# Real Time Constraints in System Level Specifications Improving the Verification Flow of Complex Systems

Alexander Pacholik, Wolfgang Fengler, Horst Salzwedel

Ilmenau Technical University, Germany,
Faculty of Informatics and Automation,
Department Computer Architecture,
P.O. Box 100565, D-98684 Ilmenau
{wolfgang.fengler, horst.salzwedel, alexander.pacholik}@tu-ilmenau.de

Oleg Vinogradov

Moscow Power Engineering Institute (Technical University)
Krasnokazarmennaya, 14, Moscow 111250 Russia

**Abstract.** Complex real time systems like large system on chips need to be verified to assure quality and save time. Today verification activities are restricted to the register transfer level or one design step above. A complete flow from early specifications down to physical implementation is still not available. An improved system level design cycle is required to overcome this limitation. In this paper we propose an extended methodology of the system level design cycle to support early validation and formal verification of temporal properties in system specifications.

## 1. Introduction

Today automatic formal verification is restricted to formal property and requirement descriptions. It is difficult to apply automatic verification methods to more abstract descriptions like system level specifications. We recommend using at least semiformal specifications (UML) for early specification phases, while executable architecture and performance models should be used at the next stage. Here executable models allow validation and a more exact specification on component requirements.
In [1] it was shown, that mission and system level design techniques permit to model and simulate complex systems at performance level and speed up the design process. Design sizing errors and errors stemming from couplings between subsystems were significantly reduced. It was shown that many design decisions can be carried out at this level of abstraction that previously had been carried out at the functional level. The question is can formal verification techniques from functional level design also be applied at system level design.

Abstract specifications are by nature semiformal. Parts of them, e.g. an UML model structure, can be automatically formally interpreted [2], while others, e.g.

vague function descriptions “encode” cannot. Exact functional descriptions are not needed, but specifications on performance and time properties. As shown in [3], behavioral models with added time properties can be checked against structural and temporal properties by using existing verification tools.

In this paper we extend this approach. First we describe a class of models where our approach is usable. Second we depict a verification flow and show the need for a performance constraint language. Third we analyze requirements for the performance constraint language. At last we introduce a simple constraint language and demonstrate the verification methodology for an example.

## 2. Model classes for verification

For early system models only semiformal model descriptions can be used. They must be able to express the main temporal behavior, while details of some components cannot be described exactly, since for these components no internal behavior is known or defined. Thus, to analyze the entire model, estimates, e.g., specifications of standards, have to be used. Incomplete descriptions lead to models which cannot be analyzed [3]. As shown in [2] and [3] interval based time notations are suitable for such early descriptions. UML does not support time annotations. A special constraint language has to be used. Models can be hierarchical and have to handle discrete events only, in order to formalize them by a transformation into a formal analyzable transition system, like real time automata or interval Petri nets [6].

### 2.1 UML

Known classes of such models are behavioral descriptions of the UML, like activity diagrams, state charts and sequence diagrams. However, there is a problem with UML descriptions regarding executable models. While simple models are easy to handle and to understand, for complex systems there exist no rules how to arrange different diagram types into an entire model. In connection with the UML model driven architecture (MDA) and model driven design (MDD) other reasons can be identified:

- Incomplete models: The models are incomplete in some way or important behavioral parts are still missing. Maybe only interesting behavior has been modeled very detailed, while the glue between these parts (often architectural information) is absent.
- Inadequate model view: Often UML is used to generate the implementation or an implementation stub directly. Then the needs for a simulation environment may be ignored and the model isn’t executable.
- Lack on simulation support: Some additional model elements are needed for simulation [4]. These parts are often called instrumentation and just needed for a correct model interpretation. Such parts are depending on the simulation environment and thus they are not directly part of the UML.

The UML MDA methodology will only be widely used, when the effort to produce models gives a real benefit, like early system simulation.

### 2.2. MLDesigner

For our investigation the tool MLDesigner was used. MLDesigner has extended modeling and simulation capabilities of Ptolemy [5] to generate executable models to test architecture, function and performance of systems of different models of execution. Different models of execution models are connected semantically into one model. MDA and MDD approaches are used from specification development to implementation, e.g. UML activity diagrams are replaced by executable mission level models [6].

