SYNTONY: Network Protocol Simulation based on Standard-conform UML 2 Models

Isabel Dietrich
Falko Dressler
Volker Schmitt
Reinhard German

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

ABSTRACT

In this paper, we discuss the need for using standardized graphical modeling languages for developing and evaluating simulation models. 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 provided by *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. Additionally, the complete process of debugging and analyzing the protocol is tool-assisted. For verification purposes, we developed an UML 2 model of the Ad hoc On-Demand Distance Vector (AODV) protocol. In comparison with a native OMNeT++ model, we were able to show, first, that the developed model works suitably and the achieved performance measures of the routing protocol are comparable, and second, that the overhead of the translation process does not lead to an essential performance degradation of the simulation process.

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, but it also hinders the exchange of simulation models between research groups, and possibly between different simulators.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NSTools '07, October 22, 2007, Nantes, France
Copyright 2007 ICST 978-963-9799-00-4.

The Unified Modeling Language (UML) is such a graphical modeling language standardized by the Object Management Group (OMG). The current version of the standard is 2.1.1 [21]. 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 model systems make complex models far easier to comprehend. This leads to the earlier discovery of conceptual errors in the model, accelerates model debugging, and helps to find error locations sooner.

In general, using the UML allows to model network protocols very similarly as with the Specification and Description Language (SDL), which is based on communicating automatons. However, in contrast to SDL, UML is widely accepted within the industry as well as in the academic society. Due to the available variety of diagram types and the profile mechanism, UML is also a lot more flexible and easier to extend than SDL. In addition, the UML is very easy to learn, and knowledge about it is already widely spread.

The main advantage of using a modeling language such as UML for developing simulation models is that 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, to run the simulation, and to analyze the simulation results.

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 and some custom stereotypes are also possible.

To show the applicability and usage of *Syntony*, we developed a UML model of the routing protocol AODV and used *Syntony* to automatically generate an executable simulation model from it.

We then evaluated *Syntony* based on a comparison between the generated model of AODV and a native OMNeT++ implementation of it. The results show that both the protocol performance and the run-time performance of our model are comparable to the native implementation.

In short, the contributions we present in this paper are:

- a method for the modeling of network protocols using UML 2, with the following properties:
  - fully conform to existing standards and common formats
  - based on composite structure, state machine, and activity diagrams
  - allows to integrate code annotations based on OAL and C++
  - allows to integrate existing native simulation modules
- the automated translation of UML 2 models into simulation code
- the evaluation of our approach based on the analysis of a typical network protocol, AODV, using a comparison of a developed AODV UML model and a native implementation for OMNeT++

The remainder of this paper is organized as follows. After a brief overview over related work in section 2, we show how computer networks can be modeled with 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 routing protocol AODV including a detailed performance evaluation of *Syntony*. 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. [3] give a broad overview of some of the existing approaches. In their work, they mention two simulation-based approaches, namely the work by Arief and Speirs [2] and the one by De Miguel et al. [10]. Both of them are discussed 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. [15, 24]. They use early versions of the UML 1 to generate discrete-event simulations from sequence diagrams. However, they state 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 [2] transform systems specified with class and sequence diagrams into C++ or Java simulation code using a framework called *SimML*.

Borshchev et al. [6] 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. [10] define a set of custom stereotypes and tagged values to support 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 [4, 16, 17] 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 [5] 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. [18] 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 [11] transform UML 2.0 models to SDL using the ITU Z.109 [14] 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 generated state machines. However, it is unclear how performance elements are integrated into this approach.

In table 1, we give a brief summary of the approaches described so far.

From this overview, it seems like there exists a lot of related work which we could just have adapted instead of developing yet another tool. However, we found that all of the approaches have more or less serious drawbacks so that we decided to learn from their work instead of adapting it.

One big disadvantage of the discussed approaches is that most of them use custom simulation cores. We aim to support widespread standard simulation tools. This is beneficial for two main reasons. First, the newly developed UML-based models can be easily compared with already existing models, and second, existing models can be integrated into UML-based models, thus reducing the development effort.

Reference	UML version	Diagram types used	Profiles used
[2]	1	class, sequence	-
[10]	1	class, deployment, activity	custom
[6]	1	state, composite structure	UML-RT
[15, 24]	1	sequence, state, architecture	-
[4, 16, 17]	1.4	activity, use case, deployment	SPT
[5]	1.4	activity, class	-
[18]	1.5	composite structure, state, sequence	UML-RT
[8]	2.0	use case, class, sequence	
[11]	2.0	class, architecture, state, collaboration	ITU Z.109
Syntony	2.0	state, composite structure, activity	MARTE, QoS, custom

Table 1: Approaches for UML-based simulation

Another important issue is the annotation of performance aspects. About half of the tools we evaluated do not support clean and standard compliant annotations using UML profiles. However, we believe that a tool looses much of its utility by only partially complying to the standard.

