Automated Simulation of Communication Protocols Modeled in UML 2 with Syntony

Isabel Dietrich, Christoph Sommer, Falko Dressler, and Reinhard German

Computer Networks and Communication Systems
University of Erlangen-Nürnberg, Germany
{isabel.dietrich,christoph.sommer,dressler,german}@informatik.uni-erlangen.de

Abstract. This paper describes *Syntony*, an Eclipse-based framework that we developed for automated and tool-assisted development and analysis of network protocols. With the help of *Syntony*, we are able to use a simple graphical modeling language to describe complex protocols. In particular, we use UML 2 diagrams to construct simulation models to be executed in an event-driven simulation framework (currently, we are using OMNeT++). The translation of the UML 2 models is directly provided by *Syntony*. In addition to the use of standardized graphical modeling languages for developing and evaluating simulation models, the complete process of debugging and analyzing the protocol is tool-assisted. For verification purposes, we developed an UML 2 model of the communications within the AKTIV project Cooperative Cars (CoCar). We were able to demonstrate that *Syntony* can be used to generate executable simulation code and derive useful performance measures.

1 Introduction

Due to the complexity of today's networks, simulation is a widely used mechanism to evaluate the performance of new protocols or network configurations. However, the resulting simulation models are also very complex, often unclear or lacking documentation, and in most cases too complicated to be understandable at a glance. This leads not only to difficulties while debugging and maintaining the developed simulation models. The exchange of simulation models between research groups, and possibly between different simulators, is also hampered.

Fortunately, improved modeling techniques are available since several years. These techniques often include graphical mechanisms that enable humans to quickly understand the structure and behavior of the modeled system.

The Unified Modeling Language (UML) is such a graphical modeling language and is standardized by the Object Management Group (OMG). The current version of the standard is 2.1.1 [20]. A variety of academic and commercial tools exist that support model development with UML diagrams. To support the exchange and automated processing of UML diagrams, an alternate textual format called XML Metadata Interchange (XMI) has also been defined by the OMG.

Using the UML as a modeling language in the context of network simulation has several advantages compared to other available languages. The graphical representation of the main model elements and the enforcement of a modern, object-oriented approach to the modeling of systems make complex models far easier to comprehend. This also leads to the earlier discovery of conceptual errors in the model, accelerates model debugging and helps to locate errors sooner. In addition, the UML is very popular, easy to learn, and knowledge about it is already widely spread. Therefore, there is no need for another new programming language – and, in our case, there is no need to learn the internals of another new network simulator. And, of course, graphical programming is "en vogue" right now (and has been for the last years), which might increase the acceptance level among model developers.

For these reasons, we believe that the automated simulation of UML models is a very promising approach especially in the field of network simulation. This also takes UML one step further from a purely graphical formalism. The fact that the models remain machine-readable by means of the exchange format XMI leads to the applicability of UML for all kinds of model transformations. We expect that the development of new network protocols and the evaluation of their performance can be accelerated significantly with the adoption of the UML as the basis for network simulations. Therefore, we are developing an Eclipse-based tool that we named *Syntony*, which is capable of transforming UML models into executable simulation code.

Syntony is currently able to process standard-compliant UML models consisting of composite structure, state machine and activity diagrams. Performance annotations using the UML profile for QoS and Fault Tolerance Characteristics and Mechanisms, as well as some custom stereotypes are also possible.

To show the applicability and usage of *Syntony*, we developed a UML model of communications within the AKTIV project *CoCar* [2] and used *Syntony* to automatically generate an executable simulation model from it.

The remainder of this paper is organized as follows. After a brief overview of related work in Section 2, we show how computer networks can be modeled with the UML in Section 3. Section 4 discusses the inner workings of *Syntony* and details the transformation process from UML to simulation code. In Section 5, we present a case study using the CoCar scenario. Finally, Section 6 concludes the paper and gives some directions for future research.

2 Related work

Recently, a number of approaches have been published that use UML models to analyze the performance of software architectures. In most cases, UML models are not transformed into simulation, but into other mechanisms suitable for performance analysis, for example Petri nets, queuing networks, process algebras, or stochastic processes. Balsamo et al. [4] give a broad overview of some of the existing approaches. In their work, they mention two simulation-based approaches, namely work by Arief and Speirs [3] and by De Miguel et al. [9]. Both approaches are included in the discussion below.

