Assisted Generation of Frame Conditions for Formal Models

Philipp Niemann[1] Frank Hilken[1] Martin Gogolla[1] Robert Wille[1,2]

[1]Institute of Computer Science
University of Bremen
28359 Bremen, Germany

[2]Cyber Physical Systems
DFKI GmbH
28359 Bremen, Germany

e-mail: {pniemann,fhilken,gogolla,rwille}@informatik.uni-bremen.de

***Abstract*—Modeling languages such as UML or SysML allow for the validation and verification of the structure and the behavior of designs even in the absence of a specific implementation. However, formal models inherit a severe drawback: Most of them hardly provide a comprehensive and determinate description of transitions from one system state to another. This problem can be addressed by additionally specifying so-called frame conditions. However, only naive "workarounds" based on trivial heuristics or completely relying on a manual creation have been proposed for their generation thus far. In this work, we aim for a solution which neither leaves the burden of generating frame conditions entirely on the designer (avoiding the introduction of another time-consuming and expensive design step) nor is completely automatic (which, due to ambiguities, is not possible anyway). For this purpose, a systematic design methodology for the assisted generation of frame conditions is proposed.**

I. Introduction

Nowadays, formal models are more and more used in the design of complex, embedded systems as they allow for a precise specification of corresponding designs even in the absence of a specific implementation. A significant benefit of those descriptions is additionally provided by the fact that they allow for the detection of design flaws in a very early stage. Then, eliminating these flaws is by far easier and cheaper as if they were detected later in the actual implementations.

The *Unified Modeling Language* (UML, [1]) is a widely established, general-purpose language and is applied in software as well as hardware design. Employing the *Object Constraint Language* (OCL, [2]), further restrictions can be applied on the respective models and behavioral aspects can be defined precisely; in particular the functionality of operations. Due to their general acceptance and wide usage, we concentrate on UML/OCL models in the remainder of this paper[1]. The description means provided by the combination of UML and OCL allow for the creation of very sophisticated models. However, constraints provided in terms of declarative constraint languages like OCL can often be formally satisfied in many different fashions – allowing for intended, but also unintended behavior. This is crucial, since a comprehensive and determinate description of the transition from one system state to another is not provided in these cases.

This problem is known as the *frame problem* [3] and can be addressed by additionally specifying so-called *frame conditions*. These define which elements of the model are eligible to changes during the execution of an operation and, even more important, which are *not*. Various approaches for formulating frame conditions have been discussed [4], [5]. One recently suggested scheme is the specification of so-called *invariability clauses* (following the "modifies only"-approach proposed in [5]). Although this construct is not yet part of the OCL standard, it is already applied in scientific studies (see e.g. [6]) and considered in both, academia and industry. However, the actual process of generating frame conditions for a given model (e.g. in terms of invariability clauses) is a cumbersome task which requires the consideration of a significant amount of model elements as well as their relations and side effects [7], [8]. Thus far, only naive "workarounds" exist that do not provide a satisfactory solution. They either apply rather trivial heuristics or completely rely on a manual creation – requiring designers with a deep design understanding and, eventually, leading to a new, time-consuming and error-prone design step (this is discussed in more detail later in Section III).

In this paper, we envision a solution to this problem. We respect that, due to ambiguities of formal models, a completely automatic approach will not be possible or would lead to unsatisfactory results. At the same time, we aim for not leaving the burden of generating frame conditions entirely on the designer. As a consequence, a systematic methodology for the generation of frame conditions is proposed which assists the designer in a comprehensible fashion and with automatic methods in this process. More precisely, in a first step, analyses are performed on the model which lead to the creation of so-called hypotheses, i.e. proposals and suggestions of model elements to be considered for invariability clauses. Already this significantly aids the designer by pinpointing her to relevant model elements which need to be investigated when generating frame conditions. Afterwards, the designer is provided with evaluations on how the generated hypotheses would restrict the behavior of the model in detail. This leads to plausibility checks based on which the designer can make well-informed decisions on whether to add, to refine, or to discard a hypothesis. By this, designers are provided with a systematic methodology on how to deal with the frame problem for the first time.