For this investigation, MLDesigner event based models (the main domains state chart, discrete event, synchronous data flow) and hierarchical model structures are being used. In these areas, MLDesigner models and UML models support similar concepts. Different is however the MLDesigner built in simulation concept. Primitive Model elements are designed for reuse and consist of an interface based abstract description and an implementation for simulative execution. Complex model elements describe only the structural connection of other (simple or complex) model elements. To analyze MLDesigner models formally, they have to be transformed according to their domain semantics [7] in to formal transition systems. Furthermore temporal timing notations need to be added.

### 2.3. Common Model

Both model classes support the main concepts of hierarchical modeling and event based behavior description. Also both model classes don't adequately support temporal timing annotations. Differences exists in the exact semantically description of MLDesigner models and the connection of different model types (domains) as well as the implicit simulation support. UML models are more abstract by the cost of exact execution and connection semantics. However, when UML MDD / MDA will work in a kind of (early) model simulation, these points will be resolved [6]. Hence, our explorations will be based on common concepts of hierarchy and event based modeling.

## 3. Verification

As mentioned above, semiformal models have formal and informal parts. For temporal verification formal temporal models are needed. Hence a language describing temporal properties shall be used. To generate the formal temporal model we recommend the use transformation rules.

Verification works as follows. Mainly there are two different types of verification, equivalence checking and property checking. Equivalence checker prove on equivalence between implementation (IMP) and specification (SPEC), whereas property checker prove whether the implementation implies the specified properties [8].

- Equivalence: IMP $\leftrightarrow$ SPEC
- Implication: IMP $\rightarrow$ SPEC

Since early specifications are incomplete by nature, property specification (implication) is a suitable mechanism for our needs. An addition to implementation, which in our case is the model description, a property specification is needed. Here other models or also the temporal description Language can be used. Because our original models don't support constructs that differ between implementation and specification, the temporal description language shall manage this. For a hierarchical model structure, upper level models specify properties, which have to be realized by the more detailed lower level implementation. This is known as Hierarchical verification and ensures consistency in a model.

Using the words *specification* and *implementation*, while talking about early models and early specification in connection with simulation and modeling, has limitations, particularly if specification and/or implementations are incomplete. Performance values/specifications have to include their levels of trust, in order that the designer can determine, by which probability the property will be fulfilled. For designs that adhere to a standard, this level of confidence may be very high for other designs much lower.

## 3.1. Verification Flow

The verification flow, depicted in Figure 1, starts with a semiformal description completed with temporal annotations. This description is divided into specification (properties) and implementation parts (system model) and then transformed into formal models. After generating formal temporal models for the implementation and specification parts, a conventional model checker can be used to prove the implied relation between them. It can be decided, whether a specified property holds and under which assumptions. Otherwise, dependent on the used verification tool and the retransformation mechanism, a counter example can be generated.

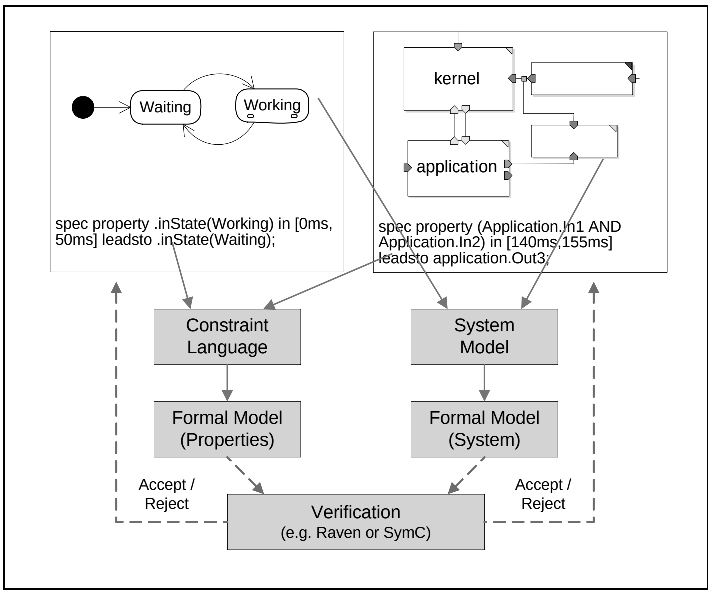

Waiting
Working
spec property .inState(Working) in [0ms, 50ms] leadsto .inState(Waiting);
kernel
application
spec property (Application.In1 AND Application.In2) in [140ms,155ms] leadsto application.Out3;
Constraint Language
System Model
Formal Model (Properties)
Formal Model (System)
Accept / Reject
Accept / Reject
Verification (e.g. Raven or SymC)

