Formal Methods for Abstract Specifications – A Comparison of Concepts

Martin Instenberg, Axel Schneider, Sabine Schnetter, Ulrich Heinkel
Lucent Technologies Network Systems GmbH, Nürnberg
{instenberg, aschneider, sschnetter, heinkel}@lucent.com

Kim G. Larsen, Gerd Behrmann
Aalborg University
kgl@cs.auc.dk, behrmann@cs.aau.dk

Abstract

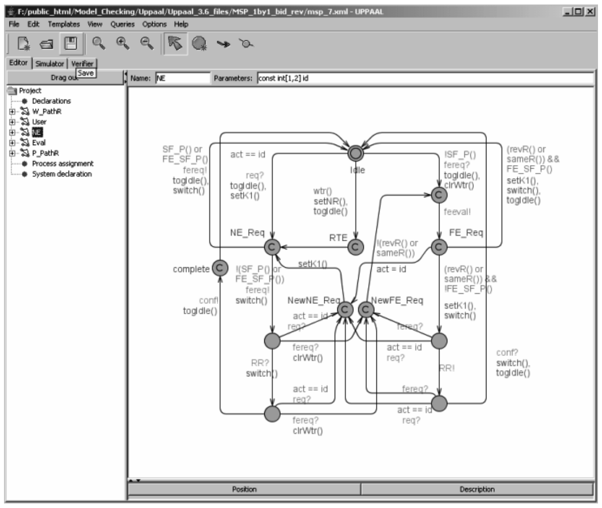
In industry formal methods are becoming increasingly important for the verification of hardware and software designs. However current practice for specification of system and protocol functionality on high level of abstraction is textual description. For verification of the system behavior manual inspections and tests are usual means. To facilitate the introduction of formal methods in the development process of complex systems and protocols, two different tools evolved from research activities – UPPAAL and SpecEdit – have been investigated and compared regarding their concepts and functionality. For this purpose both tools were applied to comparable frameworks.

1. Introduction

Usually the development of complex telecommunication systems is split into a specification and an implementation phase, each performed by different groups of engineers. To guarantee a correct realization of the system requirements it is of high importance that the documents, which describe the intended system behavior, are complete and well defined. Specifications in natural language are not able to satisfy these claims since they tend to be ambiguous and their correctness cannot reliably be proven. As a consequence the implementing engineer must have deep knowledge about the desired system behavior and is forced to make own decisions about the system behavior – which might be different from the intentions of the specification engineer. Defects in specifications usually are discovered in late phases of the development process and their correction causes unforeseen costs and delays. Getting use of formal methods for the specification and verification of complex system and protocol functionalities is expected to improve and to shorten the whole development process and to lead to more reliable and competitive products. Formalized specifications are unambiguous and their correctness can be proven automatically through software tools. Their data are easier to manage and to reuse than documents with up to several thousand pages of plain text. As a first approach UPPAAL and SpecEdit were used to model and verify the behavior of existing system functionalities based on their informal specification.

2. UPPAAL

UPPAAL is a tool suite for modeling, simulation and verification of real-time systems jointly developed by the Universities of Uppsala (Sweden) and Aalborg (Denmark). [1]


F:/public_html/Model_Checking/Uppaal/Uppaal_3.6_files/MSP_1by1_bid_rev/msp_7.xml - UPPAAL

Name: NE   Parameters: const int[1,2] id

Figure 1. UPPAAL GUI

The software is split into a graphical user interface (GUI) implemented in Java and a verification engine written in C++, both available for most common platforms. UPPAAL is designed for the formal verification of systems that can be represented as networks of timed automata extended with elementary and structured data types and channel synchronization. Model checking techniques are used to prove the correctness of the specification. It has been successfully applied in several industrial case studies concerning the verification of time critical applications like multimedia and communication protocols, e.g. [2].

2.1. System descriptions in UPPAAL

UPPAAL is based on the theory of timed automata (TA), which are finite state machines (FSM) extended with clock variables. System descriptions consist of several networked automata while time progresses globally at the same pace, i.e. UPPAAL uses continuous time. Clock variables are used to represent time and permit measuring of time progress. Thus the behavior of a timed automaton can be restricted by clock constraints. As extension to timed automata, UPPAAL supports bounded integer variables, constants, Boolean, scalars and communication channels. Furthermore multi-dimensional arrays and C-like records containing these elementary data types may be declared. Besides the GUI, the tool suite provides a basic programming language for UPPAAL's extended timed automata, which facilitates specifying systems without the graphical editor. An entire system description is composed of several concurrent processes and globally declared variables and functions. Basically the automaton has multiple locations with edges in between (also called transitions) to pass along from one to the other location. Beside the "normal" and *initial locations* there are *urgent locations* and the even more restrictive *committed locations*. Once the system is in an urgent or a committed location, time is not allowed to pass. Additionally a committed location must be left in the successor state otherwise the system is deadlocked. Special clock constraints on locations, so-called *invariants*, enforce progress in the system. Edges may be decorated with *selects*, *guards*, *synchronizations* and *updates*. The *select* label allows binding non-deterministically certain values of scalar types to identifiers. These identifiers are available as variables within the other labels. A guard is a condition on variables and clocks saying when the respective edge is enabled. UPPAAL actually offers three kinds of communication channels to synchronize two or more processes:

- *binary synchronization***:** An edge labeled as sender synchronizes with another edge labeled as receiver – both processes take a transition at the same time.
- *broadcast synchronization***:** One sender can synchronize with multiple receivers.
- *urgent synchronization***:** Delays must not occur if a synchronization transition on an urgent channel is enabled, hence clock constraints are not allowed on these edges.

One transition is non-deterministically chosen to execute if several combinations are enabled at the same point of time. Assignments to variables or clocks can be performed when a transition is taken (update). The ability to use C data types and expressions includes calls to user declared *functions*, which are evaluated as atomic statements. These functions can be used in updates on edges as well as in guards if their return value can be mapped to Boolean. They help to keep the system description clear and to reduce the need for committed locations.

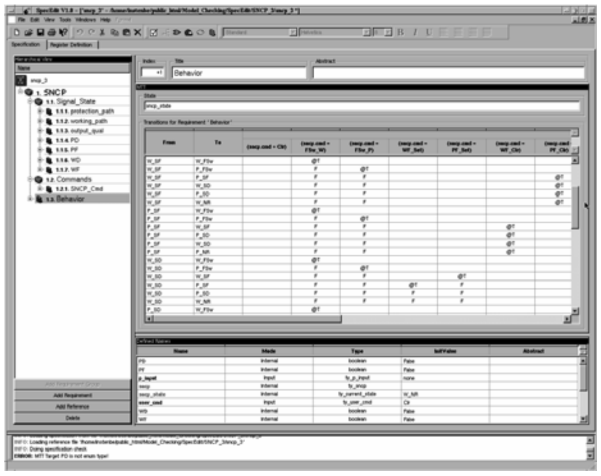
2.2. Simulation and formal verification

UPPAAL provides an integrated simulation tool. It allows the user to examine the dynamic system behavior in a graphical manner – either interactively, i.e. the user chooses which enabled transition to take, or randomly to let the system run on its own. In addition execution traces can be loaded, saved and imported from the verifier to follow them step by step. In contrast to the formal verification, simulation can only explore a particular execution trace instead of the whole reachable state space. However, it is an inexpensive approach to fault detection, especially in early stages of defining a system concept, as well as for small systems. By exploring of the whole reachable state space UPPAAL's integrated symbolic model checker proves the modeled system behavior against the desired behavior described in a set of requirement specifications. Requirement specifications (properties) can be defined using a subset of CTL (computation tree logic) restricted to operators for *reachability*, *safety* and *liveness* properties. In addition there is the possibility to search for deadlocks.

The GUI provides an environment for managing queries and launching the verification. The verification engine itself can thereby be executed on the local workstation or on a (more powerful) remote system connected via TCP/IP. Several possible adjustments concerning the search order (breadth/depth first), the state space representation and reduction as well as the type of the diagnostic trace (shortest/fastest) are helping to optimize the verification results. Besides the conclusion that a certain property is or is not satisfied the model checker may create such a diagnostic trace as a witness for the correctness or incorrectness of the specific requirement. This trace can be loaded directly into the simulator for the subsequent system debugging.

3. SpecEdit

SpecEdit has been developed by Lucent Technologies Nürnberg as a tool solution for the compilation and maintenance of specifications based on the formal specification language **ADeVA** (**A**dvanced **De**sign and **V**erification of **A**bstract Systems) [3]. Initially it has been designed for the specification and verification of ASICs [4]. SpecEdit provides a platform for the structured management of specification data, proving their consistency and plausibility and rendering them into different models e.g. VHDL, C or SMV. These models as well as the original specification data allow further processing by other tools for code generation, model checking etc.



```
INFO: Loading reference file '/home/roberbe/public_html/Model_Checking/SpecEdit/SNCP_3/sncp_3'
INFO: Doing specification check
ERROR: MTT Target PD is not enum type!
```

Figure 2. SpecEdit GUI

3.1. System descriptions in ADeVA

The table based specification language ADeVA provides semantics for the formal specification of abstract system functionality and communication protocols. With ADeVA, the control flow of a system is modeled by asynchronous parallel automata and abstract data types. A system description consists of a set of user declared variables (also called signals) and several automata represented through two kinds of tables: **M**ode **T**ransition **T**ables (MTT) and **D**ata **T**ransformation **T**ables (DTT).