We also consider some of the diagram types used in the related work unsuitable for system specification. These are especially sequence diagrams, following Pooley's argument already cited above. Also, most of the tools are based on UML 1 which lacks a lot of features concerning the modeling of actions and internal structures of classes.

Finally, the input format used by the discussed tools is often proprietary and can not be easily exchanged with other UML modeling tools. This is a serious disadvantage because the model designer is forced to use one specific tool for modeling. In contrast, we chose to build upon the popular Eclipse UML 2 plug-in. The exchange format defined by this plug-in is natively supported by a few modeling tools, and several more provide importers and exporters for it.

In summary, the main difference between *Syntony* and the related work discussed above is that our approach provides an integrated process for the modeling and simulation of systems that is fully compliant to the UML 2 standard. Our approach also builds upon widely known simulation tools, allows to embed functionality described with symbolic languages and enforces the use of UML profiles to annotate performance parameters.

3. MODELING SYSTEMS AND NETWORKS WITH UML 2

3.1 System structure and behavior

The UML offers a multitude of modeling elements and diagram types which can be used to model the structure and 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 [12], or composite structure diagrams as in [6].

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 composed of 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 [16] and [10], sequence diagrams in [2] and [8], and state machine diagrams in [6].

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* to describe these details. In this paper, we will only describe the second approach in detail.

3.2 Non-functional properties

So far, only the functional properties of a system have been modeled. However, the non-functional properties of a system are equally important for the analysis of its performance. The non-functional properties include for example the consumption of various resources such as time, CPU, or energy at various points in the system.

The UML standard does not describe methods to model 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. The stereotype `<<simulationModule>>` can be used to embed 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.

3.3 Integration of code

The elements described above are sufficiently suitable to model arbitrary systems and networks. 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 [20]. 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. *Syntony* uses UML models as described in the previous section as input. The tool then analyzes the model, does some transformations, and outputs a simulation model specified in C++ as required by the used simulation core OMNeT++ [25]. The details of this process will be described in this section. Further information and developments regarding *Syntony* will be published on the project website[1].

4.1 Basic Concepts

Syntony is a software tool written in Java as a plug-in for the popular open source development environment Eclipse[2]. 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. CASE tools able to export UML models into this format include for example the IBM Rational Software Modeler[3], Papyrus[4], Sparx Systems Enterprise Architect[5], and Omondo[6].

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 an element of the graphical user interface (see figure 1).

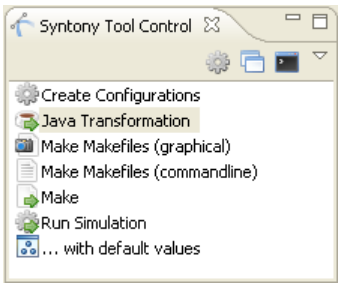

Figure 1: Tool control view

[1] http://www7.informatik.uni-erlangen.de/syntony/
[2] http://www.eclipse.org/
[3] http://www.ibm.com/software/awdtools/modeler/swmodeler
[4] http://www.papyrusuml.org
[5] http://www.sparxsystems.com.au/
[6] http://www.omondo.com/

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.

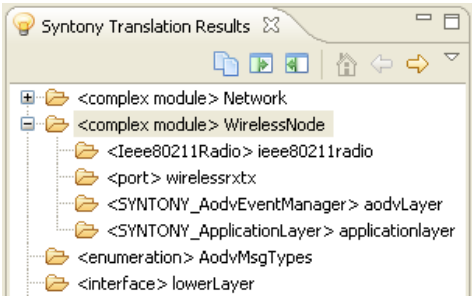

Figure 2: Translation results view

4.2 OMNeT++

We currently rely on OMNeT++ [25] as the underlying simulation core. OMNeT++ is based on C++ and was designed to support efficient network simulation. OMNeT++ distinguishes two kinds of classes, complex and simple modules. Classes that are composed of other classes and 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 the parameters and 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 will become simple modules. Classes with composite structure diagrams are transformed into complex modules. UML ports are represented as OMNeT++ gates.

State machine diagrams are embedded in OMNeT++ simple modules. We do not use the 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 [19]. 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. Next to 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 shows 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.

