Verification of xUML Specifications in the context of MDA

Günter Graw[1], Peter Herrmann[2]
[1]ARGE IS KV
[2]University of Dortmund
Email: grawg@gmx.de, herrmann@ls4.cs.uni-dortmund.de

Abstract. This paper deals with the application of verification techniques in the context of MDA. The main emphasis is on the UML profile xUML and on TLA based verification and specification methods.

Introduction

The MDA (Model Driven Architecture) is the most recent initiative of the OMG to facilitate the creation of object-oriented software. This approach has the goal to specify software for different independent domains using abstract high level models. These high level models are specified using the UML (Unified Modelling Language) as specification language, which is another standard adopted by the OMG. The UML models are used as input for the generation of code. MDA distinguishes two different kinds of models: platform specific models (PSM) and platform independent models (PIM). In the last year UML semantics was extended by an action specification language [AS01], which had the aim to enrich the action semantics of the UML. Actions of this language are declarative by nature. This result has been used to create the xUML (executable UML) profile [MB02] which supports the execution of UML models. The semantics of UML is restricted in a certain way in order to make UML models executable. Meanwhile several companies have created tools (e.g., bridgepoint, iCCG) that support the execution of xUML models.
Since several criticism to the semantical precision of the UML exists, many researchers have proposed the combination of formal methods and UML models which aimed at the formalisation of UML specifications and the verification of properties of an UML specification. Some of these formal methods are also action based, in particular those that are specification styles and extensions of TLA (Temporal Logic of Actions) which was originally designed by L. Lamport [L94] and offers a high potential for the combination with xUML based action languages. Lamports method has been applied successfully so far in hardware [YML99], controller [HK00b] and protocol [HK00a] design. An approach like MDA that is based on the generation of code from abstract models requires a high degree of formal correctness to be economical manageable. This position paper discusses the reasons why verification is necessary for xUML specifications and the fields of application of formal methods in the context of MDA based on xUML.

Executable UML

Former versions of UML suffered from the decisive disadvantage that they were not executable due to semantical incompleteness or ambiguities which have been discovered by the members of the precise UML group [pUML]. The reason for this was founded in an extremely limited set of actions which are *send*, *call*, *return*, *create*, *terminate*, *destroy*, *uninterpreted* and *local invocation* actions. This led to an extension of UML in late 2001 by an action semantics. This actions semantics offers a complete set of actions at a high level of abstraction resulting in the creation of a profile for the execution of UML specifications. Thus

xUML is a single language in the family of UML languages which semantics is described in a formal manner using a set of rules describing how particular things in UML fit together to form a profile that supports the execution of UML models [MB02]. A couple of UML's former model elements [UML] haven't changed so drastically in this profile and are still part of the xUML. These are:

- Use Case diagrams which are used for the description of requirements in the early phase of requirements engineering. The model elements of this kind of diagram are actors, use cases and their according relations.
- Sequence diagrams which might be used to specify the interaction of use case objects and the actors related to use cases using communicates relationships.
- Class diagrams describing the static structure of xUML objects, their attributes, and operations as well as their relationships. Because the focus is on dynamic aspects, it is no more necessary to distinguish aggregation and composition in associations. Now there are additional object and attribute actions as well as link actions. These specify the creation of object instances, the assignment of values to object attributes as well as the creation of links between object instances which have been specified as associations in the class diagram.
- Statechart diagrams specify behavioural aspects object instances belonging to the same class. These consist still of pseudostates, states and transitions (based on Event[Guard Condition]/Action semantics), specifying the behaviour and lifecycle of object instances, but the kind of actions that can be used as entry or exit actions of states as well as the kinds of actions that might be used in transitions to specify the dynamics of state changes are drastically enriched by the action semantics and are explained in [AS01].
- Collaboration diagrams are used to model the interaction of objects using messages.
- Moreover OCL constraints which have the character of rules can be written in an action language and thus can restrict the value of attributes and associations in an xUML model. There are translations defined between OCL constraints and action languages [MB02].

Component and deployment diagrams whose main emphasis is on the description of structural and static model elements in late phases of the software development process don't have any influence in xUML and aren't subject to deeper consideration, because they are in strong contact to the implementation platform.