**Figure 1: Verification flow for semiformal models.**

While the verification task (dashed arrows) is well understood and available in verification tools [11], formalizing a semiformal model as well as model transformation is difficult [12]. Figure 2 depicts the relations between the verification task and a constraint language used to formalize models.

## 4. Requirements for a performance constraint language

The general approach is similar to the SymC approach, presented in [9], however our intent is not to create new verification tools but to make them available to other (semiformal) models. Some questions concerning the temporal property language are open so far:

- Expressiveness: Temporal logics and properties have different expressivenesses. Mainly three classes are considered: CTL, LTL and CTL*.[1]
- Connection to the semiformal model. The semiformal models have different semantic and syntactic constructs. How to use one Language for several different models?
- Which language: Can a known property language be used, or has a new language to be defined?

Concerning the last question, for semiformal models no well defined temporal specification language exists. The OCL, connected to UML models can describe some properties. Some temporal extensions to the OCL have been defined, but they didn’t become a standard. For formal models, particularly for hardware verification, several languages have been defined. The most suitable temporal specification language is the property specification language (PSL). [10] This Language can express CTL and LTL constraints and properties and is able to handle normal, composed as well as series expressions, and sequences of events. The PSL is an assertion based Language and exist in several dialects for VHDL, Verilog etc. so it is universally usable in the RTL description level. Additionally the PSL has directives that tell a verification tool how to handle a particular PSL expression.

PSL is very near to a language fulfilling our requirements. However, PSL is made for an environment, where a global clock exists. That is different in our abstract descriptions, where times have to be expressed like a physical description (with number and time unit), and, if modeled by a clock, events would occur in long clock distances. Furthermore the semantic difference between cycle-based (PSL) and event-based (Our descriptions) may cause problems [11]. Additionally there is the need to describe assumed temporal properties about system parts. This is different to the assumed mechanism of PSL, where only environmental properties, e.g. input behavior, are restricted as part of the verification task instead of integrating them to the system model. So we prefer to define our own performance description language, where we keep the language structure, especially the four layers, boolean layer, temporal layer, modeling layer and verification layer, and use as much of the directives of PSL as possible. Why it is necessary to define several layers? Generally a property can be

[1] A very good introduction can be found in [7].

seen as a composition of the three layers, which makes it easier to compose and analyze them systematically [11]:

- The Boolean layer describes Boolean expressions (model specific events) and their simple compositions.
- The temporal layer describes the relationships between Boolean expressions over time.
- The verification layer describes how to use a property during verification.

