Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 109 (2004) 57–69
www.elsevier.com/locate/entcs

Towards Inconsistency Handling of Object-Oriented Behavioral Models

Jochen M. Küster[1]

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany

Abstract

With the Unified Modeling Language being used in diverse contexts, the ability of defining and checking customized consistency conditions is of increasing importance. Often, consistency checks rely on existing formal analysis tools such as model checkers and require the translation of models into input languages of these tools.
The technique of inconsistency handling aims at systematically dealing with inconsistencies detected by such consistency checks. Resolution of inconsistencies typically involves changing the model, with guidance of the software engineer or completely automated in the ideal case. As a consequence, in cases where formal analysis tools are used for consistency checks, the output of these tools must be presented in a form understandable for the software engineer.
In this paper, we develop a concept for inconsistency handling of object-oriented behavioral models and discuss how graph transformation can be used for reconstructing UML models from outputs generated by analysis tools.

Keywords: consistency management, UML, model transformation, model analysis

1 Introduction

With the Unified Modeling Language [10] being used in diverse contexts (given by application domain, methodology, and platform) the ability of defining and checking *customized consistency conditions* is of increasing importance.

Besides well-formedness rules in OCL as part of user-defined UML profiles, little support is available for customized specification and checking of consistency conditions. In particular, no support is provided to the developer

[1] Email: jkuester@uni-paderborn.de

doi:10.1016/j.entcs.2004.02.056

to specify behavioral consistency conditions, like specific notions of compatibility between statecharts and sequence diagrams. If the UML is to be used successfully in different contexts, method and tool support are required to specify and check consistency conditions both at the syntactic and the semantic level.

In [3,1] we have developed a methodology for consistency management in UML-based development. Our approach to defining consistency concepts is by means of partial translations of models into a formal method that provides a language and tool support to formulate and verify semantic consistency conditions. The translation is specified by means of graph transformation rules at the level of the meta model [7,1]. Depending on the development process and application domain, different translations into different semantic domains may be defined. Taking this approach, different forms of consistency can be treated: Horizontal consistency problems involving submodels of a larger model at the same abstraction level as well as vertical consistency problems occurring between submodels belonging to different phases within a development process (and typically requiring a consistency condition expressing a certain refinement condition). Further, both syntactic consistency conditions as well as semantic consistency conditions can be defined.

Consistency checks are specified by translating one or more submodels and checking the specified consistency conditions. In case of an inconsistency, an activity of inconsistency handling is required that supports the software engineer in dealing with the inconsistency, such as resolving or tolerating it.

Up to now, no general concept for inconsistency handling of object-oriented behavioral models with regards to semantic consistency conditions is available. Inconsistency handling of these models is complicated by the fact that results of a consistency check are given in terms of a formal language used for this approach and that the correction of such inconsistencies usually requires complex changes of the model that influence model semantics.

In this paper, we study the problem of inconsistency handling for object-oriented behavioral models, following our general methodology for consistency management developed in [3]. First, we briefly report on existing work on consistency checking, introducing an example consistency problem. Then, we elaborate on the concept of inconsistency handling. Finally, we focus on using graph transformation during inconsistency handling for reconstructing UML models from outputs generated by model checkers.

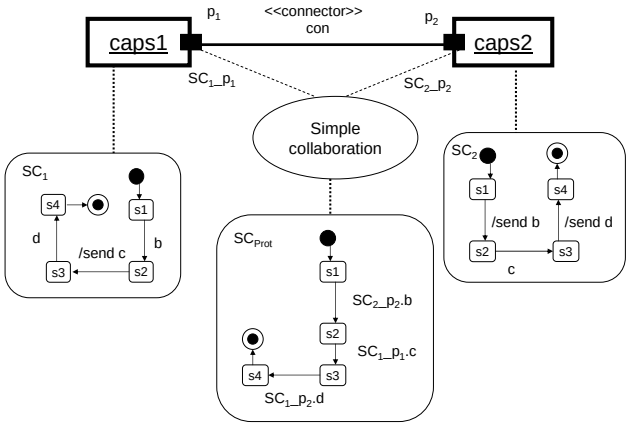

Fig. 1. Consistency problem example

2 Checking Behavioral Consistency using the Consistency Workbench

We will start with introducing a behavioral consistency problem called *Protocol Consistency* known from UML-RT [13], now also arising in the core UML 2.0 specification [11].