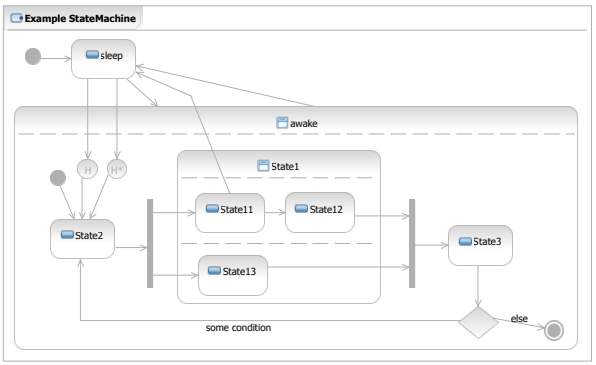

Figure 3: Example of a state machine diagram

The effects of transitions and the entry, do, and exit actions of states may be specified in detail with activity diagrams. Figure 4 shows an example how this might look like. To illustrate the usage of OAL statements, the note in the lower right corner of the diagram displays the OAL code contained in the corresponding action.

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. Figure 5 shows a composite structure diagrams which represents the internal structure of a wireless node containing various communication layers. Finally, figure 6 shows how this diagram is transformed into a NED description of the node (for the sake of readability, the description has been shortened).

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 at 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.

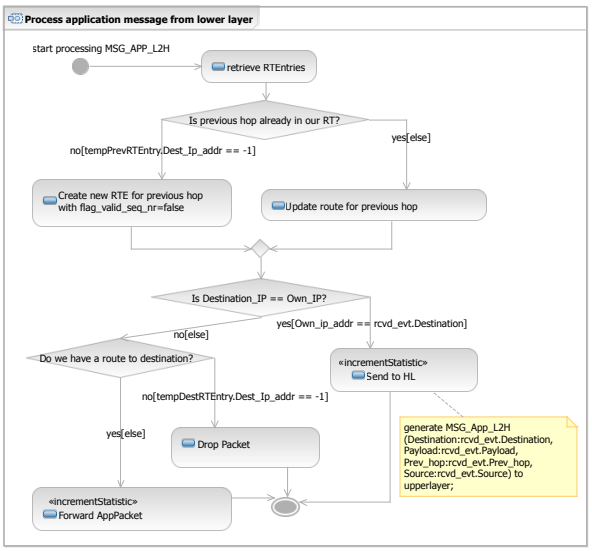

Figure 4: Example of an activity diagram

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 [13]. 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 can 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 do not need to 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 classes are stereotyped as `<<simulationModule>>`. The

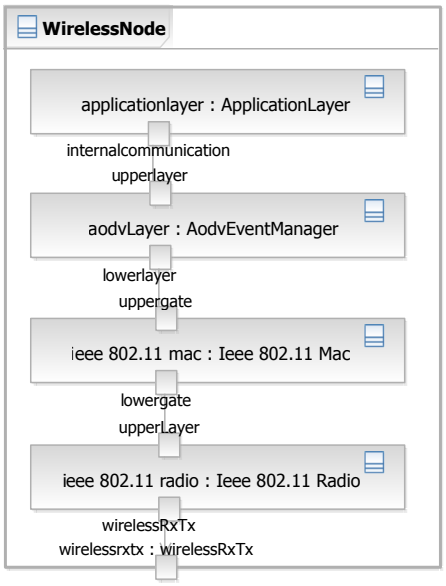

Figure 5: Internal structure of a wireless node

```
module SYNTONY_WirelessNode
parameters:
gates:
out: Fromwirelessrxtx;
in: radioIn;
submodules:
aodvLayer: SYNTONY_AodvEventManager;
applicationlayer: SYNTONY_ApplicationLayer;
mobility: NullMobility;
display: "p=20,160;i=block/cogwheel";
ieee80211radio: Ieee80211Radio;
display: "i=block/wrxtx;";
mgmt: Ieee80211MgmtAdhoc;
display: "i=block/cogwheel;";
mac: Ieee80211Mac;
display: "i=block/layer;";
[...]
connections nocheck:
aodvLayer.FromUpper --> applicationlayer.ToLower;
aodvLayer.ToUpper <-- applicationlayer.FromLower;
ieee80211radio.radioIn <-- radioIn;
ieee80211radio.uppergateOut --> mac.lowergateIn;
ieee80211radio.uppergateIn <-- mac.lowergateOut;
[...]
endmodule
```

Figure 6: NED description of a wireless node

stereotype 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: AD HOC ROUTING WITH AODV

In order to further outline the capabilities of *Syntony* and to demonstrate the general feasibility of our approach, we created a detailed model of the AODV routing protocol in UML. Based on this model, selected details of the UML modeling process are outlined. Finally, we compare the *Syntony* simulation model to an existing implementation of AODV, i.e. AODV-UU[7], which is natively available for the INET framework extension of OMNeT++.