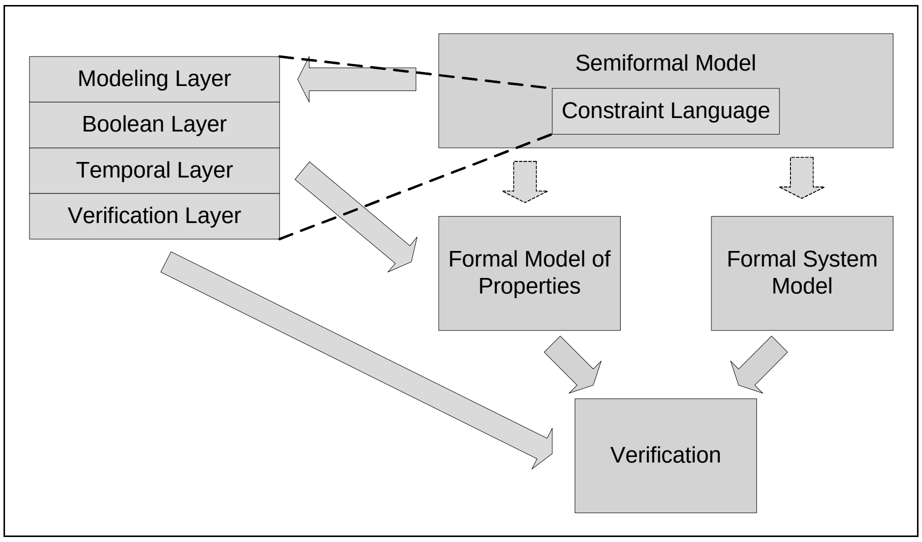


**Figure 2: Constraint language in a verification flow.**

As depicted in Figure 2, an additional modeling layer is required. It is responsible for connecting several different models and description techniques, without affecting other language layers.

## 5. A simple constraint language definition

First we will give an introduction into the theoretical background [7] of event based specification. Then we outline restrictions of this approach and possible solutions.

In event based systems, every event is connected to a point in time. Because one event can occur infinitely often, only the last event (with its occurrence time) is known, when an expression is evaluated. An event is called to be true at its last occurrence.

To build combinatorial assertions we define some rules, whereas $x$ is an event, $X$ is the set of events and $\alpha(X)$ is an assertion over the set of events:

- $x \in X \Rightarrow x \in \alpha(X)$: When $x$ is an event, an assertion of $x$ exists.
- $x \in \alpha(X) \Rightarrow (x) \in \alpha(X)$: Every composed assertion can be treated like an elemental assertion.

- $x, y \in \alpha(X) \Rightarrow x \wedge y \in \alpha(X)$: An assertion can be a conjunction of other assertions.
- $x, y \in \alpha(X) \Rightarrow x \vee y \in \alpha(X)$: An assertion can be a disjunction of other assertions.

Because every assertion is connected to a point in time, rules exist to connect a combined assertion to time:

- $x \in X \Rightarrow t(x)$ is the occurrence time of the event $x$. The assertion $x$ is true at time when the event $x$ occurred.
- $\varphi \in \alpha(X) \Rightarrow t(\varphi)$ is the time when formula $\varphi$ became true the first time.
- $\varphi, \psi \in \alpha(X) \Rightarrow t(\varphi \wedge \psi)$ is the time, when both, $\varphi$ and $\psi$, became true the first time.
- $\varphi, \psi \in \alpha(X) \Rightarrow t(\varphi \vee \psi)$ is the time, when at least, $\varphi$ or $\psi$, became true the first time.

Using the negation is not allowed, because an exact semantic can't be given.[2] However, to negate a property, the temporal expression could be negated.

A primitive temporal specification, using the temporal operators $A$ (always), $[]$ (all paths) and $<>$ (some paths), can be written as follows:

- $A[](\varphi \Rightarrow A <> [t_1, t_2]\psi)$,

in which the sets of events used in $\varphi$ and $\psi$ have to be disjunct. This means, $\varphi$ always causes $\psi$ inside the given time interval. Then, because of

- $(\varphi_1 \vee \varphi_2 \Rightarrow \psi_1 \wedge \psi_2) \equiv (\varphi_1 \Rightarrow \psi_1) \wedge (\varphi_1 \Rightarrow \psi_2) \wedge (\varphi_2 \Rightarrow \psi_1) \wedge (\varphi_2 \Rightarrow \psi_2)$

Complex specifications can be split and thus simplified to

- $A[](\bigwedge x \Rightarrow A <> [t_1, t_2] \bigvee y)$,

whereas $x$ and $y$ are primitive events.