The remainder of this work is structured as follows: Section II provides a brief review on both, formal models provided in UML/OCL as well as the frame problem. The central problem of generating frame conditions, including the basic idea of the proposed solution, is covered in Section III. Afterwards, our envisioned systematic methodology for the generation of hypotheses as well as their evaluation is outlined in Section IV and Section V, respectively. Section VI discusses the resulting methodology and concludes the paper.

II. Background

This section briefly reviews the main description means of UML/OCL which are considered in this work. First, we cover how to specify the structure and behavior of systems. Afterwards, the frame problem of behavioral models and possible solutions in terms of frame conditions are discussed. All aspects are illustrated by means of a running example.

A. Structure and Behavior in UML/OCL

In general, UML/OCL offers a broad variety of description means which allow for the specification of the structure and the behavior of systems. *Class diagrams* are usually applied to represent the structure of a system. Here, *classes* describe the main components of a system. Each class is composed of *attributes* (representing the information that is stored in the class) and *operations* (representing possible actions that can be executed in order to change the system state).

[1] Nevertheless, the methodology proposed here can similarly be applied to models provided in other modeling languages as well.

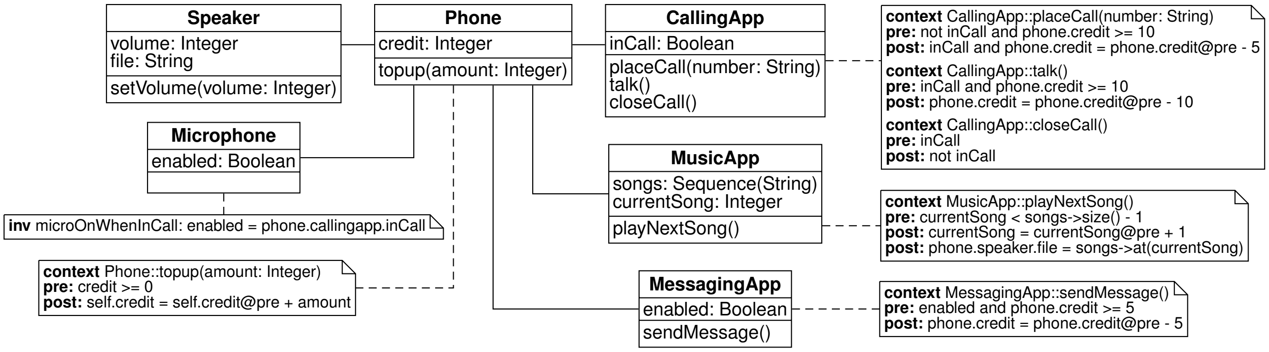
Speaker
volume: Integer
file: String
setVolume(volume: Integer)
Phone
credit: Integer
topup(amount: Integer)
CallingApp
inCall: Boolean
placeCall(number: String)
talk()
closeCall()
context CallingApp::placeCall(number: String)
pre: not inCall and phone.credit >= 10
post: inCall and phone.credit = phone.credit@pre - 5
context CallingApp::talk()
pre: inCall and phone.credit >= 10
post: phone.credit = phone.credit@pre - 10
context CallingApp::closeCall()
pre: inCall
post: not inCall
Microphone
enabled: Boolean
MusicApp
songs: Sequence(String)
currentSong: Integer
playNextSong()
context MusicApp::playNextSong()
pre: currentSong < songs->size() - 1
post: currentSong = currentSong@pre + 1
post: phone.speaker.file = songs->at(currentSong)
inv microOnWhenInCall: enabled = phone.callingapp.inCall
context Phone::topup(amount: Integer)
pre: credit >= 0
post: self.credit = self.credit@pre + amount
MessagingApp
enabled: Boolean
sendMessage()
context MessagingApp::sendMessage()
pre: enabled and phone.credit >= 5
post: phone.credit = phone.credit@pre - 5

Fig. 1. UML/OCL model of a smartphone specification

Example 1. *Throughout the paper we are making use of a running example which is shown in Fig. 1. The model represents an early design of a smartphone with apps. Specifications for both, the designated software components (apps) as well as the designated hardware components (speaker, microphone), are provided in terms of individual classes which have attributes and operations depending on their purpose.*

In order to express further properties or restrictions, textual constraints provided in OCL can be added to a model – so-called *invariants*. These are associated to a certain class and must be satisfied by all system states.