Next to the availability of a hand-coded model which can be used for comparative evaluations, the second reason why we chose AODV for the case study is the protocol's complexity. Although it is certainly not as complex as for instance TCP (about 60 printed pages for the original TCP RFC 793 versus about 25 printed pages for AODV RFC 3561), it is still complex enough to demonstrate that it can be effectively modeled using a graphical description.

5.1 Ad hoc On-Demand Distance Vector

The AODV routing protocol [22, 23] is tailored to fit the needs of mobile ad-hoc networks. Most importantly, this means that the protocol tries to minimize the number of control messages needed to achieve the successful transmission of data messages. For that reason AODV is designed as a reactive routing protocol, which means that route discoveries are done only on the initiative of a node which actually needs to send a data packet to a specific destination.

In that situation, the originating node broadcasts a route request message and waits for a corresponding route reply. Only then can it start to send data messages to the destination node. Several mechanisms have been built into AODV to make this flooding of route request messages as efficient as possible.

Destination sequence numbers are used to prevent multiple broadcasts of the same route request message received from several neighboring nodes. They are also used to ensure loop-free routes. The time-to-live header field is used to limit the route request flood to a certain number of hops. To reduce congestion in the network, a binary exponential backoff is used while reattempting to send route requests.

Once a route is established, no further control messages are needed. As long as a route is frequently used, every involved node knows how to forward packages. Stale routes will be removed from routing tables after a certain timeout in order to save system resources. In the case that a node notices a link break to one of its neighbors, it is possible to inform the potential interested neighbors by sending route error messages. For this purpose each nodes keeps a list of nodes which already have used this node to reach a destination, the so-called precursor-list, for each single destination.

The advantages of AODV are the low resource requirements in terms of memory usage and computational complexity, and the low communication overhead in most scenarios for ad hoc networks. Drawbacks are the quite high latency while establishing a new connection and the involved (partial) flooding of the network, which heavily decreases the throughput during the initialization. A number of papers have been published that analyze the performance of AODV [7, 9], which we used as drafts for our comparison.

[7] `http://core.it.uu.se/core/index.php/AODV-UU`

5.2 Our UML model of AODV

The main part of our AODV model is a class with one behavioral state machine diagram which handles all signals incoming from the upper and lower layers. The class has two ports which can be used to connect upper and lower layers. This is illustrated in the class *AodvEventManager* in figure 5. The state machine has only two states, *startup* and *normal operation* as can be seen in figure 7. During normal operation, there is one transition for each possible incoming signal. The actions taken in each transition are detailed with activity diagrams. Figure 4 shows the activity diagram that describes the necessary actions if an application message arrives from the lower layer.

This model represents one possible approach to model the AODV protocol. An alternative approach would have been to identify more states from the specification and use several orthogonal regions for the treatment of different message types.

Due to lack of time, we restricted our UML-based AODV implementation to include only required functionality as described in the IETF standard [22]. In general, this means that we left out almost all features that are not required by the RFC - in other words: we modeled all the *MUST* features, and skipped *SHOULD* and below. However, we believe that modeling the missing features would not be harder in terms of modeling techniques than modeling the features we have included.

One important feature we left out is the buffering of application messages in case a route to the destination is not known. This results in a high percentage of application message loss if application packets are sent at intervals that are larger than the active route timeout. Basically, this means a shift of responsibility toward the application layer which has to take care of re-sending its messages itself. It is important to note that this is allowed by the AODV specification.

Both *hello* messages and link layer notifications as a means to keep track of a node's local connectivity are not included in our UML model, as well as the possibility to send route error messages when the breakage of a link is detected.

Route reply acknowledgments are a mechanism that is mainly useful if there is a chance of unidirectional connections along the route. In that case, it is a *SHOULD* requirement of the standard and has therefore been left out.

The mechanisms described so far are all included in the AODV-UU implementation for OMNeT++, without an easy way to switch them off.

According to the standard, local repairs *MAY* be used. We did not model this feature, but also switched it off in the OMNeT++ implementation.

From this UML model, *Syntony* generates about 5000 lines of C++ code, including many debug statements (automatically generated from the names of states and actions) and statistical counters. The corresponding parts of the OMNeT++ implementation contain approx. 3000-4000 lines of code. While this shows that there are still some redundancies in the code generated by *Syntony*, 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.

Due to the fact that *Syntony* generates regular OMNeT++ modules, it is in principle possible to include UML-based models into OMNeT++, or even to replace existing modules with ones that were modeled graphically.