Currently there exist several action languages of different vendors which have their roots in the actions semantics (e.g., the ones described in [MB02] and [KC02]), but the standardisation process, however is already in progress.

Executable UML in the context of MDA

In the industrial practice as software architect one of our authors daily sees that the first phases of current object-oriented software processes especially Object Oriented Analysis (OOA) suffer from several problems. In several international projects in which software for the domain of health insurance is developed, UML 1.3 is used. OOA is performed identifying classes and their relationships from use cases and their descriptions. Statechart diagrams are seldom used to specify the behaviour of very important classes. Also interaction diagrams have no big importance in many projects during this phase. Although this approach correlates with the usage of an instant UML core this fact leads to the following shortcomings concerning the artefacts produced in the phase of OOA:

- Only analysis classes and their relationships are taken into account and in most of the cases their behaviour is neglected. It would not be fair to blame the business analysts for the shortcomings of their work because some months ago there was no chance to execute an UML model from this phase of the software process at all.

- Behavioural analysis in OOA artefacts is incomplete, inconsistent or is not considered at all. The designer in Object Oriented Design (OOD) is lucky if the analysis model is provided with some business rules.
- Due to this absence of formality in OOA models, the designer has to rethink analysis decisions. The tracking of decisions which lead to an OOA model are difficult to trace back, because the designer sees only the class diagram and the use case descriptions based on textual templates. There's a lot of communication overhead between business analysts and designers
- The review of OOA artefacts performed by members of a quality assurance team is difficult and sometimes a nightmare because of its ineffectiveness. Thus there is no effective way to restrict or at least reduce the propagation of analysis errors and inconsistencies into the subsequent phases of OOD and Object Oriented Programming (OOP).
- The boundaries between OOA and OOD [SPHGJ01] are not clearly defined. This leads to frequent changes in OOD. It's not clear if a business object can be taken over from analysis to design without changes or if it has to be refined with other objects, transformed into other design objects using splits or compositions.

All these reasons lead to frequent iterations between several people working in different phases resulting in high economic efforts.
To understand and estimate the consequences of these problems it is necessary for the reader to know that OOA and OOD are often performed by business analysts and designers of different companies which means that artefacts are exchanged between different companies crossing the boundaries of different companies. Moreover a typical project for health insurance has at minimum 800 business objects. This combination of problems and risks has resulted in difficulties in particular for big projects.
It is the aim of OMG MDA initiative to overcome these deficiencies by the separation of an architecture for software systems into platform independent models (PIM) and platform specific models (PSM). In the PIM all analysis relevant and a couple of design relevant models are concentrated [OOA]. Moreover the early consideration of aspects of system integration is proposed. The PSM deals with the realisation of the PIM on a particular computing platform.
Once the PIM models are specified in xUML, there are stronger completion criteria which are fulfilled if a PIM model executes correctly. Furthermore these completion criteria enable the effective reviews for members of a quality assurance team which leads to an increased quality of platform independent models. The idea is that validation (which has the character of a simulation) and verification of models start in early phases of the software process. Thus the propagation of errors and inconsistencies to subsequent phases can be restricted. Tools that support the execution of PIM models specified in xUML help business analysts to integrate behavioural aspects in their specification and give them an impression of the correctness of their models.
Furthermore it is recognised that a refinement relation between PIM and PSM exists that is expressed by the application of a mapping. MDA claims that if the PIM and the mapping are defined with high precision the PIM to PSM mapping may be fully automated, which requires the formalisation of properties. Using mappings for code generation of the PIM results in reduced costs, because the PSM and the platform specific implementation (PSI) are not products that underlie maintenance which has its reason in the automatic generation.

