

# Towards Explicit Behavioral Consistency Concepts in the UML

Jochen M. Küster  
Department of Computer Science  
University of Paderborn - C-LAB  
Germany  
jkuester@upb.de

Jan Stehr  
omp AG  
Paderborn  
Germany  
jan.stehr@omp.de

## Abstract

*In this paper, we show how different behavioral consistency concepts can be formed for sequence diagrams and statecharts. Our approach relies on the partial translation of models into a semantic domain and on the definition of explicit consistency conditions. Partial translations and consistency conditions can be combined to form an explicit consistency concept. In order to make our approach applicable in practice, we discuss the issue of tool support and favor the development of a so-called consistency workbench.*

## 1. Introduction

With the Unified Modeling Language [10] becoming the industrial standard for object-oriented modeling, its application for both scenario and statechart modeling is gaining importance.

Whereas until now, scenarios are usually expressed in MSCs and statecharts in some statechart dialect, the UML can be considered as a family of languages and incorporates both sequence diagrams, a variant of MSCs, and statecharts. Nevertheless, despite the syntactic integration into a common metamodel, with regards to semantic integration no coherent concept is available.

As a consequence, currently there is no integrated consistency concept for behavioral models. On the one hand, this is not surprising as it is evident in the debate about scenario and statechart modeling and the development of various synthesis and verification algorithms. It is indeed more than doubtful that a single consistency concept can be established for all application domains and make its way into the language definition.

Nevertheless, consistency of sequence diagrams and statecharts is an important issue within any development process because sequence diagrams are often used in an early phase of the development to fix user requirements, while

statecharts are used for deriving an implementation. If consistency is ensured within model-based development, then testing of the code can be reduced and the overall software process becomes more reliable.

Our approach to overcoming the problem of ensuring consistency of sequence diagrams and statecharts is to develop an explicit consistency concept. This consistency concept is supposed to be adapted depending on the application domain and development process. It is based on a formal translation of behavioral models into a suitable semantic domain and explicit consistency conditions. Such a formal translation is complemented by an operational translation using graph transformation which can be used for automated translation of UML models into the formal semantic domain. Consistency conditions specified within the semantic domain are verified by existing model checkers.

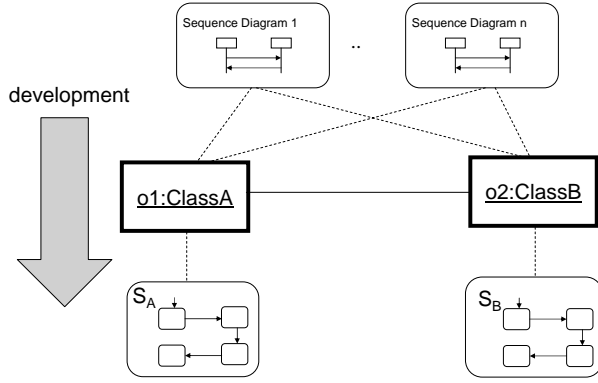
It is our belief that there are quite different consistency concepts possible for different application domains. However, in order to enable automated consistency checks, these consistency concepts shall be made explicit and sufficient support must be offered to the software engineer for being able to deal with different consistency concepts.

## 2. Our Approach

Our approach to establish consistency of sequence diagrams and statecharts is based on a general methodology for specification and analysis of object-oriented behavioral models [3]. In the following, we first concentrate on the technical issues of the methodology applied to the consistency problem of sequence diagrams and statecharts. Then, we elaborate on the idea of explicit consistency concepts and discuss the problem of offering appropriate tool support.

### 2.1. Problem Description

Within system development, typically during an analysis phase sequence diagrams are created which model typical



**Figure 1. Problem description**

scenarios of the system. The collection of these sequence diagrams can already be seen as an early behavioral model of the system. Later within the development process, statecharts are used for modeling full functional behavior of each class. In Figure 1, this approach is illustrated.