5.3 Performance evaluation setup

We combined the UML modeled AODV layer with a few existing modules from OMNeT++ (802.11 MAC and physical layers) in a composite structure diagram to form the wireless node shown in figure 5. Several of these nodes and the OMNeT++ wireless transmission were then put together to form a complete network. This is illustrated in figure 8. Network traffic is generated by a simple application layer modeled in UML that can send data messages at randomly distributed points in time.

The OMNeT++ model is built correspondingly, with the difference that it employed the UDP and IP layers available in the INET framework, and a UDP application as the traffic source and sink.

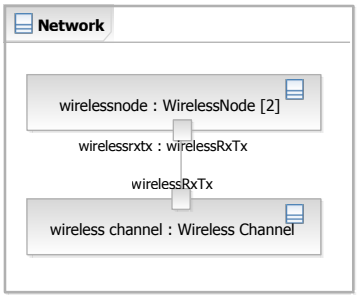

Figure 8: The sample network

Scenario	*# nodes*	*placement*	*sources*	*destination*
1	2	line	all	node 0
2	10	line	all	node 0
3	10	line	node 9	node 0
4	15	random	all	node 0

Traffic pattern	*interarrival time*	*simulation time*
1	uniform(1,2)	2 hours
2	uniform(2,4)	5 hours
3	uniform (8,10)	15 hours

Table 2: Simulation setup

In order to evaluate the performance of our model, we compared the protocol performance and the run-time performance of both models on the basis of a number of simulation scenarios. In all simulations, the nodes remained fixed on their positions.

The first scenario consisted of two wireless nodes placed within each other's reach. In the second and third scenarios, ten nodes were placed in a straight line, and each node was in the transmission range of only its immediate neighbors. In the second scenario, each node sends application messages, while in scenario 3 all nodes except the last node in the line are silent. The fourth scenario consists of fifteen nodes randomly placed on a simulation area of 200x200 meters. The nodes' communication radius is fixed at 62 meters.

The traffic generators were set to generate a new application message after a uniformly distributed period of time between 1 and 2 seconds, 2 and 4 seconds or 8 and 10 seconds. Correspondingly, the simulated time for the three traffic patterns was fixed at 2 hours, 5 hours and 15 hours, respectively. This is summarized in table 2. Each simulation setup was simulated in five independent replications.