In Figure 1, two structured objects caps1 and caps2 are shown, connected by a connector via two ports p_1 and p_2. Attached to this connector is behavior modeled in the protocol statechart SC_{Prot}. The behavior of the structured objects is specified in two statecharts, named SC_1 and SC_2. Intuitively, the interaction arising from executing the statecharts of the structured objects should conform to the protocol specified in the protocol statechart.

For ensuring behavioral consistency in this case, a formal consistency concept is required. Such a consistency concept consists of a set of submodel types (in this case the protocol statechart, the statechart of structured object and the collaboration), a mapping of these submodel types into a common semantic domain and a set of consistency conditions formalizing the informally noted form of consistency. Using such a formal consistency concept, consistency checks can be introduced in a development process for ensuring consistency: Within a consistency check, concrete submodels are translated into the semantic domain and then consistency conditions are evaluated.

For protocol consistency, we can define two different consistency conditions: For *weak protocol consistency* we require that all traces of the interaction of the structured objects statecharts must be contained in the set of traces of the protocol statechart. For *strong protocol consistency* we additionally assume that all the traces of the protocol statechart must occur in the system. Extending the statechart SC_1 by introducing another transition sending another event will violate the condition of *weak protocol consistency*.

Removing the last transition of SC_2 will violate the condition of *strong protocol consistency*.

In previous work, we have reported on the details of such a consistency concept. As a running example, we have chosen the formal method CSP [8] and the model checker FDR [4] to evaluate consistency conditions. To support the software engineer in the complex task of translating submodels and defining consistency checks, we have developed the consistency workbench [2]. Briefly, this workbench allows the definition of rule-based translations of UML models into semantic domains and the definition of consistency checks as workflows, composing activities for translation and triggering external model checkers. Currently, the consistency workbench contains pre-defined translations of statecharts and collaborations to CSP and allows the execution of consistency checks for the previously described consistency problem.

With regards to protocol consistency, the result produced by the consistency workbench is currently a trace violating the consistency condition.

3 Inconsistency Handling: Foundations and Techniques

Inconsistency handling [14] is a notion for activities and techniques that aim at dealing with inconsistencies in multi-view sofware development. Depending on the types of languages and abstractions used within model-based development, quite different inconsistency handling techniques have been developed. In general, one can distinguish between *changing actions* and *non-changing actions* and the general decision of either *tolerating* or *resolving* an inconsistency. Inconsistency handling comprises the identification of these actions as well as the evaluation of their costs and the evaluation of risks of not resolving an inconsistency. Concerning inconsistency handling of object-oriented models, we will restrict ourselves to the discussion of actions for handling them, leaving the evaluation of risks and costs to future work.

Concerning requirements for an approach for inconsistency handling of object-oriented models, firstly the approach should allow the resolution of inconsistencies by the software engineer. Further, it should avoid the introduction of new inconsistencies and should be performable directly in the UML model in order to be feasible for ordinary software engineers. Finally, the approach should be at least semi-automatable in order to not impose too much work on the software engineer.