These two complementary approaches to modeling behavior raise a consistency problem between the behavioral model described by a set of sequence diagrams and the behavioral model obtained from the collection of statecharts. Due to the different development processes followed and the different applications of both scenario-based and state-based behavioral modeling, a fixed consistency concept is currently not available and also not part of the UML standard.

In the literature, different approaches are followed to establish consistency of these models. Generally, we can distinguish between two different approaches. One can be characterized by the goal of establishing consistency by deriving statecharts from a set of scenario sequence diagrams. These *synthesis approaches* [13] [8] prove beneficial in several cases but also have their disadvantages when it comes to the problem of checking already given sequence diagrams and statecharts for consistency. Although, in such a case, the synthesis approach allows to derive a set of statecharts from sequence diagrams, these statecharts must then manually be compared with existing statecharts. To show such an equivalence between two statecharts requires a sound formal equivalence notion for statecharts and, in practice, additional tool support. Slight disadvantages can also be seen in the problem that current synthesis approaches require additional information for each scenario or make implicit assumptions.

Another, quite different approach to establishing consistency of sequence diagrams and statecharts is to provide a translation into a formal language and provide a set of consistency conditions to hold (see for example the work by Cheng et al. [1] and Moreira et al. [9]). Following such a

formal approach, one problem consists in the limited capacity of model checkers to verify consistency conditions. Another problem that arises in the context of UML is the need for many different formalizations due to different application domains and development processes.

In the following, we will first briefly show how consistency can be established by translating both sequence diagrams and statecharts to the process algebra CSP [7]. Once this has been achieved, consistency conditions can be expressed in terms of CSP assertions. Using formal translations and consistency conditions, explicit consistency concepts for sequence diagrams and statecharts can be established. The translation of statecharts to CSP has been inspired by Hiemer [6]. The details of this translation have been introduced by Stehr in [11].

The following formalization can be seen as one example how consistency of sequence diagrams and statecharts can be established. Other than existing approaches, our main aim is not the formalization itself but rather the support of the software engineer when defining and dealing with such a formalization. Therefore, after discussing the details of the formalization itself we will concentrate on the issue how the software engineer can be supported in dealing with such formalizations.

## 2.2. Translation of Sequence Diagrams to CSP

Given a set of UML sequence diagrams, we translate these to the process algebra CSP in three steps. First, we construct the class process  $P_k$  of a class  $k$  by extracting all sequences of message interactions modeled in all sequence diagrams for objects of this class. Secondly, we construct so-called buffer processes  $B_{P_k}$  for each class  $k$  to process incoming messages. Thirdly, we interleave all buffer and class processes to a system of processes  $System_{SD}$  (note that we assume uniqueness of each class of a process in the system for reasons of brevity of our explanations given here).

For construction of the class process, we extract all message interactions of an object  $o$  of a class  $k$  and encode it in a process  $P_o$ . This process has the form

$$P_o = Message_{o,1} \rightarrow \dots \rightarrow Message_{o,n} \rightarrow P_o$$

where  $Message_{o,i} = k_r.send.async.message$  or  $Message_{o,i} = k_r.receive.async.message$  for the asynchronous sending of a message to an object of class  $k_r$  or receiving of an asynchronous message. All processes  $P_o$  are interleaved to a process  $System_{SD,k} = \parallel P_o$ .

A buffer process of capacity one for asynchronous communication is obtained by the process

$$Buffer_k = k.send.async?msg \rightarrow k.receive.async!msg \rightarrow Buffer_k$$

We can now form another system of processes, comprising the behavior of the objects in all sequence diagrams:

$$System_{SD} = (\parallel System_{SD,k} \parallel (\parallel B_k))$$

where  $B_k$  is a buffer of a desired size obtained by composing processes of the type  $Buffer_k$ .