Example 2. *In order to constrain that the microphone of the smartphone system in Fig. 1 is only enabled in conversations, invariant* microOnWhenInCall *(associated to* Microphone*) has been added to the model. This invariant accesses the* CallingApp *and its* inCall*-attribute in order to determine whether a call is currently taking place and, hence, to restrict its own* enabled *value.*

Finally, the behavior of operations can be refined by *pre-* and *postconditions* – again provided in OCL. Pre- and postconditions (denoted by pre and post) are considered only in the context of an operation call. More precisely, an operation can only be invoked if its corresponding precondition is satisfied. Afterwards, the succeeding system state needs to satisfy the operation's postcondition.

Example 3. *Consider the operation* CallingApp::placeCall *of the smartphone system in Fig. 1. According to the precondition, this operation can only be invoked if the user is not already engaged in a call (denoted by* not inCall*) and still has more than 10 units of credits left (denoted by* phone.credit >= 10*). After the execution of this operation, a system state must be reached where* inCall *is set to* true *and where the* phone*'s* credit *is reduced by 5 units in comparison to the pre-state (which is referred to using the annotation* @pre*).*

B. The Frame Problem of Behavioral Models

The abstract fashion of models provided in UML/OCL enables engineers to precisely describe a system in early stages of the design process. These formal descriptions already allow for the validation and formal verification of design drafts even in the absence of a specific implementation. Common issues which can already be addressed at this level and for which corresponding (automatic) methods have been proposed in the past are, e.g., structural aspects like *consistency* of models (see e.g. [9], [10]), and behavioral aspects like *reachability* of certain good or bad states (see e.g. [11], [12]).

However, particularly when the validation and verification of behavioral descriptions is considered, UML/OCL models as well as other formal models inherit a severe drawback: Most of them hardly provide a comprehensive and determinate description of the transition from one system state to another through an operation call. In fact, the declarative descriptions provided by the pre- and postconditions often specify the system state before and after the execution of the respective operation with a particular focus on obvious model elements only – leaving possible constraints and restrictions on the remaining model elements open. Furthermore, the corresponding descriptions often seem unique from an intuitive designer's perspective, but, in fact, allow several interpretations from a formal perspective.

Example 4. *Consider the operation* MusicApp::playNextSong() *of the smartphone system in Fig. 1. From a designer's perspective, the pre- and postconditions shall describe that the attribute* currentSong*, which represents the current integer position of the playlist* songs*, is increased by one and the current song is put on the* Speaker*'s* file*. The behavior of* currentSong *is indeed clearly described by the first postcondition. However, the second postcondition formally allows for several interpretations, i.e. could be satisfied in various, unintended ways. For example, literally any song could be put on the* Speaker*'s* file*, since no restrictions are applied on the sequence* songs*. Besides that, no information or restriction is provided on the behavior of the remaining model elements.*

Focusing on obvious model elements and intuitive descriptions is understandable and also a necessity – in particular in early design stages where the respective models have to be comprehended by a large group of various stakeholders. At the same time, the resulting ambiguities lead to inaccurate descriptions which poses major challenges for validation, verification, and later implementation. For these purposes, it is essential to know precisely which model elements are eligible to changes even if these changes are not specified in detail.

A straight-forward solution to address this so-called the *frame problem* [3] is to additionally enrich the model by so-called *frame conditions*. For this purpose, again OCL can be applied: In a naive fashion, all model elements which are to remain unchanged can be forced to do so e.g. by adding terms like model_element = model_element@pre to the postconditions. However, this approach is error-prone and and impracticable for larger systems where side-effects and implicit dependencies can easily be overlooked. Instead, recent discussions, besides others particularly those in [4], [5], have shown that it is often more elegant to specify what *may* change rather than what may *not* change. This *modifies only*-scheme led to the introduction of so-called *invariability clauses* – a shorthand notation which can be employed to specify variable model elements and their scope of change in a very precise, but compact fashion [5]. Even though this construct is not yet part of the OCL standard, it is received well and has already been used frequently, e.g. in [6].