In addition, there are certain requirements for inconsistency handling that are specific to our approach of consistency management for object-oriented behavioral models: The approach

- should reuse the information about the inconsistency given by the model

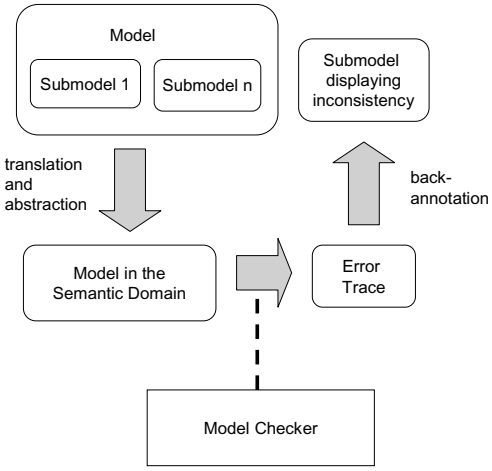

Fig. 2. Concept of consistency check and back-annotation

checker (e. g. a trace to an inconsistency), and adequately present it to the software engineer,

- must rely on the software engineer for the resolution of inconsistencies,
- must allow the software engineer to influence the resolution of inconsistencies,
- should be integratable in our consistency workbench.

In particular, the requirement that resolution must rely on the software engineer is important to notice. This is because changing the semantics of a model is usually not to be done without interaction of the software engineer. As a simple example, consider the consistency problem between sequence diagrams and statecharts, where the statechart is to realize the scenarios expressed in sequence diagrams. A common consistency problem occurs if a scenario cannot be realized by the statechart. Here, there are in principle two ways of inconsistency resolution: The scenario is removed from the set of valid scenarios, or the scenario is added to the behavior supported by the statechart. Although tool support for removing a scenario or adding a scenario is possible (e. g. by adapting work on statechart synthesis from a set of sequence diagrams [15]), the decision must be made by the software engineer. Additionally, changing of a statechart for supporting another scenario may be difficult and also influence other behavior relevant for scenarios. As a consequence, it is difficult to fully automate the changing of a statechart in this case.

In order to enable inconsistency management, it is important that the inconsistency is detected. This is performed by a consistency check. However, the result of such a consistency check is given in terms of the language in the semantic domain. In case of an inconsistency, this will usually be a trace to an

error state. For being usable by the software engineer, this trace to an error state should be given in terms of the source language, in our case UML. Hence, the problem is to translate the output of the model checker back into the source language UML. Here, two approaches can be distinguished, referred to in the following as *back-annotation*: We can either annotate an existing UML model or we can construct a new UML model for the purpose of illustrating the inconsistency. As an example for annotation of an existing model, consider the consistency problem of inconsistent timing constraints modeled in a sequence diagram. Here, a note mentioning the inconsistency should be displayed in the already existing sequence diagram. An example of constructing a new UML model occurs if we construct a sequence diagram showing the scenario under which the violation of the protocol occurs.

In Figure 2, the problem of back-annotation is illustrated. As the translation into the semantic domain for performing the consistency check abstracts from the UML model, the problem of back-annotation is not straightforward, because in the process of back-annotation the original level of abstraction must be reached in order to be useful to the software engineer.

Using the technique of back-annotation, it is possible to display a UML model illustrating the inconsistency found. In order to enable the software engineer to resolve the inconsistency, often further support is needed. This support usually depends on the type of underlying consistency concept i. e. for the wide range of inconsistencies quite different types of supports could be needed.

One possible support for the software engineer consists of simulation. In the previous example, the overall model can be simulated along the trace to the protocol violation. Such a simulation can be done on the level of UML models, including the sequence diagram created during back-annotation.

Further support consists of semi-automatic algorithms for including or removing a trace violating the protocol: Either the protocol statechart can be extended to also include the trace or the individual statecharts can be restricted not to include the trace. This can be done by special algorithms defined on the statecharts.

In the following, we will restrict ourselves to the problem of back-annotation.