### 2.3. Translation of Statecharts to CSP

The translation of statecharts to CSP proceeds in three steps. In the first step, each statechart translates to a statechart process, mirroring the syntactical structure of the statechart. Secondly, an environment for event processing is constructed. Thirdly, all statechart processes are interleaved with the environment and an additional scheduler process.

We translate each statechart into a corresponding parameterized statechart process  $SC$  which corresponds to the syntactical structure of the statechart. The generic form of this process  $SC_{CSP}$  is

$$\begin{aligned} SC(state) &= act\_SC \rightarrow \\ &if(state == s_1) then State(s_1) \\ &else... \\ &if(state == s_m) then State(s_m) \\ &else STOP \end{aligned}$$

The State processes for a state  $s$  define the behavior of a statechart within a state  $s$ . We further assume that transitions are of the form  $(s, e_i, a_i, t_i)$  where  $e_i$  denotes the event,  $a_i$  the action and  $t_i$  the target state of the transition. Then the State processes are defined as follows:

$$\begin{aligned} State(s) &= e_1?x_{e_1} \rightarrow \dots e_n?x_{e_n} \rightarrow \\ &if(x_{e_1} == 1) then \\ &post\_a_1 \rightarrow ackn\_SC \rightarrow SC(t_1) \\ &else... \\ &if(x_{e_n} == 1) then \\ &post\_a_n \rightarrow ackn\_SC \rightarrow SC(t_n) \end{aligned}$$

In the case of transitions with either empty event or empty action, the corresponding CSP expression is simply omitted.

So far, we have restricted ourselves to the description of simple, flat statecharts. In the case of composite states, the process  $SC$  must be extended by the interleaving of additional parameterized processes for modeling behavior of composite states and - in the case of concurrent composite states - their regions.

In addition to the translation of a statechart, we must also construct the environment of the statechart. The environment consists of individual processes for each event occurring in the statechart administrating the occurrence of each event.

For different verification purposes, we can now distinguish between a number of ways of combining statechart processes and environment processes. Concentrating on one statechart, say for a class  $k$ , we construct

$$System_{SC,k} = GS \parallel (SC_{CSP,k} \parallel Env_k)$$

Given an UML object diagram with objects where the behavior of each object is described by a statechart, we can, similar to the sequence diagrams, construct the system of all statechart processes and environment processes and then compose them in parallel as follows:

$$System_{SC} = GS \parallel ((SC_{CSP,1} \parallel \dots \parallel SC_{CSP,n}) \parallel (Env_1 \parallel \dots \parallel Env_n))$$

### 2.4. Specification of Consistency Conditions

After translation of both sequence diagrams and statechart diagrams to CSP, consistency conditions can be specified.

These conditions can ensure different types of consistency. Quite common is to require that all possible interactions specified within sequence diagrams for a class  $k$  should be possible in the refined system where the behavior is specified by a statechart of that class.

#### Condition 1 (Each scenario is valid within the statechart)

*Each scenario specified for a class  $k$  within a set of sequence diagrams is valid within the statechart of that class  $k$  iff  $System_{SC,k} \setminus \alpha_{SC,k} \sqsubseteq_{\mathcal{T}} System_{SD,k}$ .*

This condition can be extended to a stronger condition formulated over the systems of processes obtained from sequence diagrams and statecharts:

#### Condition 2 (All scenarios are valid within the statecharts)

*All scenarios specified within a set of sequence diagrams are valid for all statecharts iff  $System_{SC} \setminus \alpha_{SC} \sqsubseteq_{\mathcal{T}} System_{SD}$ .*

In addition, our formalization also allows the detection of implied scenarios, similar to that of Uchitel et al. [12]. For that, we can establish the following consistency conditions:

#### Condition 3 (Statechart does not contain implied scenario)

*A statechart for a class  $k$  does not contain a scenario not already specified within the set of sequence diagrams iff  $System_{SD} \sqsubseteq_{\mathcal{T}} System_{SC,k} \setminus \alpha_{SC,k}$ .*