Example 5. *Consider again* MusicApp::playNextSong() *from Fig. 1. Following the intended meaning, only the properties* currentSong *and* file *are supposed to be changed. These modifications are supposed to be conducted only within the scope of the* MusicApp *on which the operation is called (*self*) and within the scope of the* Speaker *object* phone.speaker*, respectively. Hence, a suitable frame condition resolving all the ambiguities discussed above can be formulated by the invariability clause* modifies only: self.phone.speaker::file, self::currentSong.

III. Generation of Frame Conditions

Invariability clauses offer a convenient way for the specification of frame conditions which significantly improves the expressiveness of UML/OCL models. On the one hand, they allow designers and stakeholders to keep focused on the more intuitive description of the system. At the same time, they provide necessary clarifications on ambiguities and inaccuracies for the applied verification engines. However, actually generating the clauses for a given model remains a cumbersome task. In particular for larger models, the number of elements as well as their relations and side effects to be considered becomes significantly large. At the same time, no systematic approach for the generation of frame conditions exists so far.

As a consequence, frame conditions are hardly applied in practice. Hence, the frame problem reviewed in the previous section basically remains unsolved – to the best of our knowledge, for all existing validation and verification approaches. Thus far, the frame problem is either not covered at all or simple "workarounds" are used to deal with missing frame conditions, namely solutions based on:

- *Manual specification,* i.e. completely rely on a manual refinement of the model like, e.g., the "filmstripping"-approach proposed in [12]. However, this leaves the burden solely on the designer who requires the respective design understanding and, at the same time, is time-consuming and error prone. A noteable case study that demonstrates weaknesses and limits of this strategy can be found in [7].

- *Implicit specification,* i.e. simply apply naive schemes such as enforcing all model elements which are not restricted by originally provided constraints to remain unchanged (and, hence, ignore implicit relations and side-effects of the respective model elements). Approaches such as proposed in [11] are representatives for this.

In order to avoid the problems caused by these unsatisfactory solutions, the *generation of frame conditions* has to become an integral part of the design process. For this purpose, a solution is required which neither leaves the burden of the generation entirely on the designer (avoiding the introduction of another time-consuming and expensive design step) nor is completely automatic (which, due to the ambiguities and inaccuracies, will not lead to satisfactory results anyway).

In this work, we propose such a solution by providing a systematic design methodology for the generation of frame conditions. While eventually the designer has to decide on the explicit addition of invariability clauses, we propose to assist her with automated methods for the generation and evaluation of so-called *hypotheses*. More precisely, the proposed methodology leads to the generation of frame conditions in two steps:

- First, analyses are performed on the model which lead to hypotheses, i.e. proposals and suggestions of model elements to be considered for invariability clauses.
- Second, the designer is provided with evaluations on how the generated hypotheses would restrict the behavior of the model in detail. Based on this, she can make a well-informed decision on whether to add, to refine, or to discard a hypothesis.

In the remainder of this paper we will outline first ideas on these two steps, especially on the assistance by automated methods.

IV. Generation of Hypotheses

Relying on the *modifies only*-scheme, the problem of generating frame conditions boils down to the question which model elements shall or shall not be restricted by invariability clauses. While a definitive classification on that cannot always be provided automatically due to the ambiguities and inaccuracies discussed above, it can be observed that, for each operation, a distinction between the following categories can be made:

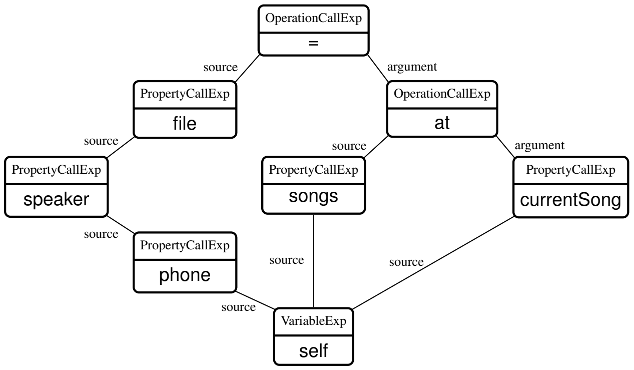

Fig. 2. AST for the postcondition of MusicApp::playNextSong()

- *Variable elements*, i.e. model elements that are evidently meant to be modified by the respective operation since this modification is (precisely) constrained. These elements have to occur in the invariability clause.
- *Ambiguous elements*, i.e. model elements where it remains unclear whether they are supposed to be modified within a state transition or not. These properties may occur in the invariability clause, but require further inspection.
- *Unaffected elements*, i.e. model elements that are evidently not meant to be affected by the respective operation. These elements are not to occur in the invariability clause as they are not at all involved in the respective operation and, hence, should simply keep their current value.