## 5.1 Concrete syntax

For a demonstration we allow only simple implications and rate constraints over successive events of one type. We add the keywords “imp” and “spec” to differ between implementation assumptions and specified properties. Additional elements concerning the verification and temporal layer are omitted in the easy EBNF syntax as follows:

```
CL_Specification ::=
     CL_Issue "property" [CL_Identifier ":="] CL_Property ";"
CL_Issue ::=
     "imp" | "spec"
```

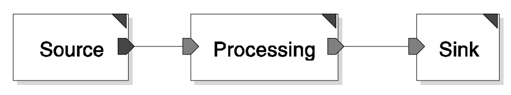
[2] !x becomes infinitely often true until x is becomes true. Because an event x can occur more then once, wouldn't !x have to be true before the last occurrence of x?

```
CL_Property ::=
     "always" CL_ConjEvent "in" CL_TimeInt "leadsto" CL_DisjEvent |
     "every" CL_TimeInt "occurs" CL_DisjEvent
CL_DisjEvent::=  {CL_EVENT, "or", 1}
CL_TimeInt ::= "[" CL_TimePoint "," CL_TimePointInf "]"
CL_ConjEvent ::= {CL_EVENT, "and", 1}
CL_TimePointInf ::= <numeric> | "inf"
CL_TimePoint ::= <numeric>
CL_Identifier ::= <identifier>
CL_Event ::= <identifier>
```

Events and identifiers are alphanumeric strings taken from the concrete constraint description. The property "every [t1, t2] occurs event" means, an event will occur every t1 to t2 time units, whereas in the case of a specification the first occurrence of the event is not included.

## 5.2 Model Transformation

For a given model structure we can use the constraint language to generate real time automata for the implementation (system model) and for the specification (requirements). With a short example we start with a MLDesigner model an transform it manually into a real time automata system using the UPPAAL tool. [13]



```
imp property every [15,20] occurs Source.output1 ;
imp property Processing.input1 in [12,14] leadsto Processing.output1;
spec property fw_delay := Source.output1 in [10,14] leadsto
Sink.input1 ;
spec property sink_rc := every [13,23] occurs Processing.output1 ;
```

**Figure 3: Simple model with timing specifications**

In Figure 3 a simple abstract semiformal model is given. Additional constraints are used to describe the temporal behavior as well as two specification properties.

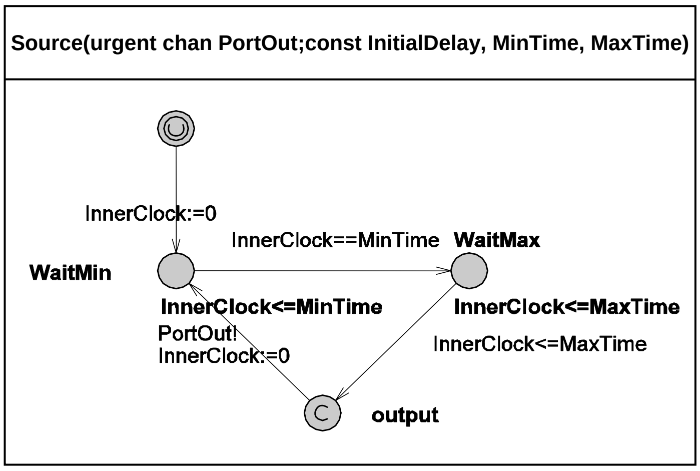


**Figure 4: Automaton for the source block property**

In Figure 4 you can see the automaton, which produces events on the output of Source in every MinTime to MaxTime time units.

