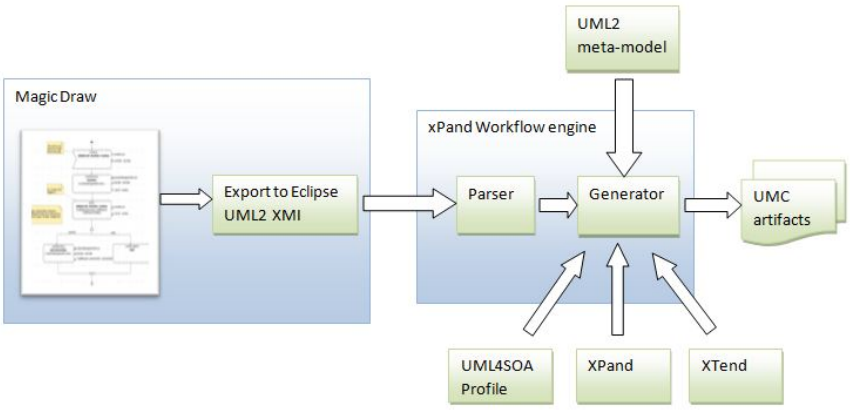

Figure 4: From a UML4SOA system design to UMC executable code.

The Xpand workflow engine executes the transformation process by first invoking a DOM-like parser to create an object graph in memory for our model, and then passing this graph to the Xpand generator in order to get the UMC artifacts. A generator usually consists of one Xpand template file controlling the output generation and a list of Xtend files (another domain-specific language, part of Xpand) containing other user-defined independent functions. An exemplary excerpt from an Xpand template file is given in Fig. 5.

The central concept of Xpand is the template declared using the DEFINE... ENDDEFINE block, which contains the basic structure of the transformation process. A template has a name (UML4SOA2UMC in our example) and is bound to a specific metatype (uml::Model in our example). Inside the main template, a FILE... ENDFILE block creates a new file

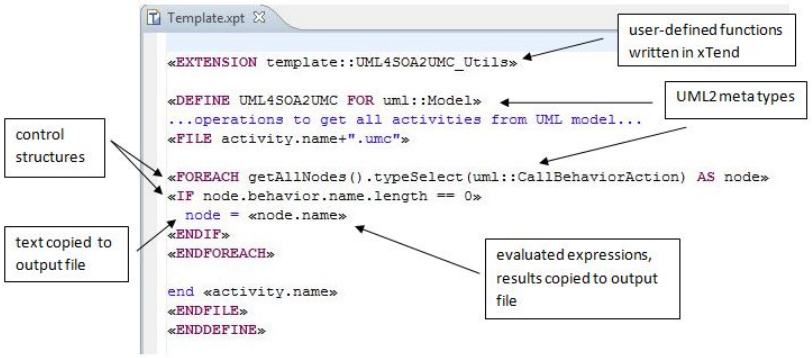

Figure 5: Example of an Xpand template.

with the specified name. All code generated inside that block is written to the created output file. Control constructs such as FOREACH and IF are provided as well.

As a specific example of the transformation from UML4SOA to UMC we consider a UML4SOA ≪send≫ node. The corresponding Xpand template code is shown in Fig. 6. This transformation mimicks the manual translation described in the previous section.