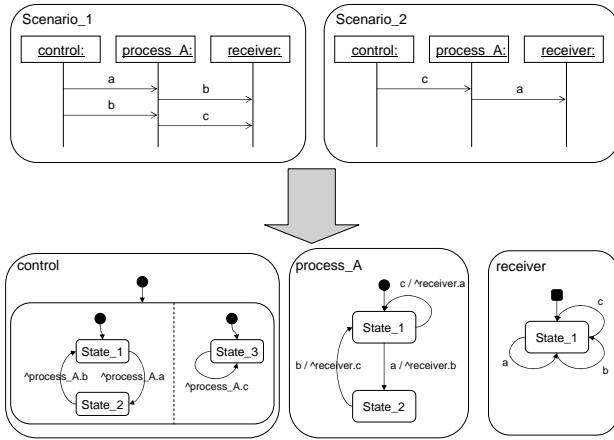


Figure 2. Example sequence diagrams and statecharts

## 2.5. Example

In this section, we illustrate the concept of our translation with a simple example. In order to keep the example short, we assume the following simplifications: Buffers have a size of 1 and messages are communicated asynchronously. Furthermore, we assume that each scenario described by a sequence diagram may be processed any time, i. e. also during the execution of another scenario. After a scenario is completed (last event occurred), it may be repeated. In addition, we will use the machine-readable version of CSP used by the model checker FDR [4].

The sequence diagrams in Figure 2 translate to the process  $System\_SD$  (see below; process definitions of  $System\_SD\_control$  and  $System\_SD\_receiver$  have been omitted).

```
Process_A_Scenario_1 =
control.receive.async.a ->
receiver.send.async.b ->
control.receive.async.b ->
receiver.send.async.c ->
Process_A_Scenario_1
```

```
Process_A_Scenario_2=
control.receive.async.c ->
receiver.send.async.a ->
Process_A_Scenario_2
```

These scenario processes are assembled to the processes  $System\_SD\_A$  and  $System\_SD$  as follows:

```
System_SD_A = Process_A_Scenario_1 ||| Process_A_Scenario_2
System_SD = ((System_SD_control ||| System_SD_A
||| System_SD_receiver) [|alpha_Buffer|]
(NBuf_control(1) ||| NBuf_Process_A(1)
||| NBuf_receiver(1)))
```

The statechart  $process\_A$  of Figure 2 translates to the following process (the statecharts for  $control\_sc$  and  $receiver\_sc$  are translated in the same way).

```
Process_A_sc(state,static) =
act Process_A ->
if (state == State_1) then
process_A_a?x_process_A_a ->
process_A_c?x_process_A_c ->
if (x_process_A_a == 1) then
post_receiver_b ->
post_in_State_2 ->
ackn_Process_A ->
Process_A_sc(State_2,false)
else
if (x_process_A_c == 1) then
post_receiver_a ->
post_in_State_1 ->
ackn_Process_A ->
Process_A_sc(State_1,false)
else
post_in_State_1 ->
ackn_Process_A ->
Process_A_sc(State_1,true)
else
if (state == State_2) then
process_A_b?x_process_A_b ->
if (x_process_A_b == 1) then
post_receiver_c ->
post_in_State_1 ->
ackn_Process_A ->
Process_A_sc(State_1,false)
else
post_in_State_2 ->
ackn_Process_A ->
Process_A_sc(State_2,true)
else STOP
```

In addition, we need to construct the environment and build the overall processes  $System\_SC\_A$  and  $System\_SC$ .

To make the statechart processes and processes derived from sequence diagrams comparable, we assume the application of a renaming operator to the systems. The events  $post\_receiver\_x$ ,  $process\_A\_x$ ,  $control.receive.async.x$  and  $receiver.send.async.x$  become  $receive\_x$  and  $send\_x\_to\_receiver$  for any message  $x$ . All other events of the systems are hidden.