Of course, UML models have also been used as the basis for simulation studies. One of the earliest works in this area was done by Pooley et al. [14,21]. They use early versions of the UML 1 to generate discrete-event simulations from sequence diagrams. However, they maintain that state and activity diagrams are more suitable for system specification because sequence diagrams only capture specific message flows, but not all possible and legal message exchanges. They conclude that sequence diagrams are better suited for the animation of a simulation behavior than its specification. Arief and Speirs [3] transform systems specified with class and sequence diagrams into C++ or Java simulation code using a framework called *SimML*. Borshchev et al. [7] describe an approach that uses UML 1 state machines and composite structure diagrams, and annotations taken from the unofficial profile *UML for Real-Time*. From these models, they generate simulation programs in Java. They implemented their approach in the commercial tool AnyLogic. De Miguel et al. [9] define a set of custom stereotypes and tagged values to support the modeling of performance parameters. They specify systems using class, deployment and activity diagrams and annotate them with their custom profile. Their Simulation Model Generator (SMG) is able to transform these UML models into simulation models for the commercial simulator OPNET. Marzolla and Balsamo [5, 15, 16] use activity, use case, and deployment diagrams to specify systems and annotate performance aspects with the UML profile for schedulability, performance, and time (SPT). These models are transformed into code for a custom, process-oriented C++ simulator. Barth [6] uses activity and class diagrams to describe a system. Performance aspects are included from an external library and thus not modeled in UML. The system can be analyzed with a Java simulator. Michael et al. [17] developed a mapping of UML-RT models to the simulator OMNeT++. They specify the system using composite structure and state machine diagrams. Application requirements are specified with sequence diagrams which are generated from use cases. De Wet and Kritzinger [10] transform UML 2.0 models to Specification and Description Language (SDL) using the ITU Z.109 [13] profile. The ITU Z.109 profile defines a subset of the UML 2.0 and how the elements are mapped to SDL specifications. Existing methods for the incorporation of temporal aspects into SDL specifications are then used to analyze the model with process-oriented simulation. Choi et al. [8] use class and sequence diagrams to model systems. The sequence diagrams are then transformed into state machines. A discrete-event simulation model is generated from the classes and state machines. However, it is unclear how performance elements are integrated into this approach.

The main differences between *Syntony* and the related work discussed above are that our approach is fully compliant to the UML 2 standard, uses a set of diagrams that is particularly suitable for the modeling of network protocols (but also for most other applications), is built on widely known simulation tools and uses standardized UML profiles to annotate performance parameters.

3 Modeling systems and networks with UML 2

The UML offers a multitude of modeling elements and diagram types which can be used to model the structure and the behavior of systems. As the diagram types are partly redundant, a subset of the diagram types should be sufficient to model all relevant aspects of a system. The description of the system structure basically comprises the problem of which system elements there are, and how these elements are connected with each other. The possibilities to model system structure include a combination of component and deployment diagrams as in [11], or composite structure diagrams as in [7].

We decided to describe the system structure with *composite structure diagrams*. These diagrams can be used to display the internal structure of classes. This includes how other classes are nested inside a class, and how the nested classes can communicate via connectors attached to their ports.

The behavior of the entire system is determined by the functional operation of each system element, and the communication between the elements. There are three main options to model system behavior with UML 2 which have already been used in the literature. Activity diagrams are employed for example in [15] and [9], sequence diagrams in [3] and [8], and state machine diagrams in [7].

We chose to model the system behavior with *state machine diagrams*. UML state machines are very rich in features, which enables the modeler to produce very clearly structured, uncluttered models. At this point, there are two levels of detail to choose from. The less detailed variant is to annotate all transitions with transition probabilities. These probabilities can either come from measurements of an existing system, or from estimations. The detailed variant requires a complete specification of all transition effects and state actions (*entry*, *do*, *exit*). We use *activity diagrams* for their description. In this paper, we will only describe the second approach in detail.

The UML standard does not describe methods to model non-functional properties such as performance aspects of systems. However, UML profiles are defined as a flexible extension mechanism that can be used for this purpose. The OMG is currently in the process of standardizing a profile called Modeling and Analysis of Real-Time and Embedded Systems (MARTE) that will be suitable for the modeling of performance aspects. As the standardization is not yet finished, we had to rely on a combination of preliminary versions and own profile elements in this paper.