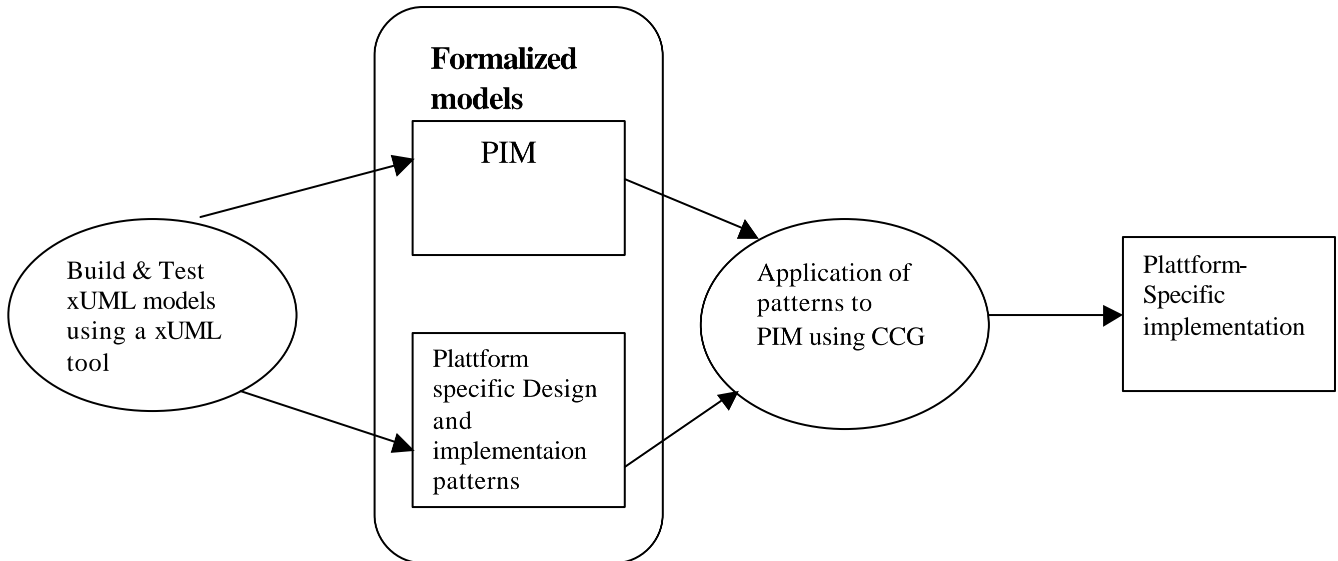

Figure 1: PIM to PSM Mapping

A configurable code generator (CCG in [KC01], [KC02]) or model compiler [MB02] is used to generate the platform specific implementation which gets the PIM platform specific design and implementation patterns as input. The relationships between platform independent models and platform specific design and implementation patterns is depicted in Figure 1. The prerequisite for the generation of a correct platform specific implementation are correct models and correct mappings. Unfortunately, xUML lacks proof rules which are required to perform an effective verification. In the following sections a formal method will be introduced that helps to guarantee the correctness of models and mappings.

The compositional specification style cTLA

The specification style cTLA [HK00a] is based on L. Lamport's Temporal Logic of Actions (TLA) [L94] and supports the definition of parameterised processes and system types. A specification of a simple process or a (sub)system is created by the instantiation of a cTLA process type. As in the formal description language Lotos systems are composed from processes which interact by the means of joint actions.

```
Process SensorObject(maxValue : Integer)
Variables
      Qu : Queue ; (* message queue *)
      Value : Real ; (* Sensor value *)
      State : ("init", "called", "processed", "returned")

INIT ==
      Qu = << >>       /\
      State = "Init"   /\
      Value = 0
ACTIONS
      getValueCall(val : Real) == (*send action*)
            Value < maxValue    /\
            State = "called"    /\
            State' = "returned" /\
            Value = val         /\
            Unchanged qu        /\
            Unchanged Value;
```

```
        Calibrate() == (*computation action*)
              Value > maxValue     /\
              Value' = Value/2     /\
              Unchanged qu         /\
              Unchanged State;
End;
```

Figure 2: cTLA process SensorObject

As an example of a cTLA process we outline the process *SensorObject* in Figure 2 describing the behaviour of an UML object. In the process type header the name *SensorObject* and the process parameters are declared. The state variables *state*, *qu* and *value* model the process state. The set of the initial setting is described by the predicate INIT. State transitions are specified by means of actions. A TLA action (e.g. *getValueCall*) is a predicate about action parameters, state variables describing the state before the execution of the action and so-called primed state variables modelling the state after executing the action. Besides of state transitions specified by actions, a process may perform stuttering steps where it does not change its state whereas the process environment performs a state transition.

