From Business Modeling to Verified Applications

Christian Ammann and Stephan Kleuker
Hochschule Osnabrück, University of Applied Sciences
Postbox 1940, 49009 Osnabrück, Germany
{c.ammann, s.kleuker}@hs-osnabrueck.de

Elke Pulvermüller
University of Osnabrück
Albrechtstr. 28, 49076 Osnabrück, Germany
elke.pulvermueller@informatik.uni-osnabrueck.de

Abstract: UML activity diagrams can be used to model business processes which are implemented in a software project. It is a worthwhile goal to automatically transform at least parts of UML diagrams into software. Automated code generation reduces the total amount of errors in a software project but the model itself can still violate specified requirements. A quality improvement is the usage of a model checker which searches through the whole state space of model and checks whether all requirements are met. A model checker requires a formal description of a model for a complete verification. Activity diagrams often describe processes informally which is difficult to verify with a model checker. We therefore propose the transformation of activity- to statechart diagrams which allow a more detailed and formal description. Several algorithms exist to map UML statecharts into a model checker input language for a successful formal verification. Afterwards, the model checker searches through the whole state space of a statechart and therefore has to store each state in memory. UML statecharts can reach a high degree of complexity which is problematic for a complete state space traversal because the total amount of available memory is exhausted. Accordingly, we present the domain specific language UDL (UML Statechart Modeling Language) and a transformation from UDL into the Spin model checker input language Promela. UDL contains features for property preserving abstraction which reduces the models state space and therefore the memory consumption of a model checker. Furthermore, we introduce an optimisation technique for the transformation process from UDL to Promela which focuses on a reduced model checker run-time. A case study with a movement tracking system demonstrates how our approach could significantly reduce the memory consumption of a model checker and allows the verification of complex models.

1 Introduction

The development of a software system usually starts with an informal description of its features expressed in natural language. Such informal descriptions can be read and understood by persons without programming knowledge but have the disadvantage of being imprecise and ambiguous.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

The *Unified Modeling Language* (UML) [OMG10] is developed by the *Object Management Group* (OMG) [omga], contains several diagram types for requirements, structural and behavioral aspects and allows a more precise description of software systems. UML activity diagrams can be used to model behavior as a business process and allow the modelling of data and control flows in a system. They have the advantage of being understood by domain experts for business modeling without programming knowledge. Another diagram type for behavioral modeling are statecharts. UML statecharts are finite state machines and equivalent to *Harel Statecharts* [Har87] in an object oriented context. They allow a more detailed modeling than activity diagrams and can be used to implement the behavior of classes.
It is a worthwhile goal to automatically transform at least parts of certain UML diagrams (and other models) into software. *Model-Driven Development* (MDD) [Sch06] is a software development process which focuses on the creation of models and their transformation. Accordingly, parts of a software system are written manually, other parts are generated automatically from models. Additionally, developers have to write generators for the model transformation process. The advantages of MDD and the automated code generation are:

- The total amount of errors is reduced. MDD is therefore a contribution to quality assurance.
- Domain experts with no programming knowledge can participate in the software development process.

Domain specific languages (DSLs) can be used to implement textual models. Tools like *Xtext* and *Xpand* [xte] allow the fast creation of DSLs, their transformation and an easy integration in software projects. Xtext and Xpand are developed as plugins for the *Eclipse IDE*.
MDD reduces programming errors but it is still possible that the models are incorrect. A model checker [CGP00][Kle09] can be used to verify a model for correctness. Models and their requirements have to be transformed into a model checker input language for a successful formal verification. It is therefore important to extend the MDD approach: Models are enriched with formal requirements (e.g. assertions or LTL-formulas [Pnu77]) and a second code generator is written which transforms the model into a model checker input language. The model checker verifies the complete state space of a model and typically generates an error path if a requirement is not met. The model is modified until the model checker proves the correctness of the model. Afterwards, it can be used for source code generation.
This paper investigates whether the combination of MDD and model checking can be applied to an expansion for the 3D movement tracking system *AssyControl* [sof] which is developed by the *Soft2tec GmbH* for the *Otto Kind AG*. AssyControl is described in more detail in section 2 and monitors the manual assembly process of workers at a workbench.
In this paper we model parts of AssyControl's behavior as business processes with UML activity diagrams. Activity diagrams can be used to model informally at a high level of abstraction but some features like the specification of requirements are missing. Therefore, a more formal description of behavioral aspects is necessary for the verification of