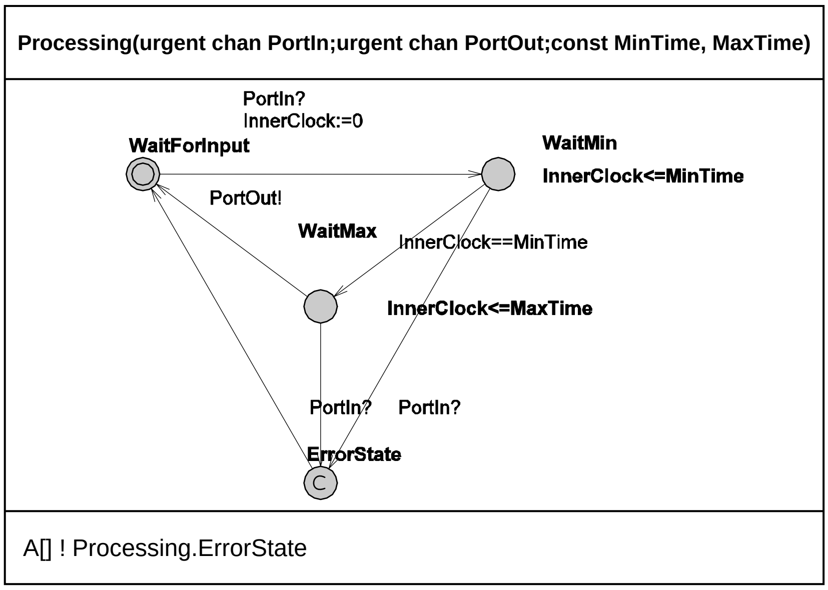
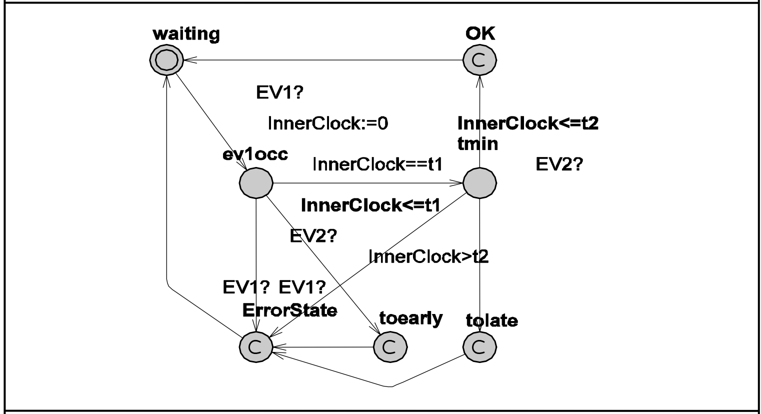


**Figure 5: Automaton for the processing block**

The automaton in Figure 5 realizes the specified implication property of the processing block. It produces one output event in MinTime to MaxTime time units after an input event occurred. Because a single automat can only handle one event each cycle, an error state and a temporal condition (A[] ! Processing.ErrorState) had to be added.

**Fw_delay(urgent chan EV1, EV2;const t1,t2)**

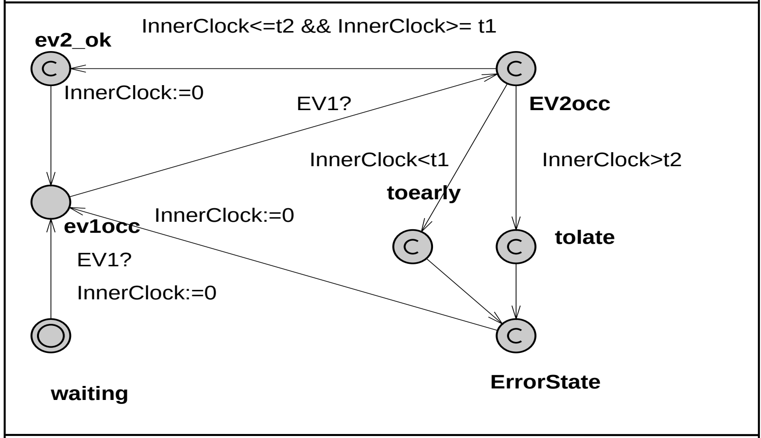


A[] ! Fw_delay.ErrorState

**Figure 6: Automaton for fw_delay property**

To process the two specified properties of the system for each an additional automaton was created. You can see them with their temporal property in Figure 6 and Figure 7. In UPPAAL the automata are connected trough their parameters during instantiation according to the model structure in Figure 3.

**Sink_rc(urgent chan EV1;const t1,t2)**



A[] ! sink_rc.ErrorState

**Figure 7: Automaton for the sink_rc property**

After generating the automata, a final life property (A[] ! deadlock) must be added, then the automata system can be checked using the verifier. In our case the all specified properties of the simple model could be verified.