The cTLA process *SensorObject* describes safety properties. It is also possible to specify liveness and real-time constraints by the use of additional weak or strong fairness or real-time constraints which are able to force the execution of an action. A fair action has to execute eventually only if otherwise infinitely many states exist where the action is enabled as well as its execution is tolerated by the environment. Systems and subsystems are described as compositions of concurrent processes which encapsulate their state variables and change their local states according to the process actions. The vector of process state variables represent the state of the entire system. System state transitions are described by system actions which are logical conjuncts of process actions and process stuttering steps. Since each process contributes to each system actions by exactly one action or a stuttering step, concurrency is modelled by interleaving and the coupling of processes by joint actions. The action parameters are used to describe data transfer between processes. In Figure 3 the cTLA process *abstractComposition* specifying a controller subsystem is depicted. The subsystem consists of three processes describing a sensor object SO modelling a sensor, an actor object modelling an actor and a task object

```
PROCESS abstractComposition
PROCESSES
SO : SensorObject;
AO : ActorObject;
TO : TaskObject;

ACTIONS

ValueToSensor(value : Real) == SO.getValueCall(value) /\ AO.Stutter /\
TO.enqueue(value);

SensorCalibrate == SO.Calibrate /\ AO.Stutter /\ TO.Stutter;

ActorCalibrate == SO.Stutter /\ AO.Calibrate /\ TO.Stutter;

. . .
END abstractComposition
```

Figure 3: A cTLA process describing a subsystem

modelling a task. The system actions *ValueToSensor*, *SensorCalibrate* and *ActorCalibrate* are defined. The system action *ValueToSensor* models the interchange of a message between a *SensorObject* and a *TaskObject*. It is enabled to be executed if the action *getValueCall* of the process *SO* and the action *enqueue* of the process *TO* are enabled. If it is executed the process *TO* will perform a stuttering step. The system actions *SensorCalibrate* and *ActorCalibrate* cause no interaction with foreign processes.
Moreover cTLA facilitates the combination of different property types like safety and liveness. In the constraint oriented specification style one can specify different aspects of a component by separate constraint processes.
The specification style cTLA has interesting properties concerning the specification of the statecharts of object instances as well as the specification of collaborations which are translated into separate cTLA processes modeling state transition systems. Since cTLA is based on TLA that supports predicate logic as well as temporal logic it is also possible to translate most of the OCL expressions. Furthermore xUML actions are easy to translate in most of the cases.
Like TLA the specification style cTLA supports safety, liveness and refinement proofs by the aid of several proof rules of temporal logic. The refinement of specifications on different levels of abstraction is proved using implication. To perform a refinement proof a refinement mapping has to be constructed. Moreover tools exist that support the translation of a specification which is composed of several cTLA processes into one cTLA process. Proofs can be performed manually or by the aid of the model checker TLC (Temporal Logic Checker) [YML99], [L02] which delivers interesting results up to a restricted complexity of a specification. Because cTLA is compositional it is possible to construct subsystems which contain only the interesting processes of a specification (e.g., an OOA or OOD model).
In former research [GHK99], [GHK00] it was shown that cTLA is a suitable language for the specification and the verification of the behaviour and the properties of the object instances of analysis and design patterns.