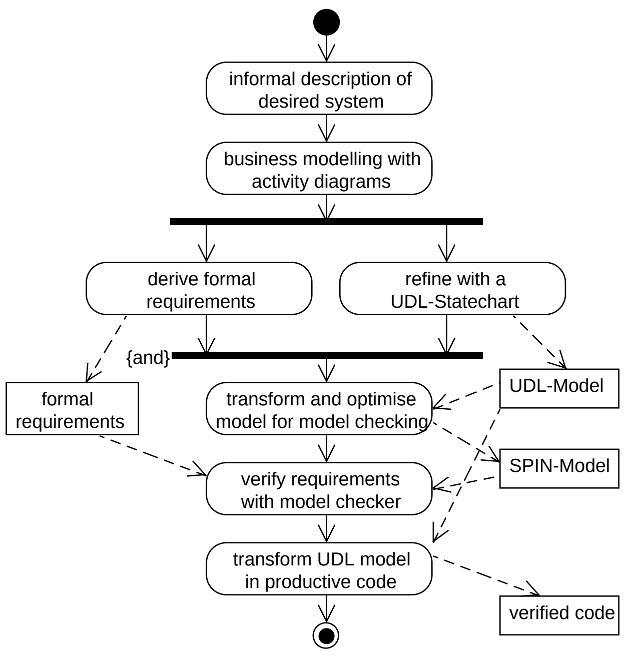
informal description of desired system
business modelling with activity diagrams
derive formal requirements
refine with a UDL-Statechart
{and}
formal requirements
transform and optimise model for model checking
UDL-Model
SPIN-Model
verify requirements with model checker
transform UDL model in productive code
verified code

Figure 1: From business models to verified applications

business models. This paper proposes the following methods to solve this problem and verify more complex models based on business processes: We present in section 3 a mapping from activity- to statechart diagrams and an enrichment with more implementation details. The corresponding UML statechart is transformed into the *Spin* [Hol03] model checker input language *Promela* in section 4. The statechart diagram of the AssyControl case study reaches a high degree of complexity which makes a state space traversal for a model checker more difficult because of the state space explosion [CG$^+$01].
Accordingly, we use Xtext to develop the domain specific language UDL (UML-Statechart Description Language) which is designed for code generation and an efficient formal verification of statecharts with a model checker. UDL supports only a subset of the UML statechart features which can be easily transformed into the Spin model checker input language Promela. An action language allows the implementation of statements and expressions. Features can be implemented in detail or with property preserving abstractions to reduce to models state space. Furthermore, we present some optimisations for the transformation process from UDL to Promela which reduce the memory consumption of the model checker. The complete approach of this paper is visualized in figure 1.
Measurements in section 5 demonstrate how our approach could significantly reduce the run-time and memory consumption of the model checker. Related work is presented in section 6. An overview about further work is given in section 7.

This paper is part of the *KoverJa Project* [kov] and is supported by the German Federal Ministry of Education and Research (BMBF). KoverJa aims at the increase of software quality in (distributed) Java applications which includes methods like testing, model-driven development and formal verification.

2 AssyControl

AssyControl is a 3D tracking system developed by the *Soft2tec GmbH* for the *Otto Kind AG*. Its main emphasis is quality control in industrial assembly processes. It consists of the following main components:

- Computer with touch display
- Basic unit (ultrasound receiver)
- Marker (ultrasound sender)

A worker wears two markers on his hands. Every marker generates an ultrasound signal. The basic unit is connected with the computer and receives those signals. The computer calculates the 3D coordinates of each marker. Markers can be installed at each part of the worker's body. Therefore, AssyControl can keep track of the worker's movements. It is also possible to extend the system with additional I/O ports to connect different sensors, traffic lights, etc.
AssyControl gets continuous information about the positions of the worker's hands during the assembly process. The worker grasps components, moves and assembles them. AssyControl observes each step of the worker and warns him, when an error occurs (e.g. assembling two components in the wrong order) and proposes a solution for the problem if possible.
To monitor the assembly process of a certain product, AssyControl has to be programmed via the touch display with the following information by a production manager:

- Components: Entities which are assembled by a worker at a workspace
- Positions: The whole workspace is divided into 3D areas called positions. A position represents an empty space, a box which contains components for the assembly process, etc. A component is associated with one or more positions.
- Recipes: A recipe is a list of components and specifies the order in which components must be grasped from positions and assembled with each other.