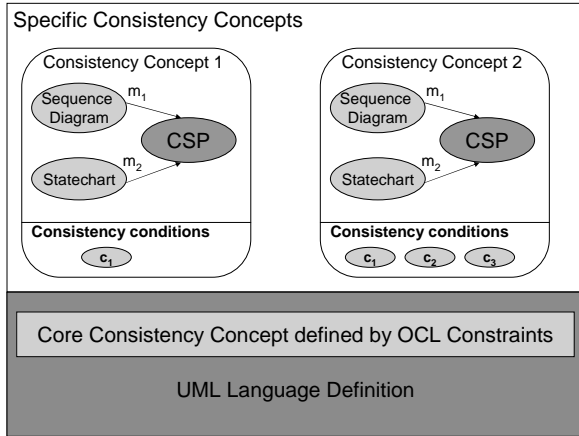
Concerning condition 1, this condition can be checked with the assertion  $assert\ System\_SC\_A\ [T= System\_SD\_A]$ .

Condition 2 can be checked by the statement  $assert\ System\_SC\ [T= System\_SD]$  and does not hold. Due to the assumed combination, the scenario's traces include the possibility that  $c$  is received by  $Process\_A$  after receiving  $a$  - this is not possible for the process derived from the statechart.

Condition 3, expressible by  $assert\ System\_SD\ [T= System\_SC]$  should hold because the process specified by the statecharts is not able to show behaviour extending the given scenarios.

## 2.6. Consistency Concepts and Consistency Checks

Having formalized both sequence diagrams and statecharts and defined several consistency conditions for them, this foundation can be used for defining explicit consistency



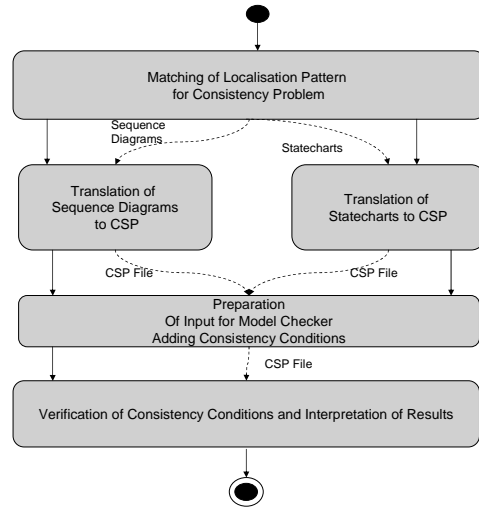
**Figure 3. Explicit consistency concepts for the UML**

concepts. In general, a consistency condition (sometimes also called consistency rule) is a predicate that determines whether or not a model is considered consistent with respect to that particular condition. Concerning UML, the in-built consistency conditions can be considered as syntactic as they are directly specified on the syntax of models.

In addition to syntactic consistency conditions, a consistency concept may also contain semantic consistency conditions. In order to specify these semantic conditions, a mapping into a semantic domain is also considered part of the consistency concept. Examples for such mappings have been described in the previous sections for sequence diagrams and statecharts. Similarly, the conditions described above are so-called semantic consistency conditions. Intuitively, they deal with the semantics of behavioral models because they require certain conditions about the execution of a system to hold.

In Figure 3, we illustrate our ideas about explicit consistency concepts for the application in UML. In the lower part, we visualize the UML language definition which comprises a built-in core consistency concept specified by the OCL constraints. On top of these, application domain and development process specific consistency concepts can be established. For example, concerning our case of sequence diagrams and statecharts, we may only require that each scenario is also valid in the corresponding statechart and decide that we do not care about issues such as implied scenarios, leading to a consistency concept 1 in the figure. Or, requiring a much stronger notion of consistency, we might construct a concept requiring all consistency conditions to hold.