We defined a custom profile with three stereotypes. `<<simulationModule>>` can be used to reference existing simulation models in the UML model. Elements that are stereotyped as `<<simulationParameter>>` can be varied without recompiling the simulation, and can also be subject to systematical variation within given bounds. The stereotype `<<incrementStatistic>>` realizes a simple statistical counter.

The elements described above are sufficient to model arbitrary systems and networks. In particular with the multitude of UML 2 actions available for use in activity diagrams, it is assured that any behavior can be modeled using UML alone. However, this would become quite cumbersome as soon as the modeled algorithms reach a certain complexity. It is therefore desirable to allow the usage of code in a textual programming language at least at certain places in a model. Appropriate UML elements for this are easily identified: OpaqueActions and OpaqueBehaviors allow the specification of a textual body and a corresponding language. We decided to support two different languages: the native language of the underlying simulation core (we are currently using C++), and the Object Action Language (OAL) [1] as a convenience language building on the UML action semantics standard [19]. OAL facilitates for example the specification of message sending and timer generation.

4 Syntony

We are developing the tool *Syntony* to enable simplified, flexible, and statistically sound simulation of models specified in the UML modeling language. As its input *Syntony* uses UML models as described in the previous section. The tool then analyzes the model, transforms it, and outputs a simulation model specified in C++ as is required by the used simulation core OMNeT++ [22]. The details of this process will be described in this section.

4.1 Basic concepts

Syntony is a software tool written in Java as a plug-in for the popular open source development environment Eclipse[1]. As such, its graphical user interface is realized as a set of Eclipse views. The input models have to be available in the XMI format as supported by the Eclipse UML 2 plug-in. Computer-Aided Software Engineering (CASE) tools able to export UML models into this format include for example the IBM Rational Software Modeler[2], Sparx Systems Enterprise Architect[3], and Omondo EclipseUML[4].

After the import of a particular UML model, *Syntony* transforms the model into C++ code. This transformation is described in detail below. The code is then compiled into a simulation program and executed. Apart from the initial import, these steps may be executed automatically. The execution of these steps is controlled from the *Tool Control* element of the graphical user interface (see Figure 1).

Another element of the user interface is the *Translation View* as depicted in Figure 2, which illustrates the structure of the input model and annotates error and warning messages generated during the transformation process.

4.2 OMNeT++

We currently rely on OMNeT++ [22] as the underlying simulation core. OMNeT++ is based on C++ and was designed to support efficient network simula-

[1] `http://www.eclipse.org/`
[2] `http://www.ibm.com/software/awdtools/modeler/swmodeler`
[3] `http://www.sparxsystems.com.au/`
[4] `http://www.omondo.com/`

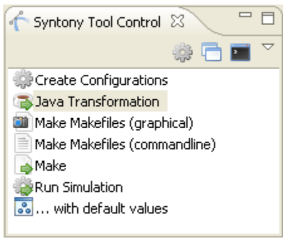

Fig. 1. Tool control view

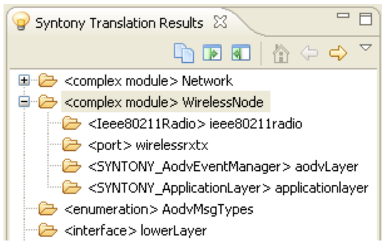
Syntony Translation Results
<complex module> Network
<complex module> WirelessNode
<Ieee80211Radio> ieee80211radio
<port> wirelessrxtx
<SYNTONY_AodvEventManager> aodvLayer
<SYNTONY_ApplicationLayer> applicationlayer
<enumeration> AodvMsgTypes
<interface> lowerLayer

Fig. 2. Translation results view

tion. OMNeT++ distinguishes two kinds of classes, complex and simple modules. Classes that are composed of other classes, plus connections between them, are called complex modules. They are specified in the network description language (NED). Atomic classes with an associated behavior are called simple modules. They are written in C++, and are accompanied by a short NED description of their configuration parameters and interfaces, called *gates*, available for this class.