4 Back-Annotation of UML models using Graph Transformation

To illustrate the problem of back-annotation, consider a trace produced by the FDR model checker in the con-

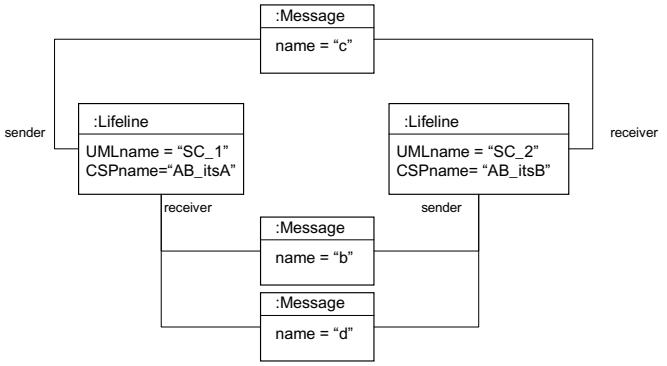

Fig. 3. Reconstruction model

text of protocol violation. For the simple example, we get $\langle AB\_itsB\_p\_out.send\_b, AB\_itsA\_p\_out.send\_c, AB\_itsB\_p\_out.send\_d\rangle$ This denotes that the system of the structured objects performs the trace on UML level $\langle SC_2\_p_2.b, SC_1\_p_1.c, SC_2\_p_2.d\rangle$ which might not be included in a faulty protocol statechart.

The goal of back-annotation is to construct from such a trace given in CSP a UML model helping the software engineer to handle the inconsistency in the UML model. One problem associated with back-annotation is that we must be able to express information compatible with the source UML model: in the process of translation and abstraction to the semantic domain CSP, we may have renamed concepts from UML to CSP and left out details of the UML model. For example, in the concrete example $AB\_itsB\_p\_out.send\_b$ corresponds to a sending of SC_2 via the port p_2, the information $out.send$ is something added during translation into CSP. Further, the details of the receiver of the message b have been left out and are not visible in the CSP trace.

What is needed here is an approach that allows us to reconstruct the abstraction level needed when translating the CSP error trace back into a UML model. This approach must be intertwined with the translation from UML into CSP at the first place. There, the information needed for back-annotation must be determined and stored.

Our concept for tackling this problem is as follows: When designing the translation from UML to CSP (i. e. when abstracting), one has to determine which aspects will be necessary to reconstruct a UML model from the more abstract CSP model. These aspects have to be mapped into a so-called *reconstruction model* which can then be used for generating/back-annotating a UML model in the process of inconsistency handling.

In Figure 3, we show such a reconstruction model for our running example. The information needed for reconstruction of a sequence diagram from a simple trace, given as output from the FDR model checker can be described as follows:

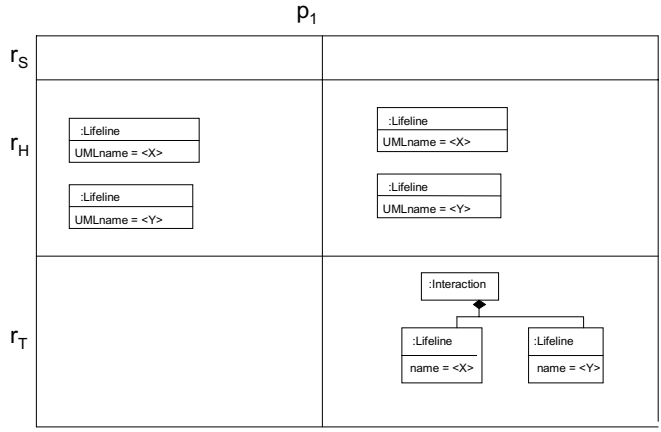

Fig. 4. Compound rule p_1, consisting of r_S, r_H, and r_T

Firstly, for each participant of the interaction, a lifeline in a sequence diagram should be present. In the concrete interaction, we need a Lifeline for SC_1 and one for SC_2. Secondly, each message sent to an object by another object must be marked in the reconstruction model. Further, we need to reconstruct the UML names of the sender and receivers, for that purpose we include a UMLname and CSPname in the lifelines.

In the following, we will assume the existence of a set of graph transformation rules for translating UML statecharts and collaborations to CSP. Details of this can be found in [6]. The overall goal is to construct a sequence diagram containing lifelines for the objects that participate in the protocol, together with the trace to the protocol violation.

