Extracting Frame Conditions from Operation Contracts

Philipp Niemann Frank Hilken Martin Gogolla Robert Wille
Department of Computer Science, University of Bremen, 28359 Bremen, Germany
e-mail: {pniemann,fhilken,gogolla,rwille}@informatik.uni-bremen.de

***Abstract*—In behavioral modeling, operation contracts defined by pre- and postconditions describe the effects on model properties (i.e., model elements such as attributes, links, etc.) that are enforced by an operation. However, it is usually omitted which model properties should not be modified. Defining so-called frame conditions can fill this gap. But, thus far, these have to be defined manually – a time-consuming task. In this work, we propose a methodology which aims to support the modeler in the definition of the frame conditions by extracting suggestions based on an automatic analysis of operation contracts provided in OCL. More precisely, the proposed approach performs a structural analysis of pre- and postconditions together with invariants in order to categorize which class and object properties are clearly "variable" or "unaffected" – and which are "ambiguous", i.e. indeed require a more thorough inspection. The developed concepts are implemented as a prototype and evaluated by means of several example models known from the literature.**

I. Introduction

The modeling of behavior plays an important role in modern system development. Being able to simulate models and to detect errors early in the development significanly reduces the costs to correct them. The *Unified Modeling Language* (UML) together with the *Object Constraint Language* (OCL) offers a broad spectrum of methods to describe behavior, e.g., by means of activity diagrams or operation contracts using pre- and postconditions.

In this work, we concentrate on the description of behavior in class diagrams using operation contracts defined by OCL pre- and postconditions. We propose a solution to a problem that occurs when verifying such models. Pre- and postconditions describe the effects of an operation in a declarative way, but – by design – do not pose an implementation. As a result, verification engines have to make decisions about which model properties may be changed in order to satisfy the operation contract. However, this decision is not trivial and cannot automatically be inferred from the constraints themselves (Section II reviews this problem in more detail).

This problem of determining the precise behavior from a declarative operation contract is called the *frame problem*. The pre- and postconditions only focus on the desired effects of an operation and typically omit those parts of the model that may not change. However, this does not mean that they are indeed unaffected, e.g., when implicit dependencies through invariants or model constraints require changes. Several solutions (to be discussed in detail in Section II as well) exist to enable more precise definitions of the affected model properties by manually describing all elements that may be affected by the operation. But none of the approaches assists the modeler in extracting the required information from the model. Thus, the ability to precisely describe the operations comes with a substantial cost.

As a solution, we propose a methodology[1] which analyzes operation contracts of a model and, based on that, categorizes all model properties into three categories: *variable* properties that are definitely affected by the operation; *unaffected* properties that are not related to the operation or shall not change; and *ambiguous* properties that cannot automatically be determined due to multiple possible interpretations of the OCL expressions. The goal of this categorization is to assist the modeler in precisely describing operations that behave as intended and are ready to be used for verification purposes. In fact, having to consider only a (small) subset of model properties (namely those categorized as ambiguous) significantly reduces the time and effort required for this process.

As a result, our approach supports the modeler by an automatic determination of those properties that are relevant to the operation – based on the their occurrences in pre- and postconditions as well as implicit relation to invariants. The approach has been implemented in a prototype and evaluated by means of several example models known from the literature. Our evaluation clearly confirms the potential of the proposed methodology.

The rest of the paper is structured as follows. In Section II the considered problem is reviewed and discussed with related work. Section III presents the methodology to categorize model properties and, in Section IV, the methodology is evaluated by means of several example models. The paper closes with a conclusion and a presentation of future work in Section V.