For AssyControl the assembly process is a sequence of positions reached by the markers: $p_i, p_j, ...$ It can keep track of the components and the workers assembly steps because of the information provided by the production manager.
It is important to verify the correct functionality of AssyControl. Errors in the AssyControl software can lead to the following problems:

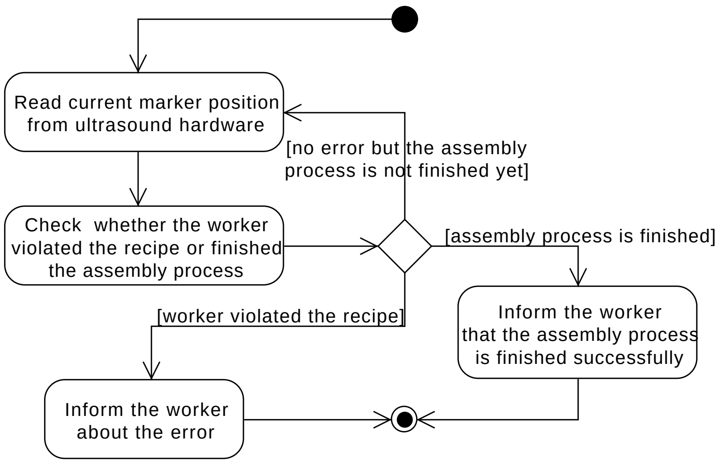
Read current marker position from ultrasound hardware
[no error but the assembly process is not finished yet]
Check whether the worker violated the recipe or finished the assembly process
[assembly process is finished]
[worker violated the recipe]
Inform the worker that the assembly process is finished successfully
Inform the worker about the error

Figure 2: Activity diagram of AssyControl

- False positives: AssyControl detects an error in the assembly process which is not an error at all.
- False negatives: A violation of the recipe is not detected by AssyControl.
- AssyControl hangs and has the be restarted (e.g. because of deadlocks).

Each kind of error described above slows down the assembly process, can lead to wrong assembly of components and therefore causes financial damage for the corresponding company.

3 From Activity Diagrams to Statecharts

We model AssyControl in figure 2 as a business process with an activity diagram and assume for simplification that a worker wears only one marker. AssyControl starts and reads the current position of a marker (the workspace is divided into several positions, see section 2 for details). Afterwards, AssyControl verifies that the new position does not violate the recipe. If no error occurred, AssyControl reads the current marker position again. This loop repeats until:

- The worker violated the recipe (e.g. grasped the wrong component). AssyControl informs him about the error and terminates.
- The worker has finished the assembly process without an error. AssyControl informs him that the process is finished and terminates.