On the basis of a consistency concept, consistency checks can be defined in order to validate that a model



**Figure 4. A sample consistency check**

is consistent with the concept. A consistency check must therefore validate the consistency conditions of a consistency concept. Within our approach, such a check involves the translation of a model into a semantic domain, the verification of consistency conditions by a model checker and an interpretation of the results.

Informally, the specification of such a consistency check can be visualized by an activity diagram extended by mechanisms for modeling object flow. In Figure 4, the consistency check for scenario sequence diagrams and statecharts is shown (with object flow visualized by dotted arrows). Within the first activity, a UML localization pattern is used for locating and identifying, within a larger UML model, those sequence diagrams and statecharts relevant for the consistency check. These are then given to the translation activities. Within the translation activities, the partial translation to CSP is performed. Resulting CSP files are then assembled to a single file which can be handed over directly to the model checker.

## 2.7. Tool-Supported Consistency Checks

In order to make consistency management feasible within a development process, the software engineer has to be supported by appropriate tools. Currently, case tools support rather limited consistency checks, based on the UML language definition and validation of OCL constraints.

In an ideal case, an appropriate tool should support the software engineer in all necessary activities to establish a consistency concept. At University of Paderborn, we are currently developing a so-called consistency workbench, aiming at

- the support of the definition of consistency problems by offering a consistency problem catalogue,
- the support of the definition of a partial translation of UML models into a semantic domain,
- the support of the definition of consistency checks on the basis of previously defined partial translations.

Concerning the first item, we are of the opinion that when modeling with UML, often the same consistency problems may occur, due to the usage of same diagram types. One example for this is the consistency problem of sequence diagrams and statecharts. We think that such consistency problems can better be identified in an existing development process if they are properly documented in a catalogue. Method engineers can then resort to such a catalogue and see whether a well-known consistency problem also occurs within their development process.

With respect to partial translations of UML models, these are quite difficult to define and require great expertise. We have recently explored a graph transformation approach [5] [2] [11] which allows the translation to be specified by a set of compound graph transformation rules. In our case, such a compound graph transformation rule consists of two parts, a source production rule specified by a UML metamodel extract and a target production rule rule in the semantic formalism, here CSP. As we do not want to change the source model, the source production is the identical production, with equal left and right side. In Figure 5, two compound graph transformation rules are shown for translating statecharts to CSP.

Graph transformation rules of this form can then be used to specify a model transformation from a given source UML model to a target CSP model. The semantics of rule applications is briefly described as follows: Given a concrete UML model, a match for the UML metamodel extract is searched for in the concrete UML model. Once such a match is found, a match of the left side of the target production is searched for in the CSP model. Once the two matches have been found, the match of the CSP model is replaced by the right side of the target production.

The consistency workbench allows the software engineer to specify such rule-based model transformations. The workbench further contains a transformation engine that, given a source model, uses a set of graph transformation rules to construct a target model. By offering support for formalization, the software engineer can convert an existing formal declarative specification, as the one above, in an operational one consisting of graph transformation rules. In the future, existing formalizations can be used to construct a catalogue of reusable partial translations.

With regards to consistency checks, their definition can take place by taking partial translations and consistency

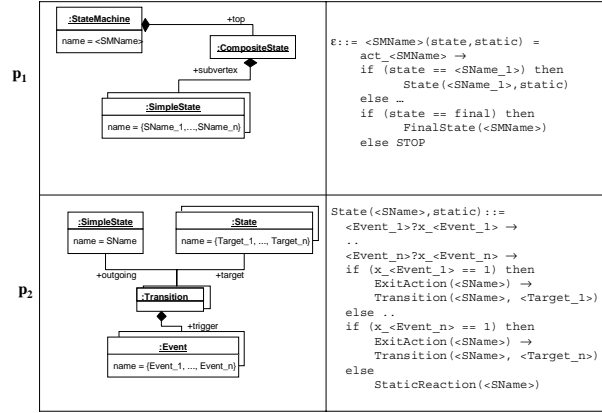


Figure 5. Two rules for statechart translation

conditions as building blocks. Our idea is to provide within the workbench the partial translations, consistency conditions and model checkers as building blocks such that consistency checks as in Figure 4 can be formed. For that purpose, the workbench contains a visual editor for definition of consistency checks. Such a definition of a consistency check is then interpreted by a consistency check execution engine.