MTTs describe the behavior of state-machine driven signals. Hence, a particular signal value is also called state. The table shows the appropriate states as well as the conditions and the corresponding logical values causing state transitions. **T**/**F** stands for **T**rue/**F**alse and "-" for "don't care". The '@' symbol is used to indicate an event, i.e. a change of the value either to **T**rue or **F**alse. Transitions are taken as soon as all conditions are fulfilled and without time consumption. All rows in an MTT are required to be disjoint, hence a non-deterministic choice between multiple possible transitions at a distinct state is prohibited.

From	To	Condition1	Condition2	Condition3
state_a	state_b	@T	F	-
state_a	state_c	@T	T	-
state_b	state_c	F	@F	T

Figure 3. Mode transition table (MTT)

As MTTs are not sufficient to model realistic systems, DTTs were introduced to describe the change in value of a data signal. DTTs can be interpreted in the same way like MTTs and they have to follow identical restrictions. The major difference between these two kinds of tables is, that regarding DTTs the previous value of the signal is not relevant.

Output	Value	Condition1	Condition2	Condition3
signal_a	a	@T	T	-
signal_a	b	@T	F	T
signal_a	c	-	-	@F

Figure 4. Data transformation table (DTT)

Basically ADeVA differentiates between three kinds of user declared system variables: *input*, *internal* and *output* signals. Supported data types are integer and Boolean as well as user defined types like enumeration, array and record. Input signals allow integrating external signals (e.g. user inputs) in the abstract model of a system. All variables are allowed as parameters for conditions. In case they aren't already of type Boolean their value has to be mapped to the Boolean domain. In practice usually MTTs are used to model the behavior of a specific system component whereas signals for communication between these components are described in DTTs.

3.2. Simulation and formal verification

To ensure the consistency and correctness of all MTTs/DTTs SpecEdit offers static checks using satisfiability checking techniques. It proves (amongst others) the determinacy of state transitions, the satisfiability of transition conditions and advises against dead end states. For the dynamic verification of ADeVA specifications like simulation and model checking SpecEdit itself does not provide own features. Due to the integrated code generation tools, however, the models generated from ADeVA specifications can be further processed by other commercial or non-commercial applications. With the current SpecEdit version VHDL (for GateProp) [5], SMV (for NuSMV) [6] and C models (for VeriSoft) [7] may be created. In addition VHDL and SMV models can be used within a variety simulation tools. This leaves the choice to the user and keeps the system flexible and open for many purposes.

4. UPPAAL and SpecEdit in comparison

4.1. General

One of the challenges concerning formal specification and verification is to map the desired system behavior described in natural language to the formal concept of the according tool solution. This is difficult not only because of the ambiguousness of textual specifications and hence the broad scope of possible interpretations. Inherently the formal specification enforces engineers to formulate correct requirements and leads to less inconsistency. In case of functional system specifications on a high level of abstraction neither UPPAAL nor SpecEdit are designed for that purpose. Providing high-level abstraction facilities will reduce the complexity of system models. Based on experiences of several case studies (e.g. [5]) the following paragraphs describe the advantages and disadvantages of both tools with regard to the use for such high-level specifications.

4.2. Tool concept

First of all, one significant difference between UPPAAL and SpecEdit is, that UPPAAL is a monolithic system. That means all components (i.e. system editor, simulator and verification engine) are designed to work together and on one and the same underlying model. This is very comfortable for the user. Few tools for translating UPPAAL models to other languages exist but the UPPAAL developers do not distribute them. Since SpecEdit does not provide facilities for simulation and model checking it must be used together with other applications for that purpose. The integrated code generator ensures the mapping of the ADeVA tables to the semantics of the languages interpretable by the applied model checkers or simulators.

4.3. Specification

UPPAAL’s graphical editor allows modeling system behavior in a very intuitive and convenient manner. Additionally graphs of single automaton may be stored in a postscript file and thus integrated into documents. However, complex automata with a high level of alliance demonstrate the advantages of a table-based system: the comparatively uncomplicated dealing with extensive specification data. These tables may also be exported and reused during the design and implementation process. A combination of both approaches would be appreciated as well as the possibility to model hierarchical automata structures - a desirable but missing feature in both tool solutions. It would significantly improve the handling of complex specifications and allows examining the system behavior on different levels of abstraction.

An important fact concerning the model concepts of the tool is that UPPAAL allows non-determinism in its automata. That means a certain state can have multiple successor states, which may be reached under the same transition conditions. SpecEdit explicitly prohibits non-determinism in ADeVA models and it proves the determinacy of all tables. The possibility of having non-determinism in the formal specification may be of advance, but when the model is intended to be deterministic a check – like SpecEdit performs it – would be of avail.