Verification support for MDA

In this section interesting points where verification using cTLA is able to support xUML models in the MDA context will be identified.
As stated in [SBK01] the modelling and testing of OOA model using an early version of xUML of the Bridgepoint tool for simulation in xUML was successful and a couple of errors were found, but nevertheless a formal verification using the COSPAN model checker found additional serious errors. Thus a formal verification of the OOA model was helpful in the development of the control software of a robot.
To perform the translation of an xUML model into a cTLA model a xUML/cTLA translator is currently under development, which accepts xUML models with according xUML action specifications as input. The model translator will generate a cTLA specification as output. The output is used for the manual verification of properties or the automatic verification using the model checker TLC [L02] supporting a subset of TLA+ as input language. TLA+ is a formal specification language based on TLA supporting several tools. For some kinds of verification (e.g. real-time properties) the manual verification is less time consuming in cTLA. Figure 4 presents an overview of the xUML/cTLA model translator. The fact that xUML and TLA are action based facilitates the construction of a model translator. Figure 4 includes an additional property editor, which supports the specification of properties for verification purposes (e.g., safety properties: is a certain state reachable at all, or liveness properties: are there traces that certain states are never reached or are there infinite cycles). These formal verifications are much more complete and show inconsistencies with more rigor than validations using a simulator with extensive test cases are able to show.

To transform a specification it is required that the following xUML diagrams are specified:

- One or many interaction diagrams which consist of objects, links connecting the objects and messages interchanged between objects. It is assumed that every object has a unique identifier and the class name is specified.
- A class diagram which consists of classes and associations. For every object of the interaction diagram an according class has to be specified in this diagram defining the names and types of class attributes and the signatures of operations.
- Furthermore a statechart diagram has to be specified for every class of the class diagram which describes the behaviour of a class. In a statechart diagram states and transitions are specified. It is assumed that a transition consists of an event, an action, and a guard. Moreover the transition has a source and a target state. Thus transitions are restricted in a way that they may have only one action. If transitions with sequences of actions are required, intermediate states containing their own transitions have to be introduced. Each transition carries its own action. The syntax of cTLA is used for action specifications preserving as much from the bridgepoint ASL as possible. There are a couple of similarities between both languages. Constants are specified as cTLA constants. Boolean expressions have their counterparts in cTLA Boolean expressions. Arithmetic expressions are translated into arithmetic expressions of cTLA. Conditional constructs like IF-THEN-ELSE and SWITCH are translated into the IF-THEN-ELSE and CASE constructs of cTLA. Loops seem to be problematic, since there is no corresponding construct in cTLA. Loop variables are transformed into process variables and loop conditions are transformed into preconditions of a cTLA action. Assignments of values to variables are handled using primed variables of cTLA.

The model translator transforms each class and its according statechart diagram into a separate cTLA process with local state variables and actions. Class attributes are transformed into the states variables of the cTLA process. An additional variable *state* keeping the name of the current state of the statechart is introduced in every cTLA process of a class. Moreover a queue *qu* modelling the queue of the statechart is added. The statechart diagram is used to generate the initial predicate and the actions of this cTLA process. The transformation of a transition of the statechart diagram is done generating a cTLA action. The precondition of this action consists of the guard condition expressed in cTLA, the variable *state* holding the value of the current state and the event being enqueued as head of the queue *qu*. Furthermore state transitions have to be added transforming the xUML action of the transition have to be generated. Every ASL statement is transformed into a conjunct of a TLA action. Additionally a state transition specifying the transition into the next state of the statechart is added. Furthermore there are actions which are responsible for enqueuing and dequeuing of events into the queue.
The interaction diagram is used for the generation of a cTLA process describing a subsystem using the cTLA processes belonging to classes of objects in the interaction diagram. Each message which is interchanged between objects of the interaction diagram is transformed into a system action conjoining a send or a signal or a return action of an object sending a message with an action which is responsible for the enqueuing and dequeuing of events into the queue of another object receiving the message.
The tool Rational Rose is used to model the UML diagrams. It can be extended using the scripting language Rosescript. Scripts for the specifications of xUML actions and properties on the base of dialogs have been developed. These properties are verified by the model checker TLC or by the aid of manual proofs. Furthermore the xUML/cTLA model translator is developed using Rosescript. During the transformation of a xUML specification the model

translator accesses the repository of Rational Rose to retrieve information about the xUML models.
To generate a flat cTLA specification which is usable as input for TLC, a tool called cTc [HMK96] is applied. This tool gets a cTLA specification consisting of several cTLA processes as input and generates one cTLA process describing the behaviour of all composed cTLA processes.
Finally some minor transformations have to be done to use this cTLA process as input for TLC. Using these tools the tedious work of manual transformations of xUML specifications into cTLA specifications is reduced.