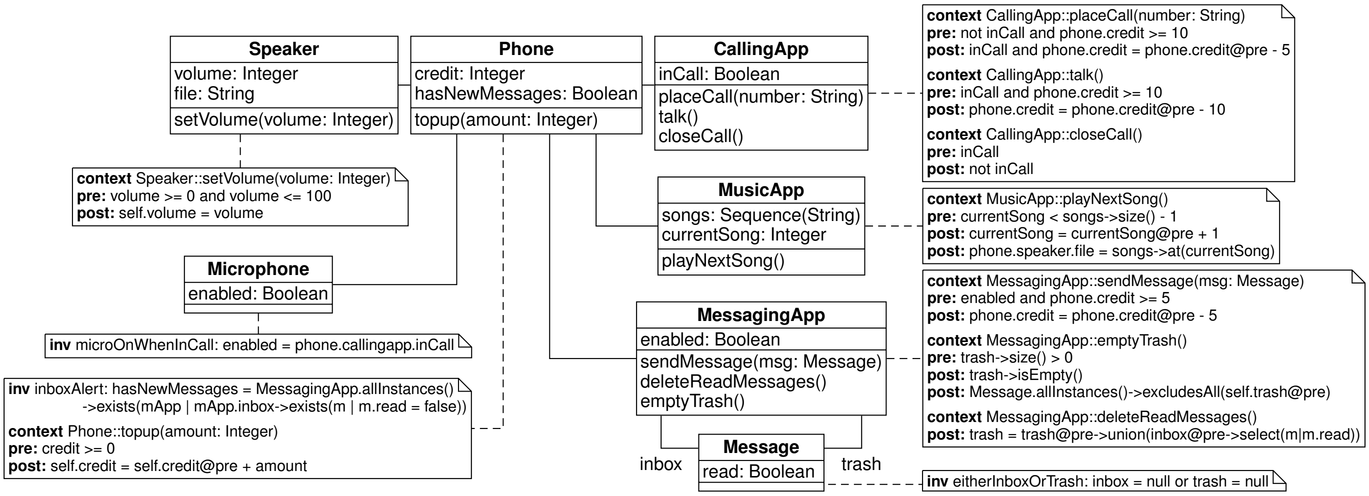
II. Background

In this section, the frame problem of behavioral models in terms of UML/OCL class diagrams is discussed and previously proposed solutions are reviewed. This provides the necessary background needed in order to keep this work self-contained and motivates the methodology introduced in this paper.

[1] Preliminary ideas of the proposed methodology have already been sketched before in [1]. In order to keep the present paper self-contained, we review the motivation and formulation of the considered problem as well as the ideas sketched before. But beyond that, we provide for the first time (1) a detailed algorithmic description and implementation of the proposed methodology as well as (2) an evaluation of the obtained results.

978-1-4673-6908-4/15 © 2015 IEEE

MODELS 2015, Ottawa, ON, Canada
Foundations



```
context Speaker::setVolume(volume: Integer)
pre: volume >= 0 and volume <= 100
post: self.volume = volume

inv microOnWhenInCall: enabled = phone.callingapp.inCall

inv inboxAlert: hasNewMessages = MessagingApp.allInstances()
        ->exists(mApp | mApp.inbox->exists(m | m.read = false))
context Phone::topup(amount: Integer)
pre: credit >= 0
post: self.credit = self.credit@pre + amount

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

context MusicApp::playNextSong()
pre: currentSong < songs->size() - 1
post: currentSong = currentSong@pre + 1
post: phone.speaker.file = songs->at(currentSong)

context MessagingApp::sendMessage(msg: Message)
pre: enabled and phone.credit >= 5
post: phone.credit = phone.credit@pre - 5
context MessagingApp::emptyTrash()
pre: trash->size() > 0
post: trash->isEmpty()
post: Message.allInstances()->excludesAll(self.trash@pre)
context MessagingApp::deleteReadMessages()
post: trash = trash@pre->union(inbox@pre->select(m|m.read))

inv eitherInboxOrTrash: inbox = null or trash = null
```

Figure 1. UML/OCL model of a smartphone

A. Frame Problem

To demonstrate existing problems with declarative behavior definitions in UML/OCL and to have a baseline for presenting the proposed methodology, we use the model of a small, modular smartphone depicted in Fig. 1. The central feature of the class diagram is the `Phone`, having hardware attached to it (on the left-hand-side of the figure), namely a `Speaker` and a `Microphone`. Several applications using the hardware are defined: a `CallingApp` handles phone calls; a `MusicApp` handles played music; and a `MessagingApp` organizes messages. Invariants are in place to ensure valid structures.