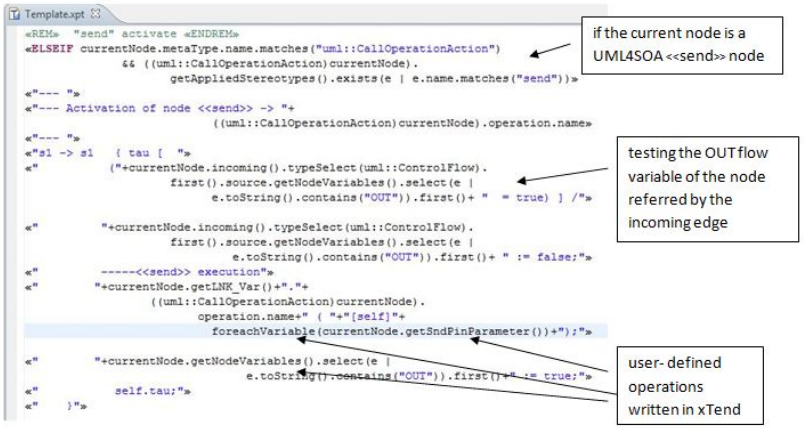

Figure 6: Example transformation of a UML4SOA ≪send≫ node.

6 Lessons Learned from the UMC Formalization Experience

A first immediate advantage of the UMC formalization of the credit request scenario from SENSORIA's Finance case study is the possibility to manually explore the graph of all possible system evolutions, verifying through inspection the correctness of the design and the roots of possible anomalies.

In our case, the statespace was not particularly large ($39,530$ states) and even if a small exploration of the initial steps were sufficient to identify some problematic aspects of the design, it is still too large to allow a complete manual inspection.

Model checking SocL formulae proved to be a powerful method for identifying unexpected violations of "supposed-to-be-true" properties, and the detailed explanations provided by UMC (in terms of counterexample/witness paths through the graph of all possible system evolutions) allowed a quick understanding of how such violations might occur. It is outside the purpose of this paper to show the details of the logic (SocL [GM10]) used to analyze our system model of the UML4SOA system design of the credit request scenario. Two exemplary verified properties, here just expressed in plain natural language, are as follows:

1. *"It is always true that if the system accepts a new CreditRequest, then it will either respond to the Client or it will receive a cancel request."*
2. *"If a negative response to a CreditRequest is sent back to the Client, then the rating evaluation was not AAA."*

In the following subsections, we will illustrate in more detail some of the traps and pitfalls that we found in the original system design in UML4SOA as well as inside the UML4SOA definition, and we will show what we believe to be the real underlying issues reflected by them. We will do so abstracting a little from the specific details of the case study, presenting instead small self-contained examples which are not overwhelmed by actually not relevant details.

6.1 Hidden implementation-dependent assumptions inside "high-level" "platform-independent" designs

Suppose we have a Client / Server architecture where the Client performs a login request followed (if successful) by an operation request, while the Server waits for the login request and, if the supplied login data is OK, waits for an operation request to be handled. Using UML4SOA, this simple design can be represented by the diagram shown in Fig. 7 and Fig. 8.

If we adopt a "flat" design for the description of the Server side of this activity (see Fig. 8, left diagram) the semantics is quite clear. However, if we use some additional structured activities inside the Server (see Fig. 8, right diagram) in order to explicitly model the existence of some login and operation mode phases, some semantic subtleties may arise. This is very similar to what happens in our case study, e.g., w.r.t. the **Initialize** and **Main** scopes inside the CreditRequest service.

In the first case (a flat design), once the login response is sent by the Server, the subsequent receive activity can immediately be enabled, while in the second case the two events (completing the send and enabling the receive) are no longer coinciding, since inbetween there is the step of completing the **Login_Phase** and entering the **Operation_Phase**. In principle it might happen that the Server receives an **operation request** before the **Operation_Phase** is entered (and before the receiving of the request is enabled). This makes the

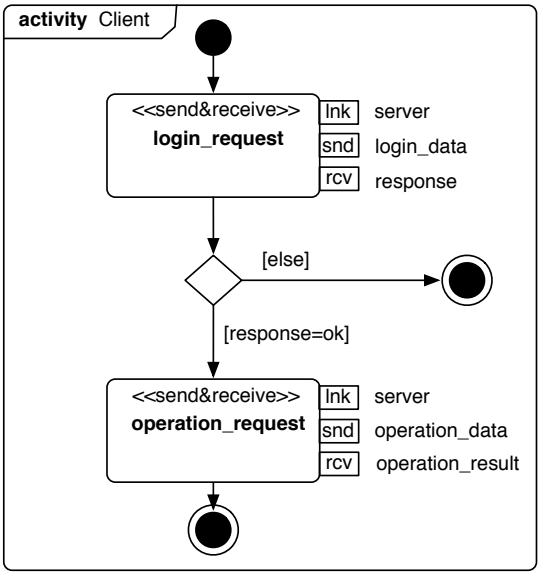

Figure 7: UML4SOA diagrams for the Client.