4.3 Transformation of UML model elements

Based on the above description of the OMNeT++ way of building simulations, it is quite clear how the main UML model elements correspond to OMNeT++ concepts. Classes with state machine diagrams become simple modules. Classes with composite structure diagrams are transformed into complex modules. UML ports are represented by a very similar construct in OMNeT++ called gates.

State machine diagrams are embedded in OMNeT++ simple modules. We do not use the Finite State Machine (FSM) mechanism built into OMNeT++ because UML features such as history states or orthogonal states would have been difficult to integrate with that mechanism. Instead, we base our translation of UML state machines on the *state* design pattern. A translation of some features of UML state machines into Java code using the *state* pattern has already been described in [18]. We loosely lean on that approach, with a few differences.

The main difference is that in the state pattern, the firing of transitions is delegated to the state objects. That is not the case in our implementation. Instead, we define a method for each state class that returns the currently enabled transitions depending on the current state, trigger and guards. The collection of enabled transitions is then evaluated at a central place, and the chosen transition is fired from there. The main advantage of this method is that the different firing priorities as defined in the UML standard, as well as extra tie-breaking rules, can be included a lot easier.

Figure 3 illustrates some of the features of UML state machines. Besides regular, initial, and final states, our implementation includes shallow and deep history states, nested states, orthogonal regions, joins and forks, junctions, and choice vertices. The figure also helps to demonstrate some pitfalls that can be met when modeling state machines. Most of these pitfalls are mentioned as semantic variation points in the UML standard. The first issue is the question of what happens if an enclosing state does not have an initial state. This is the case with both regions contained in *State1* in the figure. Another question is what happens if several transitions are enabled at the same time. The standard defines that those transitions that leave the state with the deepest nesting level have the highest firing priority. In the figure, the transitions leaving *State11* and *State13* all have the same priority. If their triggers match and all guards evaluate to *true*, the standard does not define which one will be taken. *Syntony* currently chooses a random transition in this case, while a more sophisticated tie-breaking algorithm (or even a choice from several algorithms) would be desirable. Warnings or errors are created and shown in the *Translation View* if such a pitfall is recognized during the transformation.

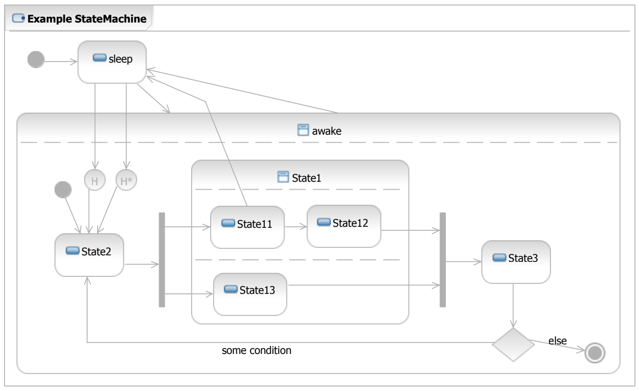

Fig. 3. Example of a state machine diagram

The effects of transitions and the entry, do, and exit actions of states may be specified in detail with activity diagrams.

All classes containing a composite structure diagram are translated to OMNeT++ complex modules. The parts contained in such a diagram are listed in the *submodules* section of the resulting NED file. The ports of the class are listed in the *gates* section, and the connections between class ports and ports on contained parts are detailed in the *connections* section.

Unfortunately, a few semantic issues concerning composite structure diagrams are left open in the UML standard. One such issue is the question of the exact connection topology for many-to-many connections. Possibilities to resolve this issue include the *star* pattern (connecting each element with all elements on the other side) and the *array* pattern (connecting the *i*-th element only with the *i*-th element on the other side). *Syntony* currently uses the star pattern, but outputs a warning message if this situation is encountered. Another problem refers to the semantics of signal forwarding. Imagine a signal arrives via some connection at one side of a port, and should be forwarded to the other side of the port. What should happen if there are multiple connectors attached to that side of the port? The signal could be forwarded on all connections, just one connection, or a subset of the available connections. *Syntony* currently forwards the signal on all connections, and also outputs a warning message.

Syntony includes a compiler for OAL statements. This compiler has been written based on the EBNF production rules for OAL contained in [1] and the SableCC parser generator [12]. The compiler translates the statements from OAL to C++ and inserts them at the appropriate places in the C++ code generated by *Syntony*. Error messages are attached to the *Translation View* if a statement could not be translated.