A user interacts with the phone by invoking operations, whose behavior is defined by OCL pre- and postconditions. These OCL expressions follow the principle *design by contract* [2] and state constraints that have to be satisfied before the operation is called (preconditions) and after the operation has terminated (postconditions). However, they do not pose an implementation. This option is provided in UML/OCL in order to be able to define abstract model definitions. But this creates problems when trying to determine the effects of such definitions. In fact, pre- and postconditions are not deterministic and most likely allow for multiple possible post-states – many of which the modeler had not intended.

Example 1. *To give an example for multiple post-states, consider the initial system state of the running example shown at the top of Fig. 2. Calling the operation* `playNextSong()` *on the* `MusicApp` *object may, besides many others, result in either of the system states below the initial one. In the system state in the center of Fig. 2, the currently available speaker is now playing the next song – as it probably was intended. However, the postconditions also allow the system state at the bottom: here, the phone gets a completely new speaker that is playing the next song from an extended (!) songlist, but has a different volume set. Even worse, the postconditions of the operation also allow for changing the phone's current credit.*

In order to avoid problems like that, the behavior of operations has to be defined such that unintended changes are explicitly prohibited. This problem is generally called the *frame problem* [3] and is of particular interest for the verification of behavioral models. In this context, one has to know exactly which properties may change and which are unaffected during an operation call. Otherwise, unintended side effects, as illustrated in Ex. 1, make a meaningful verification impossible. For this purpose, dedicated constraints – so called *frame conditions* – are employed which explicitly describe *model properties* that may change. They provide information about (a) *class properties*, i.e., information regarding the existence of class instances which is used to handle object creation and destruction (OCL: `Class::allInstances()`), and (b) about *object properties*, i.e., defining for each object which attributes and links may change.

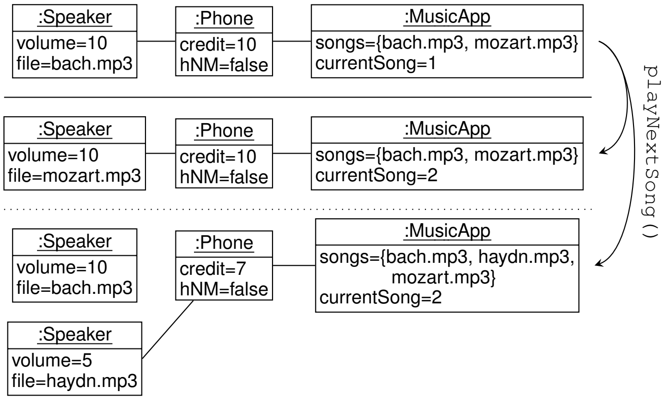


Figure 2. Valid execution scenarios for the operation `playNextSong()`

There are several approaches that address the frame problem by trying to infer the frame conditions from the model. One intuitive approach is to generally interpret the behavior of operations with the additional informal statement "*and nothing else changes*". Then, properties that are not mentioned in the postconditions are not meant to change. Recalling Example 1, the phone's current credit is not meant to change, because it is not mentioned in the postconditions. However, this still leaves the inaccuracies that are introduced by the OCL expressions themselves. Additionally, it is not clear how interdependencies to invariants are to be handled.