semantics of the design very implementation dependent as it depends on what is supposed to happen when a message arrives at a system component when the component is not yet ready to accept it (e.g. the request might be queued on the Server until it can be accepted or it might be discarded), and if such a situation should indeed be considered possible.

The problem is that the designer is probably making some *implicit* underlying assumption.

One of these assumptions might be that the time needed by the (remote) Client to receive the login response and produce a new operation request, is much higher than the time needed by the Server to perform all its internal steps from the time at which the response is provided to the time at which the subsequent receive becomes enabled. This kind of assumption might be reasonable, but should be appropriately reflected by the system design, e.g. by stating explicitly that the expected behavior of the Server component is to run as an indivisible activity until it reaches a subsequent communication action.

An alternative assumption, on the contrary, might rely instead on the fact that incoming operation request messages are implicitly supposed not to be discarded, but to be always queued until possibly eventually accepted.

If we do not make any assumption and simply generate a UMC formal model corresponding to the "structured" Server design we can easily detect the presence of deadlocks in the system, e.g. by checking the property that a successful login request is always eventually followed by a response to the operation request. In this way, we can become aware of the existence of some problem in the design. Once the modeling difficulty is understood, we can fix the missing underlying assumptions by explicitly stating one (or both) of the

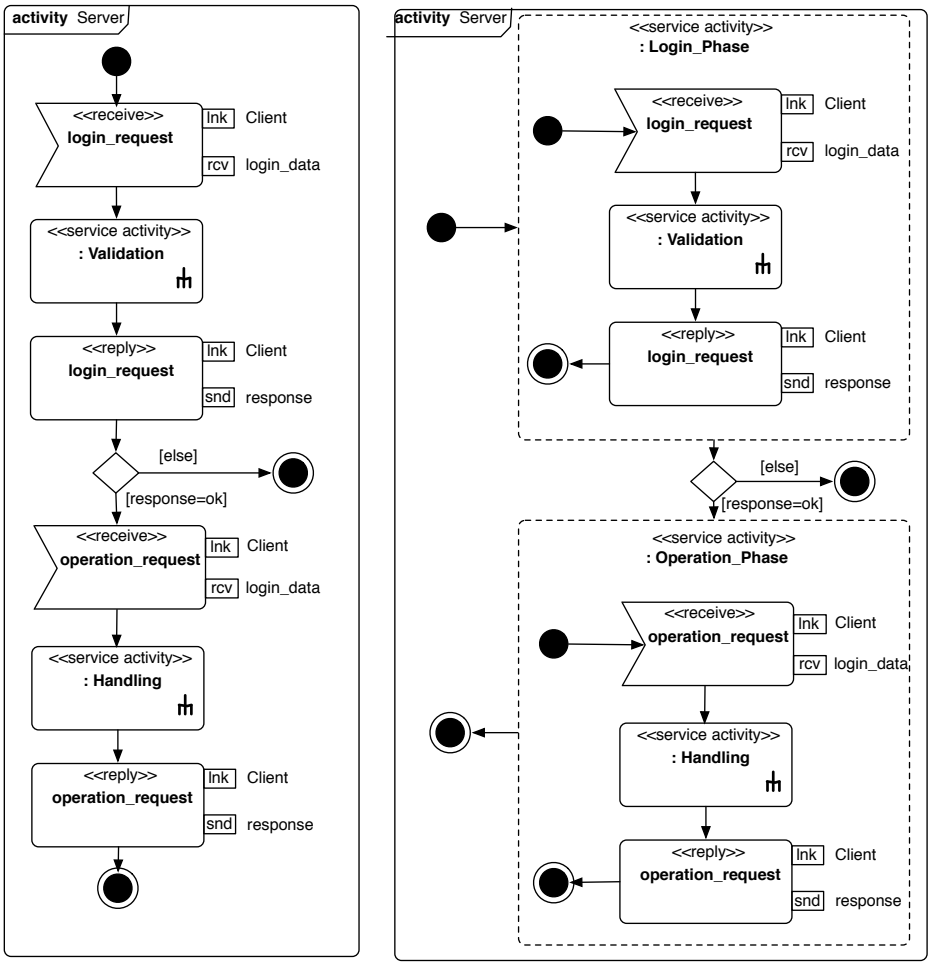

Figure 8: UML4SOA diagrams for the Server (both a flat and a structured design).

possible correct modeling strategies, and these strategies can be efficiently encoded in our UMC model. The case of desired greater granularity of the required Server behavior can be supported in UMC by exploiting the dynamic priorities feature (i.e. raising the priority of a system component for a short period of time in order to execute a sequence of internal steps as a unique indivisible activity). The case of incoming messages not supposed to be discarded even if arriving at a time in which a component is not yet able to handle them, on the other hand, is supported by UMC by allowing to define the corresponding events as "deferred" (in the UML statecharts sense) to achieve the behavior of queued messages. Once any (or both) of these solutions is adopted we can verify that the system behavior becomes again, as intuitively expected, equivalent to that of the flat model of the Server.

These problems can be exacerbated by a third kind of hidden assumption, shown in Fig. 9.

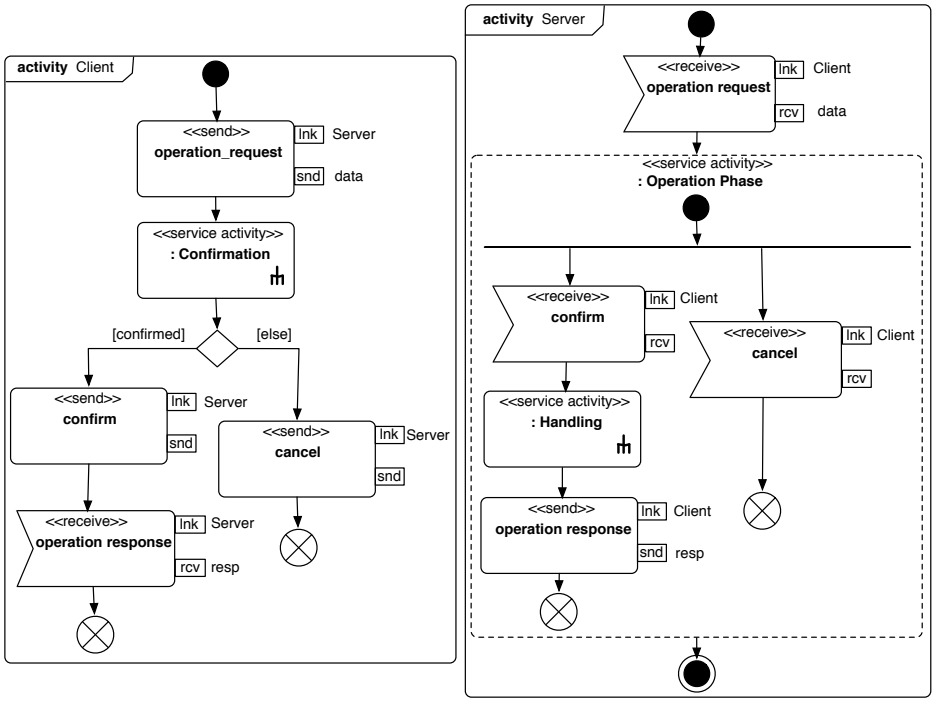

Figure 9: UML4SOA diagrams for the Client and the Server — second variant.