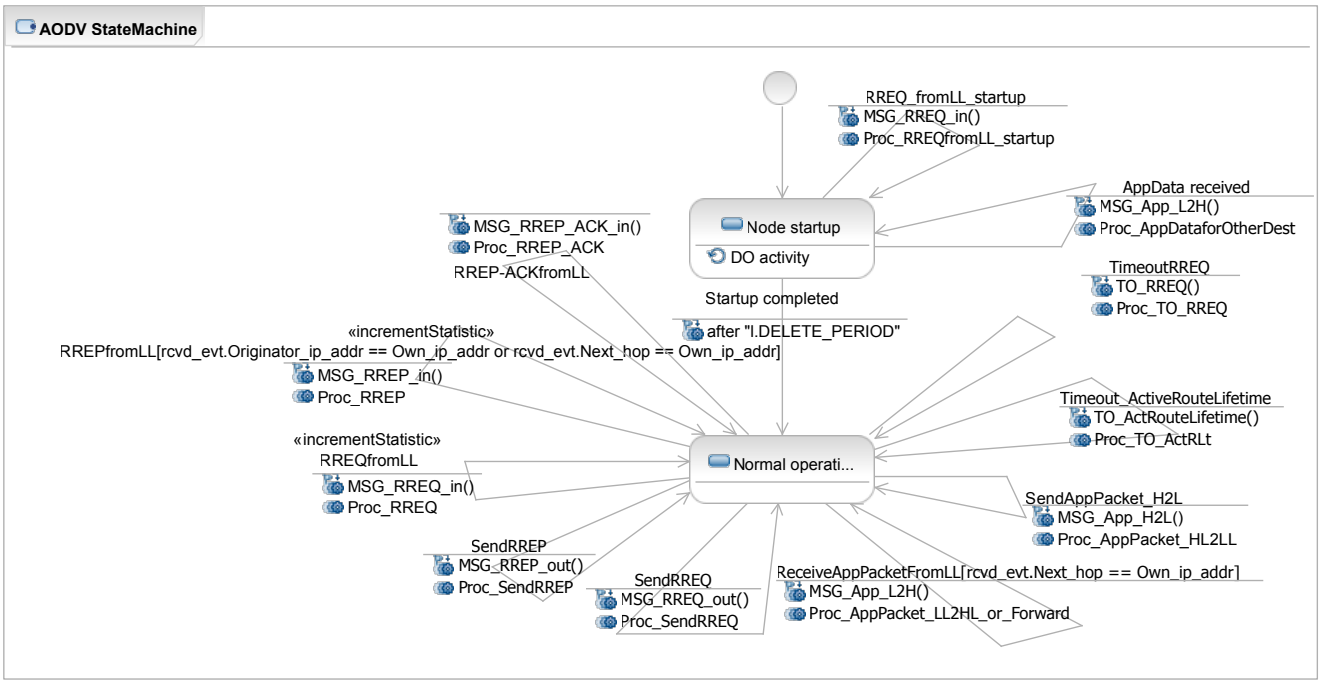

Figure 7: AODV state machine diagram

All other model parameters for AODV, and the mac and physical layers were chosen identical in all simulations. A summary of the most important values is given in table 3.

delete timeout	15 seconds
active route timeout	3 seconds
local repair	false
gratuitous rrep	false
expanding ring search	false
carrier frequency	2.4e+9
signal attenuation threshold	-85 db
alpha	2.5
frame capacity	5
mac.maxQueueSize	5
mac.rtsThresholdBytes	2346
bitrate	2e6
transmitter power	1.0
thermal noise	-100
snir threshold	4

Table 3: Simulation parameters

5.4 Comparison of protocol performance

Both models record a number of statistical counters, including the number of application traffic sent and received, as well as the numbers and kinds of all types of AODV messages. These numbers were then used to compare the models. The intent of this comparison is to gain an understanding if the simulation generated by *Syntony* functions correctly. This encompasses both the basic functionality to discover routes and forward data messages, as well as the question how efficient, i.e. with how many control messages, the basic functionality can be achieved.

The results are shown as boxplots. For each data set, a box is drawn from the first quartile to the third quartile, and the median is marked with a thick line. Additional whiskers extend from the edges of the box towards the minimum and maximum of the data set, but no further than 1.5 times the interquartile range. Data points outside the range of box and whiskers are considered outliers and drawn separately. Although we have recorded simulation results using all three traffic variants, we use only the first variant in comparisons. The reason for this is that our model does not buffer application messages. It therefore suffers from a comparably low application delivery ratio in all scenarios where the interarrival time of application messages is at least partly larger than the active route timeout.

	Scenario	App sent	App rcvd	ratio
Syntony	1	48006	47902	0.997
OMNeT++	1	48038	47998	0.999
Syntony	3	24004	23953	0.997
OMNeT++	3	23986	23527	0.981

Table 4: Application message delivery rates (in numbers of packets received/sent)

The correctness of the functional behavior is best illustrated using the first and third scenarios described above. The first scenario validates whether messages that a node sends to itself arrive correctly, while the third scenario validates the ability to setup and use longer routes. Table 4 shows the message delivery rates for both scenarios and both models. It is evident that the simulation generated from *Syntony* performs both tasks as well as the hand-coded simulation.

Figure 9 serves to further analyze the functional behavior. The first four columns of each sub-figure display the total