### 5.3 Limitations

This example is simple, but it shows the general verification flow, described in chapter 3. Although the presented constraint language doesn’t meet the requirements from chapter 4, some problems and limitations can be found, as well as additional properties, which are needed in an advanced constraint language:

- Much more constructs are needed, e.g. to express branching or case decision behavior to events.
- Counting events: Events must be countable to describe complex behaviors.
- Causality limits expressiveness: To produce implementation behavior using automata only one event can be handled per property and time. Modeling abstract pipelined processes isn’t possible so far, because consecutive event chains could overtake each other
- Arranging conjunctive dependencies: When two or more events cause another event conjunctively either all combinations have to be caught or counting places have to be used. A problem here is then the semantic. What do to with additional events, eventually they are needed to enable an event next time?
- A mechanism catching (time spaced) event chains similar to sugar regular expressions (SERE) in PSL has to be found. This is necessary to improve expressiveness and benefit by using such a constraint language.
- For some modeling domains the event semantic has to be changed. There special channels (which e.g. implement FIFOs) could be used to connect model elements according to the model structure.

## 6. Conclusion and Future Work

We outlined the need to verify system models early in the design, when models are abstract and incomplete. A methodology to formalize and verify such semiformal models by adding constraints was presented. Constraints are the heart of this methodology, so a well structured constraint language is needed. A language concept analogous the concept of PSL was given. In chapter 5 we demonstrate the methodology for an example case. Some limitations and open question are shown.

In the future we are going to create an advanced language definition, which recognizes the concept and other requirements developed before. Furthermore we will create a framework or tool, which automates the verification task. The verification flow is made for static verification, but is also possible to perform dynamic verification by checking the specified properties during simulation or execution.

## 7. Acknowledgements

This work is supported by *Bundesministerium für Bildung und Forschung* (BMBF) and *edacentrum e.V.* under programme FEST (FK 01M3072A).
MLDesigner © 2004 MLDesign Technologies, Inc. All rights reserved. *http://www.mldesigner.com/*

## References

1. G. Schorcht, P. Unger, A. George, I. Troxel, D. Zinn, H. Salzwedel, K. Farhangian, C.K. Mick: "*System-Level Simulation Modeling with MLDesigner*", IEEE / ACM MASCOT 2003 - 11th ACM / IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 12.-15. Oktober 2003, Orlando, Florida, USA.
2. Olga Fengler: „*Extension and formal verification of dynamic object oriented modeling approaches based on higher Petri nets*“, (in German language) PhD Thesis, TU Ilmenau 2004.
3. Thomas Licht: „*A timing analysis method for UML models in the design of automation systems*“, (in German language), PhD Thesis, TU Ilmenau, ISLE Verlag 2004.
4. Bernd Däne, Wolfgang Fengler: "*A Case Study for Partitioned Modelling of a Control System*", in M. H. Hamza (Ed.): "*Modelling, Identification and Control*", ACTA Press 2005.
5. J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt: „*Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*" in "*Int. Journal of Computer Simulation*", 22(4), 1994.
6. Tommy Baumann, Horst Salzwedel: "*Mission Level Design Using UML 2.0"*, Object Days, Erfurt 2005.
7. Thorsten Hummel, Wolfgang Fengler: "*Design of Embedded Control Systems Using Hybrid Petri Nets and Time Interval Petri Nets*", in: M. A. Adamski, A. Karatkevich, M. Wegrzyn (Eds.): "*Design of Embedded Control Systems*",. ISBN 0-387-23630-9, Springer Berlin Heidelberg New York 2005, pp. 141-154.
8. Thomas Kropf; „*Intoduction to Formal Hardware Verifivation*“, Springer Verlag 1999.
9. Prakash M. Pernandam, Roland J. Weiss, Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel: „*Dynamic Guiding of Bounded Property Checking*“, in IEEE International High Level Design Validation and Test Workshop 2004 (HLDVT 04), 2004.
10. "*Property Specification Language Reference Manual*", www.eda.org/vfv /docs/PSL-v1.1.pdf
11. Harry D.Foster, Adam C. Krolnik, David J. Lacey: "*Assertion-Based Design*", Kluwer Academic Publishers 2003.
12. Manuela Rapp: „*Analysis and comparison of state based and event based description techniques*", (in German language) Diploma Thesis, TU Ilmenau 2005. INV: 2005-06-01/055/IN99/2231
13. Gerd Behrmann, Alexandre David, and Kim G. Larsen: "A Tutorial on Uppaal", in "*proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*" (SFM-RT'04). LNCS 3185.