A couple of people (see e.g., [D01]) have recognised that xUML will enforce the development of several kinds of patterns. To reduce the effort, which is necessary to create formalised OOA models it is promising to create formalised libraries of domain dependent prefabricated analysis patterns [F97]. Thus the reusability of formal specifications is increased. In [GHK00] an example for the formalisation of an analysis pattern for real-time software is shown.

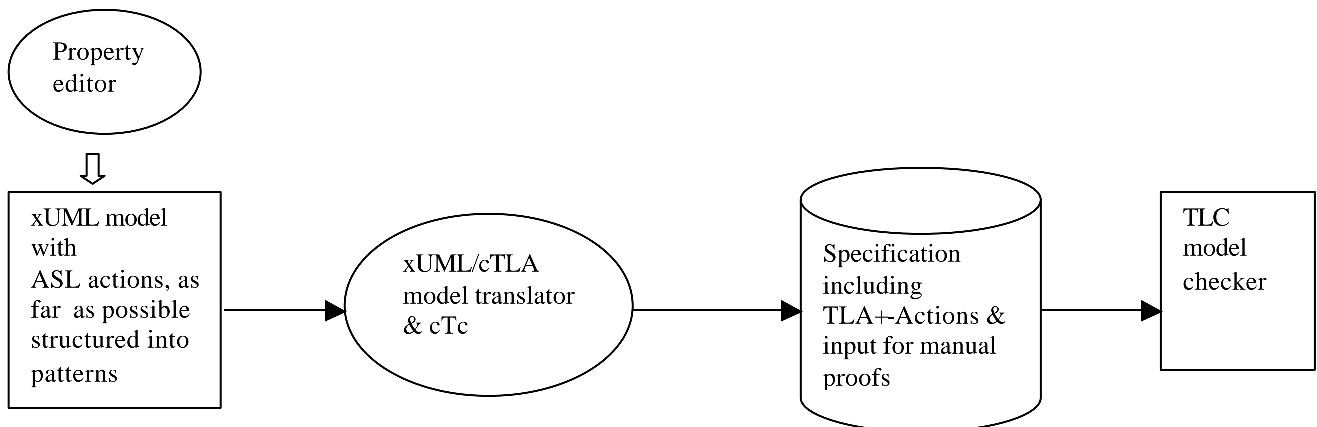

Figure 4: Compilation of xUML models for the purpose of verification

Of course, these ideas can be applied to the formalisation of platform specific OOD patterns as well.
Now the verification of PIM to PSM mapping rules should be considered. In [D01] a overview of architecture and its relationships towards refinement and architectural style is given. Designing a software system is in strong relation with refinement, because refinement describes the relationship between a specification including the functional as well as the non-functional requirements and the design, which is selected to give a solution to the functional and non functional requirements. Moreover it is necessary to know that architectural style defines the specification and design language which has constraining effects on the refinement mapping. A spectrum of architectural styles exist for MDA, but especially two architectural styles are interesting which can be composed to build a model compiler:
1. An architectural style that defines a full translation scheme which means that the architectural style is fully determined and is essentially a compiler that supports an automatic translation from the specification to the realization. In this case the correctness of the refinement is based in the function of the compiler. We do not believe that many of these styles exist because recent software systems are influenced by many non-functional requirements. A listing of these non-functional requirements would include at least real-time, performance, fault-tolerance, security, distribution, scalability. Due to this complexity of non-functional requirements, we do not believe that this approach works for many domains. Most of the model compilers currently available have few possibilities for parameterisation. Nevertheless the correctness of the compiler should be proven using formal methods.

2. An architectural style defining the language for the specification and the design. Of course, the specification should include the functional and non-functional requirements again. Furthermore a couple of well-defined rules exist that help to check whether the realization maps to the specification. A verification tool should support the design process to check the design for correctness. In our opinion this architectural style appears more frequently. We would use the structure of the domain for the verification of the correctness of a refinement mapping. The triple of an analysis pattern, the non-functional requirement and the architectural rules form the basis for the selection of design patterns. The verification of the refinement relationships between the analysis pattern and the selected design patterns leads to the introduction of so called refinement patterns. The correctness of the refinement mapping of each refinement pattern is verified separately with extensive tool support of a model checker. Furthermore domain dependent collections of refinement patterns can be formed accumulating the design experiences for a certain domain. Refinement patterns with formally verified correctness should provide optimal input for a model compiler and still have to be parameterised to generate platform specific code. Currently we have modelled some refinement patterns for the domain of control software.