In the variant depicted in Fig. 9, the Client is supposed to ask the Server for a certain operation, with the possibility to later cancel or confirm this operation. Similarly, the Server is initially supposed to wait for an operation request, after which an event **Operation Phase** is entered and, depending on the Client's decision, the appropriate activity is performed. The design of Fig. 9 seems correct at a first glance, but what happens if the **operation request** message is delayed by the network and happens to be delivered after the **confirm** or **cancel** message? Again, this is a strictly implementation-dependent situation (due to UML allowing so-called semantic variation points) and if the implementation is based on the assumption that wrongly timed messages are discarded, then the system deadlocks. Also in this case we have an apparently correct "platform-independent" "high-level" design which, on the contrary, just hides specific implementation-dependent assumptions.

When building a UMC model, the appropriate choice of modeling a communication network that can or cannot deliver the messages in a different order from the one in which they are sent, can be easily made by specifying a specific queueing policy (RANDOM versus FIFO) for the events arriving to the system components.

6.2 Uncertain semantics of UM4SOA features

Informally describing a language feature is one thing, formally reasoning on all the possible consequences of a language choice is quite another thing. This could be the summary of this subsection. This time the sample culprit is the design of the compensation mechanism of UML4SOA.

UML4SOA allows one to associate to a service activity region a ≪compensation≫ edge leading to another compensation service activity, to be executed whenever the effects of a successful completion of the original service activity have to be undone in some way. This compensation activity is requested by the execution of a ≪compensate≫ or ≪compensateAll≫ action. The main problem here is that compensation handling is a concept that is tightly coupled to that of an "atomic transaction" (i.e. an activity with an "all or nothing" semantics), but UML4SOA service activities are not required to have such a transactional semantics.

In the credit request scenario, there are several cases in which non-transactional activities are associated to compensation handlers, with the consequence that some expected system property do not hold. This typical example is shown (in a form equivalent to what happens e.g. inside the **Creation** activity of the case study) in Fig. 10.

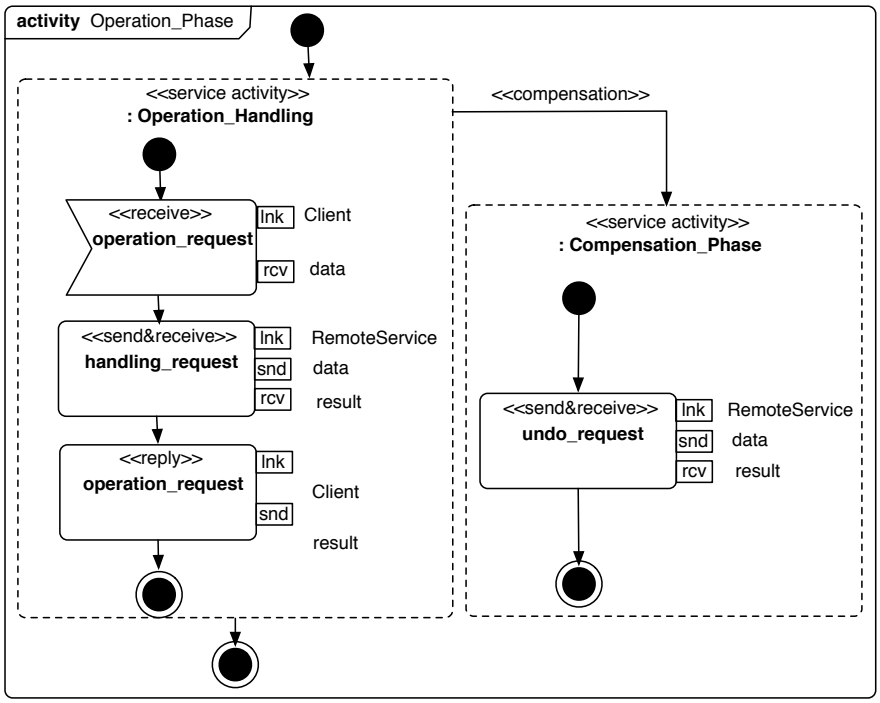

Figure 10: UML4SOA compensation.