Example 2. *The operation* `deleteReadMessages()` *of the class* `MessagingApp` *moves all read messages from the* `inbox` *to the* `trash`*. To do so, the postcondition defines all read messages to be linked to the application via the* `trash` *property. Now, some of those messages might have been connected to the application via the* `inbox` *property, which the operation does not state to be removed. However, the operation is still sufficiently defined, because of the invariant* `eitherInboxOrTrash`*, which implicitly forces the removal of the inbox links. These interdependencies between postconditions and invariants are not at all intuitive and the example makes clear that the clause "and nothing else changes" is not suitable to clearly model the behavior of an operation.*

On the other hand, it is also infeasible to define all frame conditions manually. This would result in a complete definition of the operation's behavior – including explicitly stating all properties that may change and those that do not. This is obviously very time-consuming and not maintainable. Whenever the model is changed, all operations would have to be checked and adapted to the changes. Therefore, the solution to the frame problem is generally to define which properties an operation is supposed to change and to infer the frame conditions from the rest of the model. Thus, the modeler only has to decide which model properties are meant to change.

B. Related Work

The definition of the frame problem originates from *Artificial Intelligence* (AI) [4]. The term is used there to describe the problem of deducing the implications that follow a certain action an AI is executing. A typical example is the consideration whether the wall colors of a room change – given that an action is performed in it. Being in the scope of the *real world*, indefinitely many such considerations exist and the question is how to efficiently find the relevant ones[2]. In contrast, the advantage of abstract models such as UML/OCL models is the abstraction from the real world, heavily reducing the elements to be considered to a well-defined, finite subset.

In OCL there are several ways to describe frame conditions. Unlike some other declarative languages, OCL operation contracts currently do not have a special construct to list model properties that are allowed to change during the operation call. This leads to the situation that the frame conditions have to be entirely defined as postconditions. As discussed above, this is not feasible.

To overcome this issue, extensions for OCL have been proposed to support the modeler in explicitly defining the model properties that are affected by the operation. In [6], *invariability clauses* have been introduced for this purpose. Along the pre- and postconditions, these define which class and object properties may change during the operation call using the new keywords `modifies (only)`.

Example 3. *For the operation* `playNextSong()` *from Example 1, two invariability clauses have to be added in order to precisely define the intended changes of the operation, namely that* `currentSong` *is incremented and the attribute* `file` *of the speaker changes. To do so, the operation definition from Fig. 1 is extended with the following invariability clauses:*

```
in MusicApp modifies only: self::currentSong
in Speaker  modifies only: phone.speaker::file
```

The clauses consist of the optional `in` *part defining the context, followed by the keywords* `modifies only` *(the keyword* `only` *forbids object creation and destruction) and, finally, a list of* scope-terms *defining which properties are variable and for which set of objects, i.e., in which* scope.

In contrast, the authors in [7] introduced the OCL primitive `modifiedOnly()` – an OCL operation to restrict the set of variable model properties from within the postconditions.

Example 4. *To enhance the operation* `playNextSong()` *(c.f. Example 3) with the* `modifiedOnly()` *expression, an additional postcondition is added:*

```
post: Set{currentSong,
          phone.speaker.file}->modifiedOnly()
```

The `modifiedOnly()` *primitive has the following semantic: It evaluates to true when only properties from the specified set were found to be modified in the post-state of the operation.*

The usage of the operation `modifiedOnly()` is very similar to the invariability clauses. However, the incorporation into the postcondition poses the advantage that they can be used inside of conditional expressions. This allows for an even more precise definition than using invariability clauses at the operation level and may help for complex, conditional operations. However, interfering the appropriate frame conditions becomes much more elaborate in such cases.

Many other languages using declarative descriptions have built-in functionalities that allow to define the elements of the model that are meant to change. In the Eiffel language [8], the keyword `modifies` followed by a list of variables is used to declare modifiable properties. Z [9] uses the Ξ predicate for the same purpose and JML [10] has the annotation `@modifiable`. VDM [11] and CML [12] offer two keywords `rd` and `wr` to denote properties that may only be read and properties that may be written, respectively. Another solution is described in [13], where graph transformation rules are extended to allow for the definition of frame conditions.