Another basic differentiation between the tools is the understanding of state transitions. In ADeVA all possible transition are taken as soon as the according conditions are satisfied (maximal progress). This is of advantage when modeling parallel working processes. On the other hand UPPAAL uses asynchronous composition – in every step it chooses one of the enabled edges to take. Additional, it is possible to model synchronous composition by synchronizing all edges in system via a broadcast channel. Unfortunately, these approaches are barely compatible and thus translating UPPAAL specifications into SpecEdit specifications and vice versa seems to be highly challenging.

Templates are the way to model reusable components in UPPAAL. A comparable feature for SpecEdit is planned, but not yet available. This tool component will also facilitate the integration of ADeVA tables from other projects whereas UPPAAL templates are reusable only in the same project.

As mentioned in section 3.1, SpecEdit allows declaring signals as input signals. The applied model checker can choose their according values non-deterministically. However, not all verification tools support such dedicated input signals. The select on edges provides a comparable function in UPPAAL.

In SpecEdit formal specification data can be explained by textual descriptions directly attached to the according MTTs and DTTs. This combination unites the comprehensibility of informal descriptions with the unambiguity of formal methods. The integrated document generator facilitates to create an entire document from all specification data. This will also be possible in future versions of UPPAAL. Despite this, introducing formal methods in the product development process requires a tighter integration with project management and document management tools.

4.4. Simulation

The integrated simulator is, compared to similar solutions, definitely an advantage of UPPAAL. It allows exploring the state space manually and following diagnostic traces step by step in an intuitional usable and clearly arranged environment. This is an enormous help to find flaws in the model or in the underlying textual system description.

SpecEdit itself does not provide simulation facilities, but VHDL and SMV models generated from ADeVA tables can be processed in all prevalent VHDL/SMV simulators. To some extend these tools are distributed commercial and must be purchased.

4.5. Formal verification

Concerning the verification, unfavorable the set of temporal operators in UPPAAL is rather small and the inability to use them nested limits the expressiveness of the query language. To check more complex properties often the model has to be extended with locations or variables. Then again UPPAAL's verification engine may be well used to track deadlocks. This is very useful when proving if a specific model is well defined and complete. In addition to that the model checker provides a lot of options that help to optimize the verification process and to decrease the time and memory consumption.

SpecEdit's integrated tools can perform static checks concerning the consistency and plausibility of ADeVA descriptions. For the dynamic verification of SpecEdit specifications commercial as well as free model checkers exists. This way there are a number of powerful property languages to choose from (e.g. CTL, LTL, PSL).

4.6. Conclusion

Formal specification and verification of complex systems requires a high level of abstraction to avoid state explosion. UPPAAL and SpecEdit were used to formalize textual system requirements of a telecommunication device on a high level of abstraction. Both tools originally were designed for different purposes, hence each demonstrated its own advantages. To specify such systems in an efficient manner a combination of the individual strengths of UPPAAL and SpecEdit would be appreciated.

For instance the graphical editor of UPPAAL allows modeling system behavior in a very intuitive and convenient manner. In combination with its integrated simulator UPPAAL offers means to detect and investigate flaws in the model already during modeling.

On the other hand SpecEdit with its table based specification language ADeVA and its integrated consistency and plausibility checks simplifies dealing with extensive specification data. It is convenient to decompose a specification into single requirements with text (as usual) but supplemented and concretized with formal specification data. This makes the specification easier to understand by other people while being able to utilize the formalized data.

While UPPAAL has its own formal verification facilities integrated, SpecEdit provides code generators as interfaces to several external formal verification tools. It has been successfully demonstrated that both solutions are suitable for the formal verification ofabstract system specifications. The next challenge will be to integrate formal methods right into the specification process of future systems.

5. References

[1] Behrman, David, Larsen: "A Tutorial on UPPAAL", *LNCS 3158.*

[2] Havelund, Soku, Larsen, Lund: "Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using Uppaal", *In the proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 2-13, San Francisico, 3-5 December 1997.

[3] Haas, Gossens, Heinkel: "Semantics of a Formal Specification Language for Advanced Design and Verification of ASICs (ADeVA)", *11th E.I.S.-Workshop*, Erlangen, April 2003.

[4] Haas, Gossens, Heinkel: "Integration of Formal Specification into the Standard ASIC Design Flow", *7th*

IEEE/IEICE International Symposium on High Assurance Systems Engineering, Tokio, 2002.

[5] Schneider, Bluhm, Renner et al: “Formal Specification and Verification of Abstract Telecommunication Protocol Definitions”, *9th GMM-Workshop*, Dresden, February 2006.

[6] Cimatti, Clarke, Giunchiglia et al: “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”, *International Conference on Computer-Aided Verification*, Copenhagen, July 2002

[7] Schock, Dinkel, Heinkel et al: “Comparison of Techniques for the Automatic Verification of ADeVA Specifications”, *Dresdner Arbeitstagung Schaltungs- und Systementwurf*, Dresden, April 2005.