In Fig. 10, the compensation activity consists of requesting, with the **undo_request** ser-

vice call, the "undo" of the **handling_request** service call executed inside the **Operation_Handling** activity. This logically means that "orphan" (i.e. not associated to successful credit requests) **handling_request** instances should not be present in the system. However, if a compensation request is executed after the **handling_request** service call is issued but before the **Operation_Handling** activity is completed, then no compensation is activated and an orphan **handling_request** instance still exists.

In this case, at least two kind of problems are raised and evinced by the formal modeling:

1. The semantics of compensation should probably be tied with an atomic transaction mechanism, a fact that is not sufficiently well described by the current UML4SOA definition of service activities and compensations.

2. The designer of this fragment of activity might have implicitly assumed that the execution of the **Operation_Handling** activity cannot be interrupted from the outside. Since it contains no error-path, it is supposed to always complete successfully. Unfortunately, the overall system design allows two parallel threads to asynchronously interfere with one another, and at first the impact of this possibility was not well understood in the case study. In the credit request scenario, the asynchronous interfering activity is the one originated by a **cancel** operation triggered by the Client.

Another subtle and potentially problematic aspect of the UML4SOA semantics of compensation, is related to the order in which the subactivities of a given activity have to be compensated. While UML4SOA requires that subactivities need to compensated in precisely the reverse order w.r.t. their completion order, this is not exactly what is required by the BPEL semantics (which is more lazy and allows violation of completion ordering for not causally related subactivities). In our case study, this difference is absolutely irrelevant, but in principle more complex designs could be imagined in which the difference does become observable.

If we suppose that software development approach relies on an automatic translation from UML4SOA into BPEL, directly mapping UML4SOA compensations into BPEL compensations, then for more complex designs it might happen that an apparently working system from a certain point onwards starts to show unexpected anomalies. This might happen, e.g., when the adopted BPEL execution engine passes from one version to another, or from one vendor to another. In this case, formal methods might be of help by rigorously defining the assumptions and the meaning of the high-level designs. However, they can do little w.r.t. to the verification of compatibility issues of BPEL execution engines, and even less w.r.t. the inconsistencies raised by the evolutions in time of the BPEL specification itself.

During the formalization of the credit request scenario from SENSORIA's Finance case study, several other aspects have raised discussions and clarification requests among the different partners involved in the project. The semantics of UML4SOA protocol state machines that are used to model the unspecified components of the system, e.g., was found not well specified, nor was the semantics of service instance creation and service connection establishment. For all these aspects, the formal modeling effort offered a good opportunity to clarify and disambiguate the intended meaning of the language features, a

process which is probably still not complete.

6.3 Hidden complexity of scarcely structured designs

For several decades now, the danger of using "goto" as a control flow command has been widely recognized in the field of software engineering. This is due to the difficulty of analysis and understanding that its use introduces in the algorithms. It is therefore hard to understand why nowadays, when designing "high-level" "platform-independent" design languages, we resort to recycling the goto construct just because with a graphic design notation, nodes and edges are the easiest graphical elements to design. The result can quickly become a "spaghetti design" (as shown in the leftmost diagram of Fig. 11), with the consequence that the true behavior of the system becomes obscure and difficult to understand in all its ramifications.