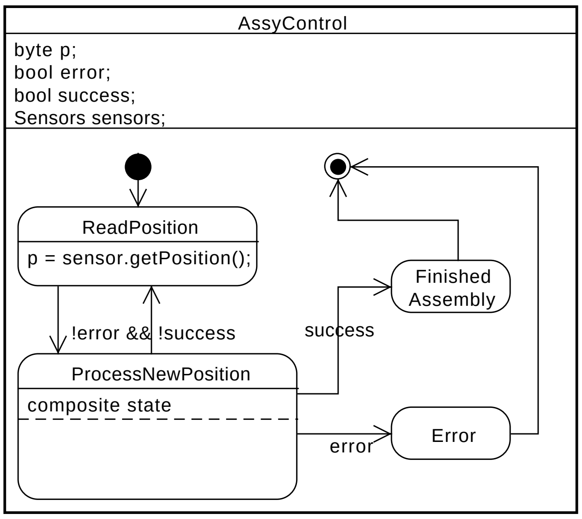

Figure 3: AssyControl as UML statechart diagram

The diagram in figure 2 can be easily read by a human but additional formalisms are necessary for a verification with a model checker or an automated transformation in a high-level programming language like Java.
We solve this problem with a mapping into UML statechart diagrams because they are similiar to activity diagrams but allow more detailed modeling. The mapping process is performed manually. An automated mapping from activity diagrams to statecharts is not in the scope of this paper (additional literature about transformations from business processes into software can be found for example in [OvdADtH06]). The corresponding statechart diagram is shown in figure 3. The state machine is embedded in the Java class *AssyControl*. It has a reference to a *Sensors* class object which controls the ultrasound hardware and is not shown in the diagram for simplification. The class *Sensors* exports the method *getPosition()* which returns the current position of the marker.
AssyControl reads the current position of the marker and stores it in the byte variable *p*. The composite state *ProcessNewPosition* checks whether an error occurred (e.g. the worker grasped components in the wrong order) or whether the assembly process is finished (the end of the recipe is reached). Afterwards, it sets the corresponding boolean variables *error* and *success*. If both evaluate to *false*, the next position is read. Otherwise, an error or the success of the assembly process is reported and the state machine terminates. A possible requirement which can be verified by a model checker is for example: $error \wedge success = false$ in each state.
Figure 3 does not show the complete implementation: The composite state *ProcessNewPosition* contains additional states which implement the behaviour of AssyControl in more detail. The whole statechart consists of 16 states and 35 transitions. On the code generation side, various tools [rha][bou] exist which can be used to model UML statecharts and

transform them into high-level languages like C++ or Java. On the verification side, it is important to find a way for an efficient verification of complex statecharts which consumes less memory to face the state space explosion problem. We will present our approach for an efficient verification of statecharts in the next section.

4 Verification of Statecharts

The OMG gives only an informal description of the dynamic semantics of UML statecharts. Therefore it is necessary to fill this gap before a model checker can be used to check whether all requirements of a statechart are met. This work adapts the formal semantics of UML statecharts defined in [LMM99].
A statechart and its requirements have to be translated into a model checker input language for a formal verification. We use the Spin [Hol03] model checker (with its input language Promela) because it focuses on the verification of concurrent software systems and their communication (which is essential in UML statecharts). Spin has native support for signal passing, is widely used and under constant development. Furthermore, other research projects like [LM$^+$99] build already the foundations for a formal verification of UML statecharts with Spin and we adapt parts of this work in this paper.
We transformed the statechart in figure 3 successfully to Promela but could not verify it because the complete state space exhausted the available memory. Therefore, we develop with Xtext the domain specific language UDL (UML Statechart Description Language) which contains a practical relevant large subset of statecharts identified in case studies and features to reduce the state space size of a model. We also present in this paper an optimized transformation process from UDL to Promela which reduces the memory consumption of the verifier. UDL supports the following statechart features:

- Initial, final, simple and composite states
- Entry and exit actions
- Asynchronous signal- and synchronous call events
- Transitions can contain a trigger, guard or effect
- Concurrency is implemented with multiple active classes which can communicate via signal events.

The semantics of these features is described in [LMM99]. UDL supports the data types *bool*, *byte*, *integer* and custom enumerations. Expressions and statements can be implemented with an action language which contains operators for arithmetic, boolean expressions, etc. UDL supports three different kinds of classes:

- *Active classes* contain a state machine and are executed as a seperate thread. Several active classes lead to concurrency.

- *Non-active classes* contain a set of functions which are implemented with UDLs action language and called by active classes as call events. They can be used to enrich a model with custom data types (e.g. a Stack) without the effort to implement them as a state machine.
- *Interface classes* contain a set of functions without a function body. They can be used for abstraction of components, when implementation is not important for the model checker or can not be described in the model checker input language.

The following listings demonstrate the concrete UDL syntax:

Listing 1: Declaration of an active class in UDL

```
active class AssyControl{
    byte p = 0;
    bool error = false;
    bool success = false;
    Sensors sensors;

    //...

    state::simple ReadPosition{
        entry{p = sensor.getPosition();}
        -> stop{}
    }

    //...
}
```

The UDL implementation corresponding to the active class AssyControl from figure 3 is shown in listing 1. It contains the state *ReadPosition* and a transition to the state *stop* (also part of the class AssyControl). The state *ReadPosition* executes an entry action when the state is entered. The entry action is written in UDLs action language and calls the method *getPosition()*. States are declared with the syntax <*state type*> <*label*> { ... }. Transitions are implemented with the language construct → <*destination*>{ ... }.