4.4 Integrating existing simulation modules

The integration into an existing simulation framework inevitably raises the question if models that are already existing in the simulator can somehow be re-used in the context of our UML-driven simulations. The benefits of such a reuse are basically the same as for the reuse of other software components. Models that are already developed and tested need not be developed again. Models created by other people can be integrated. Performance comparisons are facilitated.

For the integration of existing OMNeT++ modules into UML models, we chose an approach that combines custom stereotypes with a model library representing the existing modules. In the model library, all reusable modules are represented by classes. The module parameters are attributes of these classes. All attributes are given appropriate default values. Elements from the model library can then be used in the context of a UML model by using them as parts in composite structure diagrams.

For the transformation into simulation code, two stereotypes have to be applied to the classes and their attributes. The `<<simulationModule>>` stereotype used for classed has one tag value which contains the OMNeT++ name of the module. During the transformation, parts that are stereotyped in this way are replaced by the corresponding OMNeT++ modules. Additionally, the module parameters have to be stereotyped as `<<simulationParameter>>`. This stereotype has several effects. For one, the corresponding attributes are placed in the OMNeT++ initialization file and can then be varied without recompiling the simulation. Second, the stereotype has tags that indicate how the stereotyped parameter should be varied systematically during the simulation runs.

5 Case study: Cooperative Cars

In order to further outline the capabilities of *Syntony* and to demonstrate the general feasibility of our approach, we created a model of the communications within the the AKTIV project *CoCar* in UML. Based on this model, selected details of the UML modeling process are outlined.

Project Cooperative Cars describes itself as follows [2]: The CoCar project is aiming at basic research for Car-to-Car (C2C) and Car-to-Infrastructure (C2I) communication for future cooperative vehicle applications using cellular mobile communication technologies. Five partners out of the telecommunications- and automotive industry develop platform independent communication protocols and innovative system components. They will be prototyped, implemented and validated in selected applications. Innovation perspectives and potential future network enhancements of cellular systems for supporting cooperative, intelligent vehicles will be identified and demonstrated. In a first step, potential application scenarios will be specified, data flows and information content analyzed, communication requirements of cellular C2C and C2I applications identified. Traffic and communication models will be worked out. A network load and latency simulator will be developed. The simulator behavior will be verified against a broad spectrum of telematics applications and related communication models. The simulation results will constitute the foundations for consecutive technical feasibility studies. The results shall also identify upcoming demands for cellular network evolutions. Extensible multi-party vehicular application protocols for global deployments will be worked out and evaluated in scope of the project.

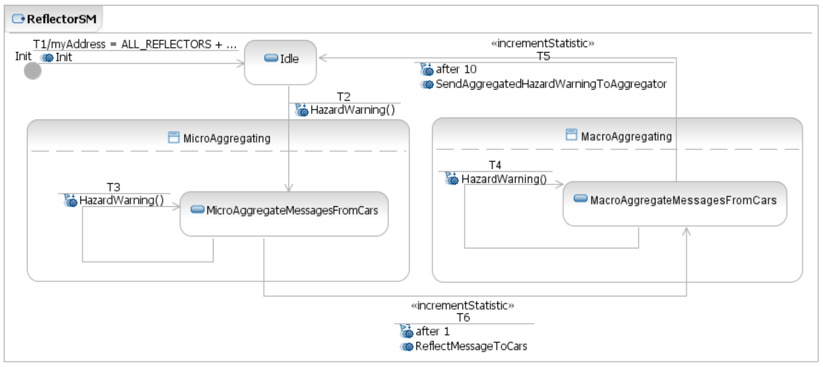

Fig. 4. State machine diagram for the Reflector component

The UML model we created represents a very early stage of the operations envisioned in the CoCar project. The behavior of all components is modeled with state machine diagrams. The state machine of one component, the *Reflector*, is shown in Figure 4. Also modeled in the simulated system are other network infrastructure components, as well as cars that communicate wirelessly via an air channel. The structure of the entire system is shown in Figure 5.

From this UML model, *Syntony* generates about 3000 lines of C++ code. While there are some redundancies in the code generated by *Syntony* and although this number also includes many debug statements (automatically generated from the names of states and actions), as well as statistical counters, it is probably safe to say that a UML model (even a complex one) is easier to maintain and understand than several thousand lines of arbitrary code.