Currently, our consistency workbench is still a research prototype, concentrating on the semantic domain CSP and on the partial translation of sequence diagrams and statecharts, as described in this paper. However, we hope that in the future we can include further partial translations into other semantic domains and thereby demonstrate the usability of the workbench.

### 3. Discussion

Consistency of behavioral models such as sequence diagrams and statecharts is a difficult problem because there is no universally valid consistency concept available, suitable for all development processes and application domains. As a consequence, several different approaches have been developed, from constructive approaches to declarative approaches such as ours.

In our opinion, it is important to make the consistency concept one uses within a development process explicit. We have proposed a declarative approach for achieving this, based on the partial translation of sequence diagrams and statecharts into the semantic domain of the process algebra CSP and the explicit definition of consistency conditions. Consistency conditions and partial translations can be composed to form different consistency concepts. For a consistency concept, consistency checks can be developed and required to be executed and ensured within the overall soft-

ware development process. Concerning the issue of tool support, we are currently developing a consistency workbench which should support the software engineer in the definition and execution of consistency checks.

From our experiences we can draw several conclusions. It is desirable to make the consistency concept explicit also for constructive approaches. Indeed, we think it would be beneficial if each of the constructive approaches could be accompanied by a declarative approach and vice versa, although we assume that this might not always be possible. Then, having a declarative approach specified for a certain development process, it would be possible to select a particular tool within the sets of constructive approaches to ensure consistency. In order to reach such a goal, the relationship between the various synthesis approaches and declarative approaches must be analyzed in detail.

Another conclusion can be drawn when considering the current UML language definition. In its present form, it does not support any type of behavioral consistency concept. In the future, it might be desirable to provide partial formalizations for different application domains in form of different consistency concepts. This could be achieved by incorporating a behavioral formalism into UML, similar as the OCL constraint language, or by adapting current initiatives for action semantics.

With respect to future work, we are currently exploring the definition of consistency concepts based on a translation of sequence diagrams and statecharts into Timed Automata.

## References

- [1] B. Cheng, L. Campbell, and E. Wang. Enabling automated analysis through the formalization of object-oriented modeling diagrams. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, pages 433–442. IEEE Computer Society, 2000.
- [2] G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML metamodel. 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, 2001.
- [3] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. 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] R. Heckel, J. M. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains. In *Proceedings of the Appligraph Workshop on Applied Graph Transformation*, 2002.
- [6] J.-J. Hiemer. *Statecharts in CSP: Ein Prozessmodell in CSP zur Analyse von STATEMATE-Statecharts*. DrKovac Verlag, 1999.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [8] K. Koskimies, T. Männistö, T. Systä, and J. Tuonmi. Automated support for modeling oo software. In *IEEE Software*, volume 15, pages 87–94, 1998.
- [9] A. Moreira and R. Clark. Combining object-oriented modeling and formal description techniques. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pages 344 – 364. LNCS 821, Springer Verlag, 1994.
- [10] Object Management Group. *Unified Modeling Language Specification, version 1.4*, September 2001.
- [11] J. Stehr. *Semantische Konsistenzprüfung von UML-Verhaltensdiagrammen zur Modellierung von eingebetteten Systemen*. Diploma thesis, University of Paderborn, 2003.
- [12] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 74–82. ACM Press, 2001.
- [13] J. Whittle and J. Schumann. Generating statecharts designs from scenarios. In *22nd International Conference of Software Engineerin, Limerick, Ireland, Proceedings*, pages 314–323, 2000.