The general approach is to use information of the trace given in CSP and the reconstruction model to construct a target model, the desired sequence diagram. The back-annotation itself will be specified using graph transformation. We will make use of a synchronized compound graph transformation [9], a form of adapted triple graph transformation [12]. Such a compound transformation has already been used for generating the CSP model in the Consistency Workbench. Briefly, a compound transformation can be described by a set of compound graph transformation rules. A compound graph transformation rule itself consists of a fixed number of graph transformation rules, where each graph transformation rule describes the transformation of a model, such as the source model (the trace in CSP), the reconstruction model, or the target model (the sequence diagram to be constructed). Within a compound transformation rule, all transformation rules are coupled by the use of common variables. Formally, a compound transformation rule can be represented by a typed attributed graph transformation rule, by joining all individual transformation rules at the attribute vertices.

For constructing a sequence diagram from a trace given in CSP, we make

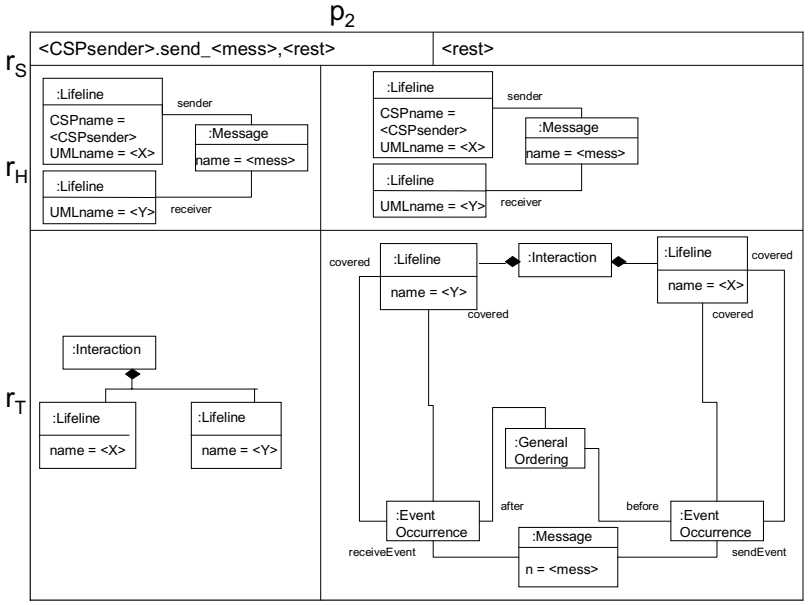

Fig. 5. Compound rule p_2, consisting of r_S, r_H, and r_T

use of a compound transformation consisting of three transformations, one for the trace called r_S, one for the reconstruction model called r_H and one for the target model, called r_T. Overall, we need two compound transformation rules, the first one generating an empty sequence diagram, shown in Figure 4 and one for then iteratively filling the sequence diagram depending on the CSP trace.

In Figure 4, we show the compound rule p_1 for generating an empty sequence diagram. In UML 2.0, such an empty sequence diagram consists of an Interaction together with two Lifelines. The lifelines are named according to the lifelines saved in the reconstruction model, using the UMLname of each lifeline. Note that in this case r_H has identical left and right sides and does not change the reconstruction model.

In Figure 5, compound rule p_2 is displayed, for starting generating the sequence diagram from a CSP trace. The rule matches in the trace the name of the object and the name of the message received and then finds the corresponding lifeline in the reconstruction model that receives message. Note here that we assume that message names are disjoint, otherwise more than one match could occur in the reconstruction model.