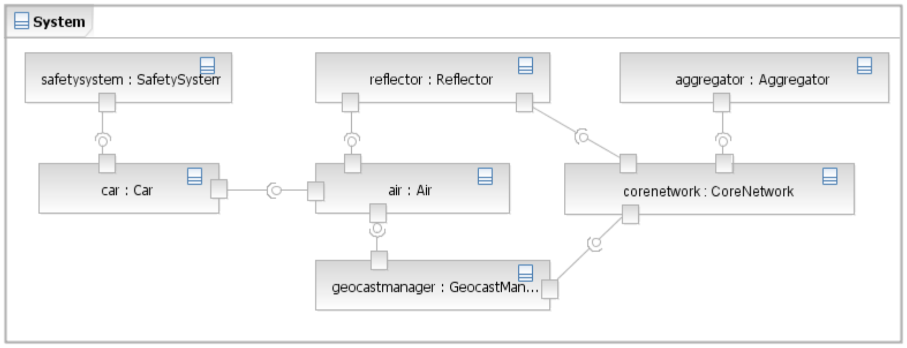
System
safetysystem : SafetySystem
reflector : Reflector
aggregator : Aggregator
car : Car
air : Air
corenetwork : CoreNetwork
geocastmanager : GeocastMan...

Fig. 5. The CoCar system

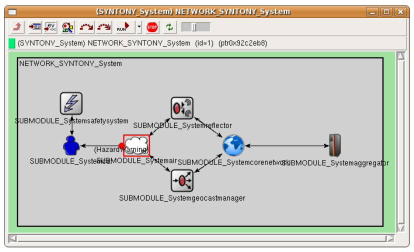
(SYNTONY_System) NETWORK_SYNTONY_System
RUN
STOP
(SYNTONY_System) NETWORK_SYNTONY_System (id=1) (ptr0x92c2eb8)
NETWORK_SYNTONY_System
SUBMODULE_Systemsafetysystem
SUBMODULE_Systemreflector
(HazardWarning)
SUBMODULE_Systemcar
SUBMODULE_Systemair
SUBMODULE_Systemcorenetwork
SUBMODULE_Systemaggregator
SUBMODULE_Systemgeocastmanager

Fig. 6. Simulation model running in OMNeT++

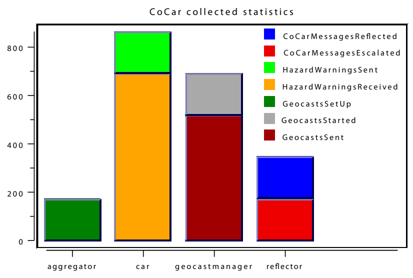

Fig. 7. Performance measures collected during the CoCar simulation

The translated code can then be directly compiled and executed as a model of the OMNeT++ simulation environment. A screenshot of the simulation execution is depicted in Figure 6. Using *Syntony* and OMNeT++, the UML model can thus be automatically simulated and typical measures such as the number of transmitted messages and other performance measures, depicted in Figure 7, can be collected.

6 Conclusion and future work

We demonstrated *Syntony*, an Eclipse-based tool for the automated translation of UML 2 models into executable simulation code. We also motivated the need for standardized modeling languages for doing so and the choice of UML 2 diagrams that fulfill our requirements to a certain extent. Additional features can be modeled (and executed by *Syntony*) using profiles that allow to incorporate non-functional properties of the system, and by annotating modeled actions with OAL or C++ code. The usability of the tool chain has been verified by the development of a UML 2 model of the CoCar project's communication architecture.

This demonstrates that *Syntony* is well suited for large applications: it is flexible, easy to use, and can handle complex modeling tasks. As a consequence, existing UML tools can be used for the description of a complex simulation model. It is therefore possible to integrate simulation seamlessly in the system design process.

In conclusion, it can be said that *Syntony* supports very convenient graphical modeling and programming paradigms while achieving both similar accuracy and simulation performance compared to native models.

In future work, we plan to integrate the modeling facilities provided by the MARTE profile as soon as its standardization is finished. In particular, this will include the modeling of stochastic timing and probabilistic choices. Resolving the open semantic issues in this context will also be a part of our work.