Listing 2: Interface class in UDL

```
interface class Sensors{
    (0,5) getPosition();
}
```

Listing 2 contains the class *Sensors*. It has access to the ultrasound hardware and its concrete implementation can not be expressed with Promela. Therefore, it is implemented as an interface class. A method of an interface class can return a data type or an integer range. Spin checks during the verification the system behaviour for all possible return types. The integer range limits the possible return value and is used to reduce the state space of a verification run. The method *getPosition()* returns the current position id and is therefore limited to a range from 0 to 5 (the *AssyControl* case study supports six different positions).

We adapt the transformation process from [LM$^+$99] and modify it to achieve a better runtime of the model checker. Non-active classes are implemented with Promela's *typedef* and *inline* elements. An active class is transformed to a Promela process and a global message queue. Composite states have to be transformed to non-composite states by a flattening algorithm. Several approaches [Was03][Was04] exist in the literature and describe how to convert hierarchical statecharts to flat statecharts. The next sections discuss two optimisations which can be applied on the generated Promela code.

4.1 Statement Merging

An advanced form of the statement merging technique [SH99] can be applied to single threaded Promela models. Spin handles each Promela statement as a separate state. Statements of the action language can be encapsulated by so called *d_step blocks*. The content of a *d_step block* is handled by Spin as a single statement but has to be non-blocking and deterministic. The usage of *d_step blocks* reduces the total amount of states during a verification run and therefore the memory consumption.
The following two Promela examples demonstrate the approach:

```
byte a; byte b;
a=1;
b=2;
b=a*b;
```

The example above consists of three Promela statements. Spin stores during verification each Promela statement as a separate state in memory. A state has a size of 2 bytes (the local variables a and b). Therefore, the verification of the example above has a total memory consumption of six bytes. The next example contains the statement merging technique:

```
byte a; byte b;
d_step{
    a=1;
    b=2;
    b=a*b;
}
```

The three statements are encapsulated in a *d_step block*. Spin handles the content of this block as a single state which leads to a total memory consumption of 2 bytes during verification.

4.2 Reset of Auxiliary Variables

Auxiliary variables which are used in the entry or exit action of a state are reset to their initial value when the state becomes inactive [Rui01]. The statechart in figure 4 is used to demonstrate this approach. It contains the class *StateMachine*, which reads two user

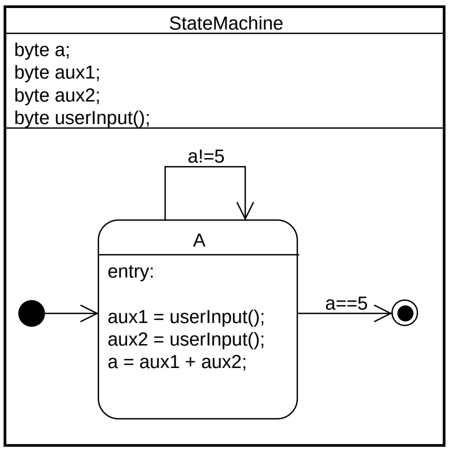

Figure 4: Simple statechart to demonstrate the reset of auxiliary variables

inputs from a keyboard (using the method *userInput()*). It stores each input in an auxiliary variable ($aux1$ and $aux2$), adds both, stores the result in the local variable a and terminates if $a == 5$. Otherwise, it reads the next user input, etc.
We assume that the user enters the following input pairs: $(1, 3)$, $(2, 2)$ and $(3, 1)$. The local variable a contains in each case the value 4 and the state machine does not terminate. Spin stores the following states in memory during verification:

- $aux1 = 1$, $aux2 = 3$, $a = 4$
- $aux1 = 2$, $aux2 = 2$, $a = 4$
- $aux1 = 3$, $aux2 = 1$, $a = 4$

The auxiliary variables are used only in the entry action and can be reset when leaving the simple state A. The reset of $aux1$ and $aux2$ in an exit action of A has the following advantage: Spin stores for the three different user input pairs only one state in memory ($aux1 = 0$, $aux2 = 0$, $a = 4$) and therefore consumes less memory.