Conclusion

We presented a method to verify xUML specifications in the context of MDA formally by means of cTLA and model checking. Up to now two xUML specifications describing controllers for distributed control software were transformed into cTLA processes. Invariance and refinement properties were verified for these specifications using the model checker TLC. Although these verifications are restricted by the state explosion problem our results seem to be promising. Future work will concentrate on the development and improvement of model translators and model compilers supporting the verification and the generation of correct code. It is planned to adapt our transformation tool to the upcoming ASL standard.

References

[AS01] Action Semantics for the UML, http://www.kc.com/as_site/home.html, 2001.

[D01] Desmond D'Souza: OMG's MDA An Architecture for Modeling, http://www.omg.org/mda/presentations.htm.

[F97] Martin Fowler: Analysis Patterns, Addison-Wesley, 1994.

[GHK99] Günter Graw, Peter Herrmann, Heiko Krumm: Constraint-Oriented Formal Modelling of OO-Systems, in L. Kutvonen, H. König, and M. Tienari(eds.), in 2nd Int. Working Conf. on Distributed Applications and Interoperable Systems (DAIS'99), Kluwer Academic Publisher, 1999, pp. 345-358.

[GHK00] Günter Graw, Peter Herrmann, Heiko Krumm: Verification of UML based real-time systems designs by means of cTLA+, in 2nd IEEE International Symposion on Object-oriented Real-time distributed computing(ISORC 2000), IEEE Computer Science Press, Los Angeles, 2000, pp. 86-95.

[L94] Leslie Lamport: The temporal Logic of Actions, ACM Transactions on Programminng Languages and Systems, vol. 16, no. 3, pp. 872-923, 1994.

[L02] Leslie Lamport: Specifying Systems, http://www.lamport.org, 2002.

[KC01] http://www.kc.com/as_site/home.html, 2001.

[KC02] http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_06v2_5b.pdf.

[HMK96] Carsten Heyl, Arnulf Mester, Heiko Krumm: cTc a tool supporting the construction of cTLA specifications, T. Magaria and B. Steffen editors, Proceedings of the TACAS-96, LNCS 1055, Springer, 1996, pp. 407-411.

[HK00a] P. Herrmann, H. Krumm: A framework for modeling transfer protocols, Computer Networks, Volume 34, Nr. 2, 2000, pp. 317-337.
[HK00b] P. Herrmann, H. Krumm: A framework for the hazard analysis of chemical plants, Proceedings op the 11th IEEE International Symposium on Computer aided control system design (CAASD2000), IEEE CSS, Anchorage, omnipress 2000, pp. 35-41.
[MB02] Stephen J. Mellor, Marc J. Balcer: Executable UML, Addison-Wesley, 2002.
[OAL] Brigepoint OAL, http://www.projtech.com/pdfs/bp/oal.pdf.
[OOA] Brigepoint OOA, http://www.projtech.com/pdfs/bp/ooa.pdf.
[pUML] http://www.puml.org.
[SPHGJ01] Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Allain Le Guennec, Jean-Marc Jézéquel: Using UML Action Semantics for Executable Modeling and Beyound, 2001.
[SBK01] Natasha Sharygina, James C. Brown an Robert Kurshan: Formal Object-Oriented Analysis for Software Reliability: Design for Verification, http://www.cs.texas.edu/users/browne/NewPapers/FASE20011.pdf, 2001.
[UML] UML Metamodel Version 1.4, http://www.omg.org/uml.
[YML99] Yuan Yu, Panagiotis Manolios, Leslie Lamport Model Checking TLA+ Specifications, in Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors, LNCS 1703, Springer-Verlag, 1999, pp. 54-66.