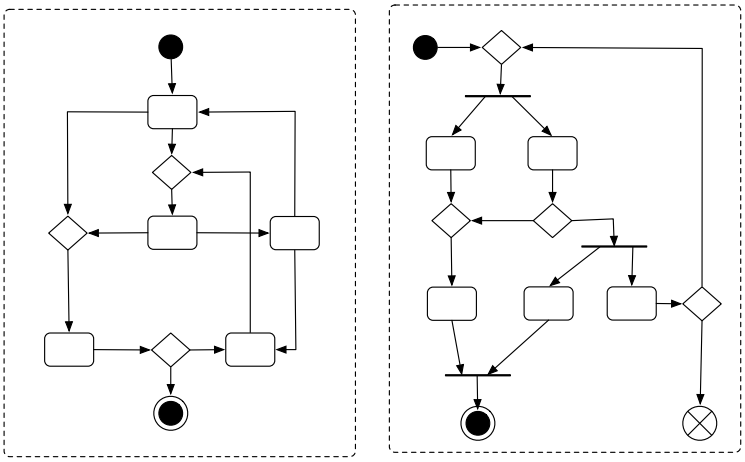

Figure 11: Weren't gotos considered harmful?

Moreover, in almost all programming languages great care has been given to the introduction of concurrent features (task, threads, co-routines) with the design goal of well identifying the concurrently evolving activities and, above all, keeping the flow of concurrent elements independent as much as possible. We repeat that it is therefore hard to understand that "high-level" design languages resort to the use of low-level graphical elements, like "fork" and "join", to handle concurrency, thus potentially allowing an even nastier form of spaghetti design (as shown in the rightmost diagram of Fig. 11). The situation is even made potentially more dangerous by exploiting elements like "activity final" nodes (killing all concurrent subactivities inside a parallel activity) or by raising exceptions, which might have the consequence that an error in one concurrent subactivity asynchronously terminates other concurrent activities in an unclear, implicit, or uncontrolled way.

Fortunately, in our case study we did not have to face such an abuse of unstructured constructs. However, we cannot avoid to remark that one of the design flaws present in the

service orchestration (as shown in the previous subsection) was in fact caused by a bad interference between two concurrent activities. The overall situation actually occurring in our case study is like that shown in Fig. 12, in which the **Operation_Phase** is unexpectedly aborted because of an exception raised inside the **Cancel** activity, asynchronously triggered as a concurrent flow by an external event.

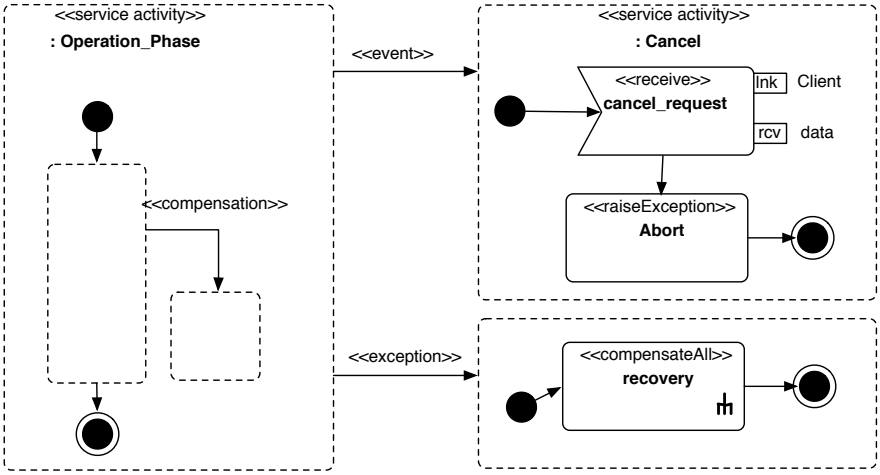

Figure 12: UML4SOA compensation — second variant.