Already such a classification of the model elements significantly aids the designer in the generation of frame conditions. In fact, it pinpoints her to ambiguous elements which require a more detailed consideration, while variable and unaffected elements only need to be approved and can, afterwards, directly be added to the invariability clauses or not, respectively.

In order to generate hypotheses for this classification, we propose to focus on the postconditions of the respective operation as the primary source. Postconditions constrain the changes performed during an operation call; all model elements which might be modified can likely be assumed to be referenced therein. For this purpose we envision an automatic analysis of the *Abstract Syntax Trees* (AST) which are constructed from the postconditions using the metamodel for OCL expressions (defined in [2, Chapter 8.3]) and provide a semantic profile of the postconditions.

Example 6. *Consider the AST in Fig. 2 which represents the postcondition of the operation* MusicApp::playNextSong(). *Here, each node and link is labelled by the corresponding expression type and relation kind, respectively. In the left sub-tree, only the model element* file *is recognized variable as it is the last element in a chain of PropertyCallExpressions. The first part of this chain is used to construct the corresponding scope-term* self.phone.speaker::file. *From the right sub-tree, the elements* songs *and* currentSong *are classified ambiguous, since they occur in an argument term. Note that OCL formally considers the right-hand-side of an equation as the argument of the "="-operation called on the left-hand-side. Thus, the model element* songs *would be classified variable if both sides of the equation were swapped.*

Postconditions are not always unambiguous, as illustrated by Example 4. Addressing this issue, standard interpretations for ambiguous OCL constructs have been suggested in [13] based on an extensive field study. Although the motivation and conclusions were quite different, the observations of this study can also be transferred to the context considered here.

Additionally, there may also be elements which do not occur in the operation's postconditions, but are connected to other model elements that were identified to be variable in a different way. Clearly, these dependencies also have to be considered in order to obtain a complete set of hypotheses for the invariability clauses.

Example 7. *Consider the constraint that the microphone shall be enabled during phone calls (as expressed by the invariant* microOnWhenInCall *of the* Microphone *class in Fig. 1). Assume that the attribute* inCall *has been classified as a variable element. Then, the attribute* enabled *shall be classified accordingly. The appropriate scope-term is computed by backwards navigation, i.e.* Microphone→Phone→CallingApp *becomes* CallingApp→Phone→Microphone*, resulting in the scope-term* self.phone.microphone::enabled*.*

V. Validation of Hypotheses

The automatic scheme proposed in the previous section provides hypotheses whether model elements shall be included in the invariability clauses or not. For finally generating the frame conditions, these hypotheses are validated and ambiguities are further eliminated. In this section, we outline how existing verification approaches and modeling tools can be applied to assist the designer in this task.

A very effective measure to validate the plausibility of a hypothesis is to consider corresponding *execution scenarios* for the respective operation. Existing consistency checkers or model finders such as provided by [10] can be applied for this purpose. They can be used to create scenarios in which e.g. a pre-state satisfying all preconditions and a post-state satisfying all postconditions occur such that only variable model elements are modified. If no such scenario can be determined, the respective classification might be too weak, i.e. further model elements should be considered as variable (e.g. from the set of ambiguous elements). At the same time, those scenarios can be used to validate that only model elements show changes that are supposed to do that. Moreover, the designer may specify own execution scenarios, preferably including as much changes between pre- and post-states as possible. Then, these scenarios can be analyzed and used to determine whether ambiguous model elements are meant to be variable or unaffected.