In [3], the authors give an overview on some more of the existing solutions and reveal problems with them as well as propose a solution. The problems evolve around complex operations with conditionals and operation contracts with disjunctions, both of which result in potentially defining too many properties as modifiable. The proposed solution is to bind the definition of modifiable elements to conditions as well. This idea is already possible with the OCL primitive `modifiedOnly()` [7] discussed above.

C. Considered Problem

The related work reviewed in the previous section provides various description means that enable modelers to address the frame problem by precisely describing the behavior of

[2] A nice little anecdote and detailed definition of the frame problem in artificial intelligence can be found in [5].

operations. However, all these description means only provide a syntax and semantic for the definition of the frame conditions. The problem how to extract these conditions for a given model and all its operations remains open. To the best of our knowledge, existing solutions for that rely on simple "workarounds" only, i.e., methods based on:

- *Manual definition,* i.e., methods which completely rely on a manual refinement of the model like, e.g., the approach proposed in [14]. This leaves the burden solely on the modeler who requires the respective design understanding and, at the same time, is time-consuming. A noteable case study that clearly illustrates the weaknesses and limits of this strategy can be found in [15].
- *Implicit definition,* i.e., methods which simply apply naive schemes such as enforcing all model properties which are not restricted by originally provided constraints to remain unchanged (and, hence, ignore implicit relations and side-effects of the respective model properties). Approaches such as proposed in [16] are representatives for this.

In order to avoid the problems caused by these unsatisfactory solutions, improvements in the *extraction of frame conditions* are required, i.e., solutions which neither leave the burden of the extraction entirely on the modeler (avoiding the introduction of another time-consuming and expensive design step) nor are completely automatic (which, due to ambiguities and inaccuracies, will not lead to satisfactory results anyway). In this work, we are proposing such a solution.

III. Extraction of Hypotheses

In order to extract frame conditions for a particular operation, *all* properties of the model have to be considered. For each of them, it has to be determined which of these properties shall be variable and, if so, for which objects, i.e., for which *scope*. These questions may not always be answered by means of automatic methods, due to the fact that the models' semantics, especially the OCL constraints, often allow for several interpretations. Consequently, we propose an automatic model analysis to provide preliminary answers (called *hypotheses* in the following) by means of a distinction between the following categories:

- *Variable properties*, i.e., properties that are evidently meant to be modified by the respective operation since this modification is (precisely) constrained.
- *Unaffected properties*, i.e., properties that are evidently not meant to be affected by the respective operation. These properties should simply keep their current value.
- *Ambiguous properties*, i.e., properties where it remains unclear whether they are supposed to be modified by the operation or not. Hence, they have to be inspected more thoroughly by the modeler.

Already such a classification of the properties significantly aids the modeler in the definition of frame conditions. In fact, it filters out rather definite cases and pinpoints the modeler to a subset of properties which require a more detailed consideration.

In the remainder of this section, we describe how hypotheses based on this classification can be extracted automatically from a given model. To this end, a brief overview of the general methodology is sketched, before details on the respective implementations are provided. The quality of the hypotheses obtained by the proposed approach has been assessed by means of an evaluation whose results are later summarized in Section IV.

A. General Methodology

As arbitrary changes are to be allowed for properties that are defined to be in an operations' frame of change, the classification *variable* is only to be assigned when there is strong evidence for it and precise information about the affected objects is known. Following the "*and nothing else changes*" scheme (cf. Section II-A), we, thus, restrict our attention to properties that are mentioned within the contractual scope of the current operation and exclude properties from our consideration which are not at all involved in the respective operation. For this purpose, we focus on the three sources that define the scope of operations in contractual designs:

- *Postconditions* are the main source as they constrain the post-state and, therefore, describe the changes performed during an operation call in a direct manner.
- *Invariants* define further requirements that have to be satisfied at all times, in particular also in the post-state of the operation in addition to postconditions. To this end, they may establish implicit dependencies to further properties which are relevant for the frame conditions as they require additional modifications in order to reach a valid system state.
- *Preconditions* provide assured information about the pre-state of an operation call, i.e., precise values or properties that can be useful to clarify whether properties keep or change their value during the operation call.