7 Conclusions

High-level graphical design notations are very intuitive and very efficient in describing the current structure and status of an ongoing software project. Especially if they are associated with automatic code generation / design verification features, they might play an interesting role inside an agile software development process as they can help in reducing the effort of evolution cycles, and they can help in maintaining the focus of the development at a level which is also understandable by the client. This might facilitate the cooperation between the clients and the developers, promoting the rapid delivery of software and its regular adaption to the evolving requirements, in the spirit of the agile approach to software development.

In this paper we show, however, that they may also be a source of problems, most of which the use of formal methods can avoid. We illustrate our claim by generalizing examples that we encountered in a case study. The three types of problems we focus on are hidden implementation-dependent assumptions inside “high-level” “platform-independent” designs, uncertain semantics of features of “high-level” design languages, and hidden complexity of scarcely structured designs.

It is our firm belief that high-level graphical design notations always need to be backed up by a precise and rigorous semantics, i.e. by formal methods, especially when they are

planned to be used inside agile software processes. The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow to explore and understand in all their hidden ramifications the high level designs. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using advanced model transformation techniques, and this is particularly valuable in the context of agile development approaches which are supposed to exploit the rapid and continuous updates of the system under development.

Acknowledgements

We thank all our partners in SENSORIA for detailed discussions of the Finance case study, but in particular Jannis Elgner from S&N, Philip Mayer and Martin Wirsing from LMU, Lucia Acciai, Federico Banti, Francesco Tiezzi and Rosario Pugliese from DSIUF, and Stefania Gnesi from ISTI.

References

[Be10] M.H. ter Beek, A. Fantechi, S. Gnesi and F. Mazzanti, A state/event-based model-checking approach for the analysis of abstract system properties. To appear in *Science of Computer Programming*, 2010.

[BM10] M.H. ter Beek and F. Mazzanti, Modelling and Analysing the Finance Case Study in UMC. Technical Report 2010-TR-007, ISTI–CNR, 2010.

[FH01] M. Fowler andJ. Highsmith, The Agile Manifesto. Software Development, Augist 2001. (see also http://www.agilemanifesto.org)

[Fo10] H. Foster, L. Gönczy, N. Koch, P. Mayer, C. Montangero and D. Varró, UML Extensions for Service-Oriented Systems. In [WH10], 2010.

[GM10] S. Gnesi and F. Mazzanti, An Abstract, on the Fly Framework for the Verification of Service Oriented Systems. In [WH10], 2010.

[Ma09] F. Mazzanti, Designing UML models with UMC. Technical Report 2009-TR-43, ISTI–CNR, 2009.

[Mar02] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR Upper Saddle River, NJ USA, 2002

[OMG] Object Management Group, Service oriented architecture modelling Language (SoaML): Specification for the UML Profile and Metamodel for Services (UPMS), April 2009. `http://www.omg.org/cgi-bin/doc?ptc/09-04-01/`

[oAW] openArchitectureWare. `http://www.eclipse.org/workinggroups/oaw/`

[Pa07] M.P. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer* 40, 11 (2007), 38–45.

[SH05] M.P. Singh and M.N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005.

[Su09] A. Sulova, Model Driven Software Development con Eclipse, StatechartUMC. In *Proceedings of the 4th Italian workshop on Eclipse technologies (Eclipse-IT 2009), Bergamo, Italy* (A. Gargantini, Ed.), Eclipse Italian Community, 2009, 113–114. An extended version appeared as Technical Report 2009-TR-050, ISTI–CNR, 2009. In Italian.

[S&N] S&N AG. `http://www.s-und-n.de/`

[SW07] J. Shore and S. Warden, The art of agile development. OReilly, 2007.

[UMC] UML Model Checker. `http://fmt.isti.cnr.it/umc/`

[UML] UML4SOA. `http://www.uml4soa.eu/`

[WH10] M. Wirsing and M. Hölzl (Eds.), *Rigorous Software Engineering for Service-Oriented Systems: Results of the* SENSORIA *project on Software Engineering for Service-Oriented Computing*, Springer, 2010. To appear. See also `http://www.sensoria-ist.eu/`