Further support for the validation of hypotheses can be provided by modeling tools such as [14]. They allow for highlighting variable and ambiguous model elements in the class diagram and, hence, pinpoint the designer to ambiguous model elements left to be specified. These can be addressed individually by defining execution scenarios to be determined by consistency checkers or model finders. More precisely, the designer may query the determination of a scenario that becomes possible only if a highlighted model element is considered to be variable or if the scope-term of a model element is modified, but is not possible for the original form. Doing this systematically, first for object instantiation and destruction, then for modification of associations, and, finally, for changes of attributes, all ambiguities are successively eliminated. Eventually, an invariability clause results that precisely describes the intended frame conditions.

VI. Discussion & Conclusion

In this work, we considered the generation of frame conditions for behavioral formal models e.g. based on UML/OCL. For this purpose, we proposed a systematic methodology which, first, generates hypotheses of model elements to be considered for invariability clauses and, afterwards, validates the plausibility of them. Automatic analysis schemes for the generation as well as validation of the hypotheses assist the designer in this process. For the first time, this provides a systematic approach on how to deal with the frame problem.

As an obvious drawback, the proposed methodology still relies on a manual interaction with the designer. However, this would be required by any other solution as well – ambiguities of descriptions can never completely be captured by automated methods and, eventually, have always to be finally decided by the designer. However, the methodology provided here significantly assists the designer in this process by providing suggestions as well as plausibility checks.

Previously proposed solutions (i.e. the "workarounds" discussed in Section III) are far away from providing such an extensive support. In fact, only naive schemes (e.g. considering a model element as variable if and only if it is referenced in a postcondition) have been applied thus far.

In contrast, the methodology proposed here offers the following two advantages:

- The designer is provided with proposals and suggestions of model elements to be considered for invariability clauses. Already taking those in a naive fashion would lead to much more elaborated frame conditions than those obtained by previously proposed methods.
- The designer is extensively assisted during the validation of the proposed model elements. This leads to a clear and fast definition of frame conditions. Moreover, these validations may even pinpoint the designer to possible inconsistencies in the design.

To the best of our knowledge, the methodology proposed here is the first systematic approach to this problem.

Acknowledgment

This work was partially funded by the German Research Foundation (DFG) under grants GO 454/19-1 and WI 3401/5-1 as well as within the Reinhart Koselleck project DR 287/23-1.

References

[1] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language reference manual.* Essex, UK: Addison-Wesley Longman Ltd., 1999.

[2] Object Management Group (OMG), "Object Constraint Language Version 2.4," 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/PDF

[3] A. Borgida, J. Mylopoulos, and R. Reiter, "On the frame problem in procedure specifications," *IEEE Trans. Software Eng.*, vol. 21, no. 10, pp. 785–798, 1995.

[4] A. D. Brucker, M. P. Krieger, and B. Wolff, "Extending OCL with null-references," in *MoDELS*, 2009, pp. 261–275.

[5] P. Kosiuczenko, "Specification of invariability in OCL - specifying invariable system parts and views," *Software and System Modeling*, vol. 12, no. 2, pp. 415–434, 2013.

[6] M. P. Krieger, A. Knapp, and B. Wolff, "Automatic and efficient simulation of operation contracts," in *GPCE*, 2010, pp. 53–62.

[7] M. A. G. de Dios, C. Dania, D. A. Basin, and M. Clavel, "Model-driven development of a secure ehealth application," in *Engineering Secure Future Internet Services and Systems - Current Research*, ser. LNCS. Springer, 2014, vol. 8431, pp. 97–118.

[8] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, "Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models," in *TAP*, 2014, pp. 99–116.

[9] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *TAP*, 2009, pp. 90–104.

[10] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL Models Using Boolean Satisfiability," in *DATE*. IEEE, 2010, pp. 1341–1344.

[11] M. Soeken, R. Wille, and R. Drechsler, "Verifying dynamic aspects of UML models," in *DATE*. IEEE, 2011, pp. 1077–1082.

[12] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France, "From application models to filmstrip models: An approach to automatic validation of model dynamics," in *Modellierung*, 2014, pp. 273–288.

[13] J. Cabot, "From declarative to imperative UML/OCL operation specifications," in *Conceptual Modeling*, ser. LNCS, vol. 4801. Springer, 2007, pp. 198–213.

[14] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.