While focusing on these constraints already gives a good approximation of the set of possibly relevant properties and immediately excludes properties for which there is no clear evidence for a relation to the operation, definitive decisions on the properties' variability cannot always be provided automatically. Instead we apply a heuristic approach to derive hypotheses from the respective OCL constraints, i.e. tuples composed of a property, a scope-term (describing the set of objects on which the hypothesis applies), and a preliminary classification (variable, ambiguous, or unaffected).

Example 5. *Consider the operation `setVolume()` of the `Speaker` class from our running example (cf. Fig. 1). Here, we extract the hypothesis that setting the volume does not affect which music is currently played (`Speaker::file`) on the same speaker (`self`). This hypothesis is represented by the tuple (`Speaker::file`, `self`, unaffected).*

To extract these hypotheses, we apply the following flow:

1) Extract a basis set of hypotheses from the operation's postconditions.
2) Scan the model's invariants for *implicit dependencies* on further properties that might be affected and add them as corresponding hypotheses. Repeat this step until no further dependencies are found.
3) Try to use semantic information from the preconditions to clarify ambiguities.

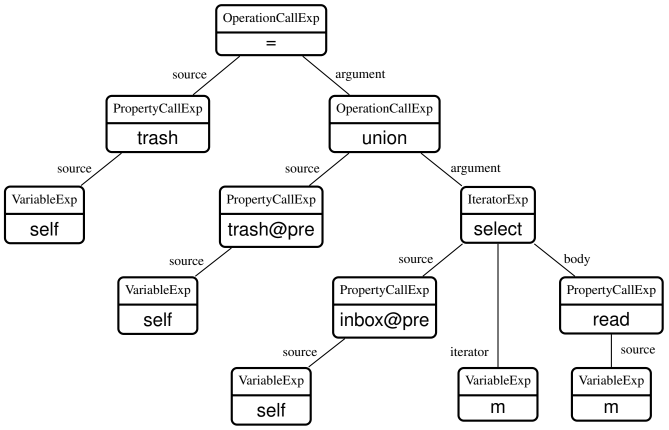


Figure 3. AST for the postcondition of `deleteReadMessages()`

To this end, we make use of *Abstract Syntax Tree* (AST) representations of the constraints in order to found our approach on a rigorous and unique basis. These ASTs are established using the rigorousness of the meta-model for OCL expressions (defined in [17, Chapter 8.3]) and can be easily obtained from standard OCL parsers. The AST is a unique representation of an OCL constraint and already provides a rough semantic profile that will be used for extracting hypotheses.

Example 6. *Consider the AST in Fig. 3 which represents the postcondition of the operation `deleteReadMessages()` of the `MessagingApp` class from our running example (see Fig. 1). Here, each node and edge is labelled by the corresponding expression type and relation name, respectively. Note that the references to* self *are not explicitly provided in the postcondition, but are added automatically by the parser. The most important (expression) types for our purpose are `PropertyCallExp`, which refers to an actual property of the model, and `OperationCallExp`, `IteratorExp`, etc., which provide information about the dependency of these properties. All of these expressions have a `source` that refines their scope. `OperationCallExps`, `IteratorExp`, etc. may additionally have a parameter (called `argument`, `body` or alike).*

In the following subsections we will describe in detail how the hypotheses are extracted from the operations' contractual scope, i.e. from postconditions and invariants, and how ambiguities may be clarified using the semantic information from the preconditions.

B. Extraction from Postconditions

The primary source for the extraction of hypotheses are the postconditions of the considered operation. Postconditions constrain the changes performed during an operation call; most properties which are supposed to be variable can be assumed to be referenced therein. However, as discussed in Example 1, those constraints are not always unambiguous.