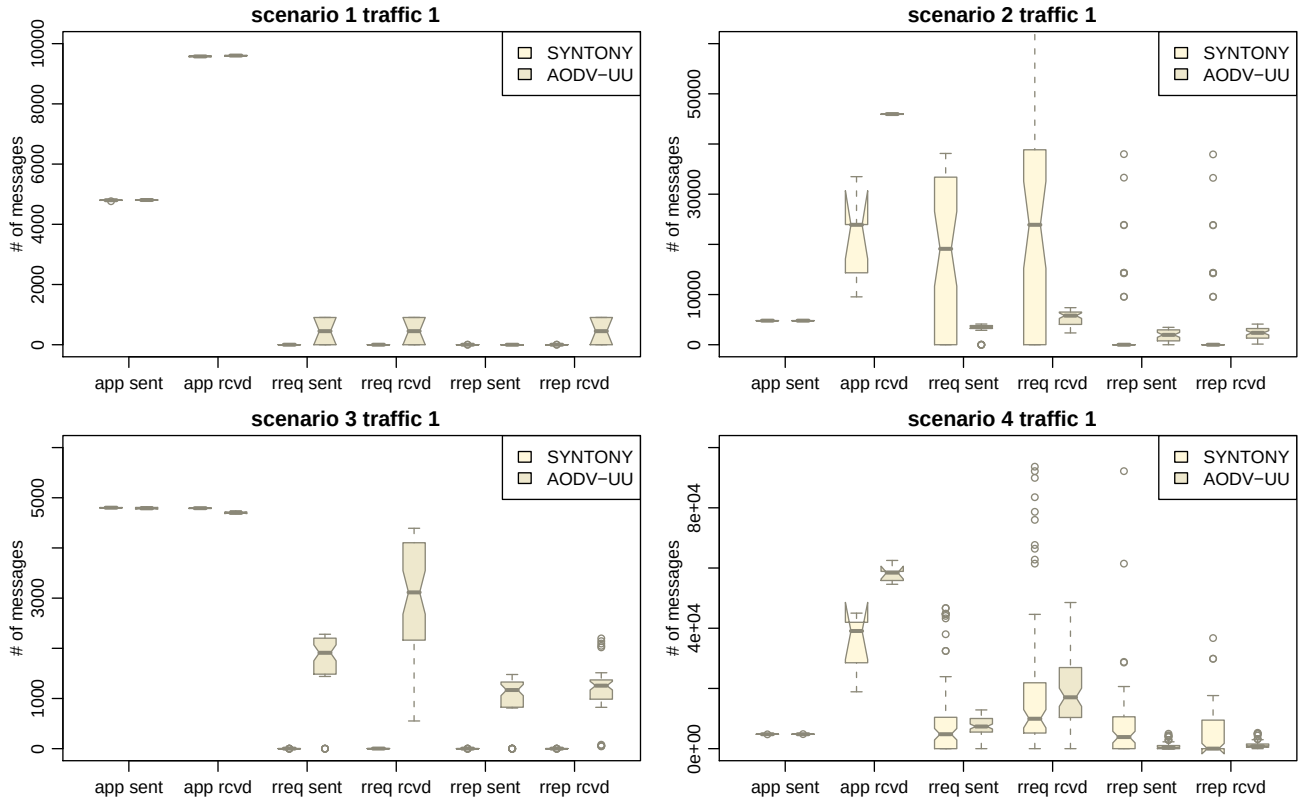

Figure 9: Quantity of the different message types

numbers of application messages sent by each node, and received correctly by the sink for both simulations. In the first and third scenarios, both models deliver nearly all application messages correctly. In the second and fourth scenarios, the *Syntony* model delivers significantly less messages than the OMNeT++ model. In these scenarios, there are a lot more traffic sources (10 and 15) than in the other scenarios (1 and 2). Through the higher number of wireless transmissions, there is also a higher number of messages getting lost due to mac layer collisions. In the AODV-UU model, these collisions are minimized by an artificial jitter introduced before message transmissions. Therefore, the seemingly better performance in the OMNeT++ model stems from an artificial effect that is not present in the *Syntony* model.

To analyze how efficient both models perform their routing tasks, we compare the quantities of the different messages types. In that way, we can gain an understanding how many control messages were needed to transmit a data message. Figure 9 shows this comparison for the four scenarios. It can be seen that in the second and fourth scenarios, the *Syntony* model needs relatively many control messages, which is due to the mac layer collisions explained above. However, in the first and third scenarios, the AODV protocol in *Syntony* performs even better than the hand-coded AODV-UU model.

5.5 Comparison of run-time performance

To compare the run-time performance of the two AODV models, we recorded the CPU usage for one simulation run of each scenario and traffic variant for both models. We also recorded some event statistics, like the total number of events generated during the simulation, and the number of events per simulated second. Figure 10 shows a comparison of these measures for the first traffic variant of each scenario. It can be seen that the simulation generated by *Syntony* performs better than the native implementation: the simulation run-times are shorter, and less events are generated.

However, it is likely that at least a part of this is due to the additional features implemented in the AODV-UU simulation. The two main features we left out in our UML model are hello messages and route error messages. Therefore, it is important to analyze how these features affect the run-time performance.

Acting on the assumption that the most time-consuming aspect in a simulation is the generation and handling of messages, we first measured how many of the generated AODV control messages are hello messages or route errors. It turns out that the amount of route error messages is quite small, in most cases less than 1% of the total number of control messages. In contrast, the fraction of hello messages ranges well above 20%, reaching up to 75% of all control messages depending on the scenario and traffic variant.

We therefore continued to examine the impact of the hello messages by increasing the interval for the hello messages from one second to 100 seconds. This effectively reduces the amount of hello messages by 99%. However, the reduction only led to an improvement of about 15% of the three run-time statistics. Also, the run-time performance of the *Syntony* simulation was still better. Corresponding to the relatively small amount of route error messages, their im-