5 Measurements

We implemented the AssyControl case study with UDL and transformed it to Promela with the algorithm described in section 4. The requirements were specified with several assert statements (e.g. $error \wedge success = false$, see section 3 for details). Furthermore, we added some states to the model which mark a critical system error. Therefore, another requirement was to verify that these states are not reachable. Spin also checked for array out of bounds access and possible deadlocks.

Model	Memory (in MB)	Runtime (in seconds)	Amount of states	Success
1	2999	39.2	29141706	no
2	2223	33.3	20663354	yes
3	674	12.8	6798196	yes

Table 1: Measurements: Runtime, memory consumption and state space size of the four Promela models

The emphasis of our measurements was to check how our approach could decrease the run-time of the model checker and its memory consumption. We created three different Promela models to check how the different optimisations of the last sections could improve the model checkers performance:

1. A transformation from UDL to Promela without any optimisation.
2. Like 1, but actions were encapsulated with *d_step* blocks.
3. Like 2, but three auxiliary variables were reset to their initial values when leaving the corresponding state

The Promela models were transformed with Spin to a verifier and executed on a *64-Bit Ubuntu Linux* with 4GB of RAM and an *Intel Core2 Duo E8500* CPU. The results of our measurements can be seen in table 1. Spin was allowed to use 3GB of RAM to map the state space into memory. The column *success* indicates whether a complete state space traversal was possible. The model checker in measurement 1 could not verify the complete state space and terminated after 40 seconds because the available amount of memory was exhausted. The advanced statement merging technique in measurement 2 reduced the total amount of states and the verifier needed 33 seconds to check the complete state space. The reinitialisation of auxiliary variables in measurement 2 reduced the state space and run-time by 60%.

6 Related Work

Formal verification of UML statecharts was already subject of many research projects. Latella et al. [LMM99] provide a complete set of semantic rules for UML statecharts and present a transformation algorithm [LM+99] to Promela. Holzman et al. [MLSH98] developed the tool *MOCES* (Model Checking Statecharts) for the verification of state machines for *Statemate* (a tool developed by *I-Logix*). Paltor et al. [Pal99] discuss a complete formalisation of UML state machine semantics and inherit from it the tool *vUML* to verify statechart diagrams.
Merz et al. [Mer02] develop the tool *Hugo/RT* which tranforms XMI (XML Metadata Interchange) [OMGb] or the proprietary domain specific language UTE (Textual UML

Format) into various target languages like Java or Promela. UTE supports classes, the assignment of a finite state machine to a class and the specification of requirements.
The *Rhapsody UML Verification Environment* [STMW04] is a similar approach, which is embedded into the *I-Logix Rhapsody* tool (now *IBM Rhapsody* [rha]). It allows the verification of UML models with a transformation from XMI into the VIS (Verification Interacting with Synthesis) [BH$^+$96] model checker input language. The behaviour of the UML models is described with statecharts, the requirements are described with LSCs (Life Sequence Charts) [DH01].
No approach in the related work extends UML statecharts with additional features to optimize the verification process like the abstraction of elements for which a concrete implementation is not important and decreases the models amount of states.
Many approaches in the related work use XMI for a transformation into a model checker input language. XMI does not provide an platform independent action language and therefore the behaviour is described with the target platform syntax (e.g. Java or C++). This makes a formal verification of XMI difficult, because not every element of a high level programming language can be easily transformed into a model checker input language (e.g. the allocation of memory).
[Mer02] and [STMW04] provide an action language which allows the specification of active classes. The missing possibility to create non-active classes leads to the following problem: Abstract data types like a stack have to be implemented as an active class, whose objects receives events (e.g. for *push* and *pop* operations). Especially if several stack objects are used in a software project, the corresponding interleaving of active objects, their message queues, etc. increases the size of the state space and has a negative impact on the verifiers memory consumption.

7 Conclusion and Further Work

This paper presents a way to achieve verified applications from business models. We model parts of AssyControl as business process with activity diagrams, transform it into a statechart and enrich it with more implementation details. The complexity of the statechart makes a verification with a model checker difficult because the complete state space consumes the available amount of memory.
We therefore present the domain specific statechart modelling language UDL and a transformation from UDL into the model checker input language Promela. UDL contains language constructs for property preserving abstraction to reduce the models state space. The transformation algorithm provides some optimisations which reduce the state space and the verifiers run-time. We model an expansion for the *AssyControl* tracking system with UDL and verify with three measurements that our approach could significantly reduce the run-time of the model checker.
We implemented UDL with Xtext (including lexer, parser, an editor with syntax highlighting and validation). We are working on the translation templates *udl2pml* and *udl2java* for an automated transformation and integration of UDL into software projects. Errors can occur when an existing UML statechart diagram is transformed manually into UDL. Its furthermore important to develop a tool which automatically converts the XMI output of UML tools into UDL.