We are also working on a standard-compliant action language as a replacement for the combination of OAL and C++ to increase the flexibility of the developed models. The integration of other simulation cores is planned as well.

In addition, we plan to extend the functional range of *Syntony* by adding facilities for the animation of the simulation execution, as well as a component for the integrated evaluation and presentation of simulation results. The inclusion of a component for statistical testing is also intended.

Acknowledgments. This work was supported by the Fraunhofer Institute for Integrated Circuits IIS, department for Communication Networks and by Vodafone Research & Development in the context of the CoCar project funded by the BMBF under grant number 01BU0692.

References

1. Accelerated Technology. Object Action Language Manual. Technical report, Embedded Systems Division of Mentor Graphics Corporation, 2004.
2. AKTIV Project. Fact Sheet: Adaptive and Cooperative Technologies for the Intelligent Traffic, November 2006.
3. L. B. Arief and N. A. Speirs. A UML Tool for an Automatic Generation of Simulation Programs. In *Workshop on Software and Performance (WOSP)*, pages 71–76, Ottawa, Ontario, Canada, 2000.
4. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
5. S. Balsamo and M. Marzolla. Simulation Modeling of UML Software Architectures. In *European Simulation Multiconference*, Nottingham, UK, 2003.

6. M. Barth. Performance Assessment of Software Models In a Configurable Environment Simulator. In *International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, USA, 2003.
7. A. V. Borshchev, Y. B. Kolesov, and Y. B. Senichenkov. Java engine for UML based hybrid state machines. In *Winter Simulation Conference*, pages 1888–1894, Orlando, Florida, USA, 2000. Society for Computer Simulation International.
8. K. Choi, S. Jung, H. Kim, D.-H. Bae, and D. Lee. UML-based Modeling and Simulation Method for Mission-Critical Real-Time Embedded System Development. In *The IASTED Conference on Software Engineering*, Innsbruck, Austria, February 2006.
9. M. de Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec. UML extensions for the specification and evaluation of latency constraints in architectural models. In *Workshop on Software and Performance (WOSP)*, pages 83–88, Ottawa, Ontario, Canada, 2000. ACM Press.
10. N. de Wet and P. Kritzinger. Using UML Models for the Performance Analysis of Network Systems. *Elsevier Computer Networks*, 49(5):627–642, 2005.
11. A. Di Marco and C. Mascolo. Performance analysis and prediction of physically mobile systems. In *6th international workshop on Software and performance*, pages 129–132, Buenes Aires, Argentina, 2007.
12. E. M. Gagnon and L. J. Hendren. SableCC, an Object-Oriented Compiler Framework. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, page 140, 1998.
13. ITU-T. ITU Recommendation Z.109: SDL Combined with UML. Technical report, International Telecommunication Union, 2000.
14. C. Kabajunga and R. Pooley. Simulating UML Sequence Diagrams. In R. Pooley and N. Thomas, editors, *UK Performance Engineering Workshop*, pages 198–207, July 1998.
15. M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universit'a Ca Foscari di Venezia, 2004 2004.
16. M. Marzolla and S. Balsamo. UML-PSI: the UML Performance SImulator. In *1st International Conference on the Quantitative Evaluation of Systems (QEST)*, 2004.
17. J. B. Michael, M.-T. Shing, M. H. Miklaski, and J. D. Babbitt. Modeling and Simulation of System-of-Systems Timing Constraints with UML-RT and OMNeT++. In *IEEE International Workshop on Rapid System Prototyping (RSP'04)*, pages 202–209, Geneva, Switzerland, 2004. IEEE Computer Society.
18. I. A. Niaz and J. Tanaka. An Object-Oriented Approach To Generate Java Code From UML Statecharts. *International Journal of Computer and Information Science (IJCIS)*, 6(2):83–98, June 2005.
19. O. M. G. (OMG). Unified Modeling Language Specification (Action Semantics). Technical report, OMG, 2001.
20. O. M. G. (OMG). UML 2.1.1 Superstructure Specification. Technical report, OMG, 2007.
21. R. Pooley and P. King. The Unified Modelling Language and performance engineering. *IEE Proceedings Software*, 146(1):2–10, February 1999.
22. A. Varga. The OMNeT++ Discrete Event Simulation System. In *European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, 2001.