The main idea of reconstructing a sequence diagram from a trace is contained in r_T, the third transformation contained in compound rule p_2. Here, the information matched in r_S and r_H is assembled to create new model elements in the sequence diagram: For a message sent by an object in the CSP trace (represented by *object.send_message*), a new message with that name is

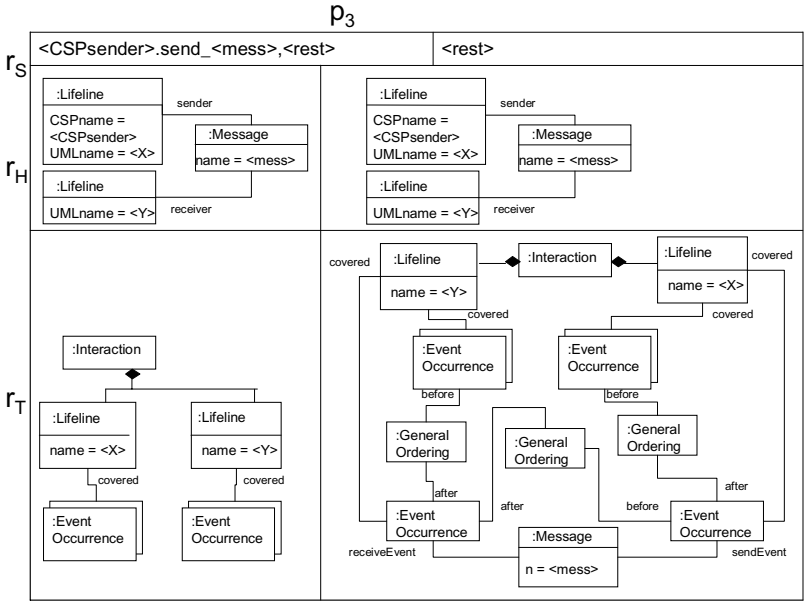

Fig. 6. Compound rule p_3, consisting of r_S, r_H, and r_T

created in the sequence diagram and attached to the corresponding lifelines. Note here that the lifeline that receives the message is only known by making use of the reconstruction model. Additionally, two new EventOccurrences are created, together with a GeneralOrdering between them. One of the event occurrences represents the sending of the message, the other the receiving of the message. These event occurrences are attached to the lifelines and the message accordingly.

After having applied rule p_2 once, a sequence diagram with one message exchange will exist. As a consequence, for each additionally message to be introduced, we will apply rule p_3 which follows the basic idea of p_2 but deals with the case that already EventOccurrences exist at the lifelines: in a sequence diagram in UML 2.0, all events along a lifeline are ordered. In order to achieve this, we must attach all existing event occurrences to the newly created event occurence, in both existing lifelines. This is done by using two multi-objects on the left side of r_T, one for each lifeline, and by attaching the newly created event occurrences accordingly.

We will now apply the rules p_1 and p_2 on the sample trace given by the model checker and to the reconstruction model in Figure 3. We assume that rules p_1 and p_2 are only applied once whereas rule p_3 is applied as long as possible, for being able to handle traces of arbitrary length. In our approach, such a control flow can be specified formally (for details see [9]). Note here that no infinite rule applications can occur because each application reduces the size of the CSP trace. After applying rule p_1 we get an empty sequence diagram,

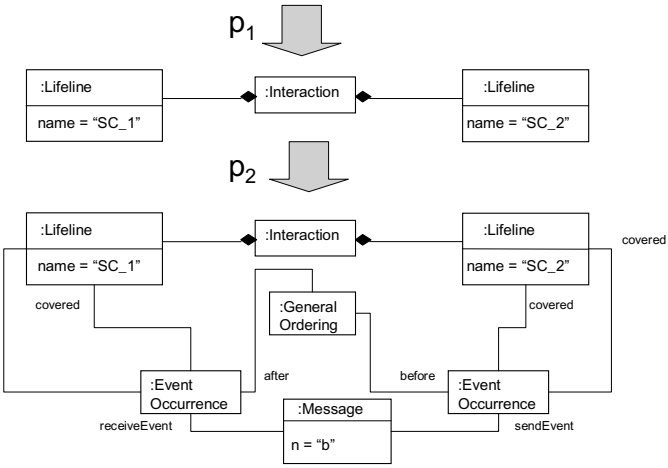

Fig. 7. Applying rules p_1 and p_2

after applying rule p_2 once we get the sequence diagram shown in Figure 7 (note that we only show the reconstructed sequence diagram as metamodel instance, leaving out the reconstruction model and the CSP trace model).