Statecharts are just a single example for behavioral modeling. We are working on a generic framework which allows the integration of UDLs action language, optimisation- and abstraction features in other Xtext and Xpand projects. The benefit is an easy way to implement and verify behavioral domain specific languages.
UDL allows only textual modelling of UML statecharts. Visualisation of large models helps the developer to handle the growing complexity of models. The Eclipse IDE supports frameworks like *Graphiti* [gra] and *GMP* [gmp] for the generation of graphical editors. One goal for further work is the development of a framework which automatically generates a graphical editor for a given domain specific language.

References

[BH$^+$96] Robert Brayton, Gary Hachtel, et al. VIS : A System for Verification and Synthesis. pages 428–432. Springer-Verlag, 1996.

[bou] BOUML. http://bouml.free.fr; accessed 14-September-2010.

[CG$^+$01] Edmund M. Clarke, Orna Grumberg, et al. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19:45–80, July 2001.

[gmp] Eclipse GMP - Graphical Modeling Project. http://www.eclipse.org/modeling/gmp; accessed 31-January-2011;.

[gra] Eclipse Graphiti - A graphical Tool development Framework. http://www.eclipse.org/graphiti; accessed 31-January-2011;.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[Hol03] Gerard Holzmann. *The Spin Model Checker: Primer and reference Manual*. Addison-Wesley Professional, 2003.

[Kle09] Stephan Kleuker. *Formale Modelle der Softwareentwicklung*. Vieweg+Teubner Verlag, 2009. in german.

[kov] KoverJa Projekt - Korrekte verteilte Java Applikationen. http://www.edvsz.fh-osnabrueck.de/kleuker/CSI/KoverJa; accessed 14-September-2010; in german.

[LM$^+$99] Diego Latella, Istvan Majzik, et al. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, December 1999.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 465–482, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[Mer02] Stephan Merz. Model checking and code generation for UML state machines and collaborations. In *In G. Schellhorn and W. Reif. 5 th Workshop on Tools for System Design and Verification (FM-TOOLS*, pages 59–64, 2002.

[MLSH98] E. Mikk, Y. Lakhnech, M. Siegel, and G.J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 90 –101, 1998.

[omga] Object Managament Group. http://www.omg.org; accessed 21-April-2011;.

[OMGb] Object Management Group. MOF 2.0/XMI Mapping, Version 2.1.1. http://www.omg.org/spec/XMI/2.1.1; accessed 14-September-2010.

[OMG10] Object Management Group. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Technical Report formal/2010-05-03, 2010.

[OvdADtH06] Chun Ouyang, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. From Business Process Models to Process-oriented Software Systems: The BPMN to BPEL Way. October 2006.

[Pal99] Ivan P Paltor. The Semantics of UML State Machines. Technical report, 1999.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[rha] IBM Rational Rhapsody. http://www.ibm.com/software/rational; accessed 14-September-2010.

[Rui01] Theodorus Cornelis Ruijs. *Towards Effective Model Checking*. PhD thesis, Enschede, March 2001.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39:25–31, 2006.

[SH99] Protocol Case Study and Gerard J. Holzmann. The Engineering of a Model Checker: the Gnu. In *In Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680*, pages 232–244. Springer Verlag, 1999.

[sof] soft2tec GmbH. http://soft2tec.com; accessed 14-September-2010.

[STMW04] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *Proc. SEFM 2004*, pages 174–183. IEEE, 2004.

[Was03] Andrzej Wasowski. On efficient program synthesis from statecharts. *SIGPLAN Not.*, 38:163–170, June 2003.

[Was04] Andrzej Wasowski. Flattening statecharts without explosions. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 257–266, New York, NY, USA, 2004. ACM.

[xte] Eclipse Xtext - Language Development Framework. http://www.eclipse.org/Xtext; accessed 31-January-2011;.