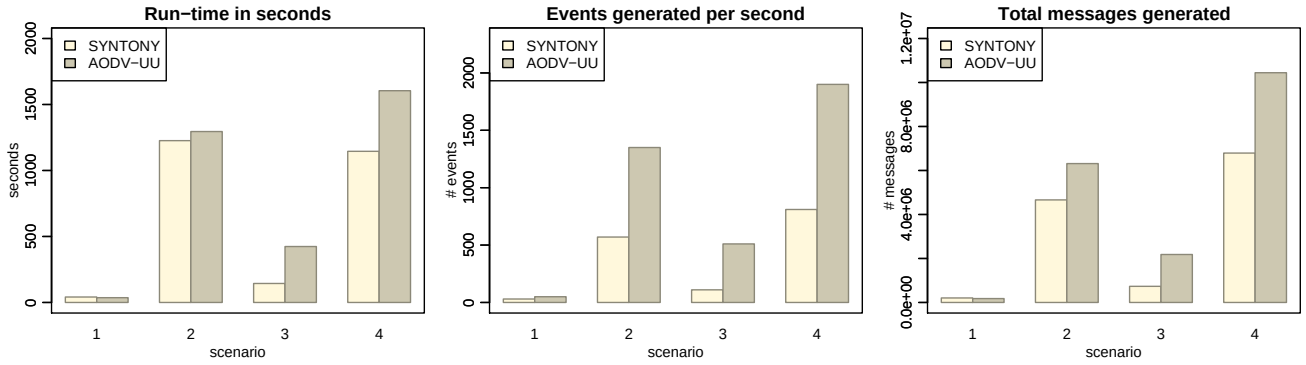

Figure 10: Run-time comparison

pact on the run-time performance should be even smaller, so we did not analyze it further.

This analysis shows that the run-time performance of a UML-driven simulation with *Syntony* is at least comparable to that of a hand-coded model. However, it is important to note that the run-time performance greatly depends on the simulated model. The transformation from UML to simulation could be very efficient, not generating any additional events and handling everything optimally. However, this can be easily negated if the modeler does not take great care how the system is modeled. If several internal events are generated for one incoming message, efficiency goes down the drain.

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 and the final 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 OAL or C++ code to modeled actions. The usability of the tool chain has been verified by a comparison of a self-developed UML 2 model of AODV and a native OMNeT++ implementation.

This demonstrates that the UML 2-based modeling approach based on *Syntony* is well suited for large applications: it is flexible, easy to use, and can handle complex modeling situations. 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 more convenient graphical modeling and programming paradigms while achieving a similar accuracy and simulation performance compared to native models.

In our 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.

7. ACKNOWLEDGMENTS

This work was partially funded by the Fraunhofer Institute for Integrated Circuits IIS, department for Communication Networks. We would also like to thank the anonymous reviewers for their valuable comments.

8. REFERENCES

[1] Accelerated Technology. Object Action Language Manual. Technical report, Embedded Systems Division of Mentor Graphics Corporation, 2004.

[2] 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.

[3] 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.

[4] S. Balsamo and M. Marzolla. Simulation Modeling of UML Software Architectures. In *European Simulation Multiconference*, Nottingham, UK, 2003.

[5] 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.

[6] 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.

[7] I. Chakeres and E. Royer. AODV Routing Protocol Implementation Design. In *International Workshop on Wireless Ad Hoc Networking (WWAN)*, Tokyo, Japan, March 2004.

[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] S. Das, C. Perkins, and E. Royer. Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks. In *19th IEEE Conference on Computer Communications (IEEE INFOCOM 2000)*, pages 3–12, Tel Aviv, Israel, March 2000.

[10] 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.

[11] N. de Wet and P. Kritzinger. Using UML Models for the Performance Analysis of Network Systems. *Elsevier Computer Networks*, 49(5):627–642, 2005.

[12] 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.

[13] 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.

[14] ITU-T. ITU Recommendation Z.109: âĂIJSDL Combined with UMLâĂİ. Technical report, International Telecommunication Union, 2000.

[15] 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.

[16] M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universit'a CaâĂŹ Foscari di Venezia, 2004 2004.

[17] M. Marzolla and S. Balsamo. UML-PSI: the UML Performance SImulator. In *1st International Conference on the Quantitative Evaluation of Systems (QEST)*, 2004.

[18] 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.

[19] 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.

[20] Object Management Group (OMG). Unified Modeling Language Specification (Action Semantics). Technical report, OMG, 2001.

[21] Object Management Group (OMG). UML 2.1.1 Superstructure Specification. Technical report, OMG, 2007.

[22] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, July 2003.

[23] C. Perkins and E. Royer. Ad hoc On-Demand Distance Vector Routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, February 1999.

[24] R. Pooley and P. King. The Unified Modelling Language and performance engineering. *IEE Proceedings Software*, 146(1):2–10, February 1999.

[25] A. Varga. The OMNeT++ Discrete Event Simulation System. In *European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, 2001.