5 Conclusion

In this paper, we have discussed the problem of inconsistency handling of object-oriented behavioral models. Inconsistency handling of these kind of models is difficult because typically behavioral consistency checks are performed in an external semantic domain. We have first introduced requirements for inconsistency handling and developed principal techniques: Within inconsistency handling, pre-defined guidelines to deal with the inconsistency, and simulation approaches for illustrating the inconsistency to the software engineer are of importance. In particular, the technique of back-annotation i.e. the reconstruction of a new UML model or modification of an existing UML model for inclusion of additional information is crucial for successful, convenient inconsistency handling of object-oriented behavioral models.

In the second part of this paper, we have elaborated on the problem of back-annotation. The concept of a reconstruction model has been introduced which serves the purpose of saving the information necessary within back-annotation during the translation into a semantic domain. We have further elaborated how an existing approach of graph transformation using a rule format of compound rules which has already been successfully used for translating UML models into a semantic domain can also be used for back-annotation.

Our approach to back-annotation has been illustrated using a simple example where a CSP trace is used for constructing a UML sequence diagram.

Future work can be performed in the following directions: Firstly, the problem of back-annotation must be studied in depth for different consistency problems, possibly leading to a richer set of transformations for back-annotation. A first idea would be to also include negative traces and make use of the rich set of model elements provided by the UML. Furthermore, a distinction between mandatory and potential behavior could be useful (see e.g. Haugen et al. [5]). Secondly, we aim at including our techniques of back-annotation into the *Consistency Workbench*. Further, techniques such as specialized algorithms for user-directed inconsistency resolution should be developed.

References

[1] G. Engels, R. Heckel, and J. M. Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *LNCS*, pages 272–287. Springer-Verlag, 2001.

[2] G. Engels, R. Heckel, and J. M. Küster. The Consistency Workbench - A Tool for Consistency Management in UML-based Development. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 356–359. Springer-Verlag, 2003.

[3] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In Volker Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, 2001.

[4] Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*, 1997.

[5] Ø. Haugen and K. Stølen. STAIRS - Steps to analyze interactions with refinement semantics. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 388–402. Springer-Verlag, 2003.

[6] R. Heckel, J. M. Küster, N. Bandener, B. Gueldali, I. Jahnich, C. Koepke, and M. Weking. Automatische Qualitätssicherung von UML Modellen. Bericht der Projektgruppe. Technical Report TR-RI-03-245, Universität Paderborn, December 2003.

[7] R. Heckel, J. M. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains. In H.-J. Kreowski and P. Knirsch, editors, *Proceedings of the Appligraph Workshop on Applied Graph Transformation*, pages 11–22, March 2002.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] J. M. Küster, R. Heckel, and G. Engels. Defining and Validating Transformations of UML Models. In J. Hosking and P. Cox, editors, *IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003) - Auckland, October 28 - October 31 2003, Auckland, New Zealand, Proceedings*, pages 145–152. IEEE Computer Society, 2003.

[10] Object Management Group (OMG). *OMG Unified Modeling Language Specification, Version 1.5. OMG document formal/03-03-01*, March 2003.

[11] Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.

[12] A. Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proceedings WG'94 International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. LNCS 903, Springer-Verlag, 1994.

[13] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–262. Springer-Verlag, 1998.

[14] G. Spanoudakis and A. Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. *Handbook of Software Engineering and Knowledge Engineering*, 2:329–380, 2001.

[15] J. Whittle and J. Schumann. Generating statecharts designs from scenarios. In *22nd International Conference of Software Engineering, Limerick, Ireland, Proceedings*, pages 314–323. IEEE Computer Society Press, 2000.