Addressing this issue, standard interpretations for ambiguous OCL constructs have been suggested in [18] 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. This leads to the derivation of the following three principles for extracting hypotheses from postconditions:

1) *All properties referenced within the postconditions* may *– but are not necessarily meant to – be affected.*

This main principle is an adapted and relaxed form of the *nothing else changes* heuristics described in [18] and is also applied by "implicit definition" approaches reviewed in Section II-C where – much more strictly – all referenced properties are automatically considered variable. This principle requires that a separate hypothesis is extracted for each property and each scope-term in which it occurs. Different classifications may result from multiple references of the same property and will eventually have to be combined to a single hypothesis.

Example 7. *Consider again the postcondition from Fig. 3. There are two references to the property `trash`. One of these will lead to a variable, the other to an ambiguous classification (details on this will be presented later in Example 8). But as both references have the same scope `self`, a single hypothesis will be extracted from both with the more definite classification variable.*

2) *In general, properties used as an operation parameter are not meant to be variable.*

To illustrate this principle, consider again the postcondition of `playNextSong()` (cf. Example 1):

```
phone.speaker.file = songs->at(currentSong)
```

Here, the argument of the `at`-operation, i.e., `currentSong`, is not meant to be variable. Note, however, that there are a few exceptions from this principle, e.g. arithmetic and logical operations, where the operands can be used interchangeably.

3) *Properties that are referenced in postconditions are meant to be variable if and only if*
   a) *they are referenced within a sub-term that allows for this categorization (cf. Principle 2),*
   b) *they are the last element of a navigation chain, and*
   c) *they refer to the post-state of the operation (i.e. are* not *marked `@pre`).*

The third principle refines under which circumstances the definite classification variable is eventually applied to a property–taking into account that the change of properties can only be constrained properly if they are referenced from an appropriate context. Following this concept, a reference to the pre-state ("`@pre`") is not sufficient to constrain a change. The same is true for front and inner elements of navigation chains, e.g. in `phone.speaker.file` only `file` is considered variable since a change of `phone` or `phone.speaker` cannot be constrained properly here.

These main principles eventually lead to Algorithm 1 which realizes the extraction of hypotheses. The technical details of this algorithm are now discussed in detail and will afterwards be illustrated by means of Example 8.

To extract hypotheses from a postcondition using Algorithm 1, the corresponding AST is traversed and the following case distinction with respect to the node's type is conducted:

Leaves of the AST (Lines 1–2): No hypotheses are created for these nodes, since they are either of type `LiteralExp`

```
   Input: AST node, context information
   Output: Set of Hypotheses
1  if node is a leaf then
2      result ← empty set of hypotheses
3  else if type of node is PropertyCallExp then
4      scope ← determineScope(source)
5      sourceHyps ← extract hypotheses from source
6      update hypotheses in sourceHyps to ambiguous
7      propertyHyp ← new Hypothesis (property, scope)
8      result ← union of sourceHyps and propertyHyp
9  else // OperationCallExp or alike
10     update context information
11     mergeRules ← lookForRulesAndPatterns(context)
12     mergedHyps ← ∅
13     foreach sub-tree s (source / argument / ...) do
14         subHyps ← extract hypotheses from s with
           respect to context
15         merge subHyps into mergedHyps using the
           mergeRules
16     end
17     result ← mergedHyps
18 end
```

Algorithm 1: Extract hypotheses from OCL expression

(constant literal) or `VariableExp` (referring to a set of objects). However, the latter are used to determine scopes for

References to properties (Lines 3–8): Here, first the property's scope is computed from the `source` sub-tree (Line 4). Note that hypotheses extracted from this sub-tree are downgraded to ambiguous (according to Principle 3b) because this part corresponds to the front part of the navigation chain which points to the property (Lines 5–6). According to Principle 1, a separate hypothesis is created for each reference to a property (Line 7)[3]. Each property is provisionally classified variable here – except for the case that it is marked `@pre` and Principle 3c can immediately be applied[4]. Consequently, to eventually obtain the appropriate classification, a reclassification is performed when processing

Other (inner) nodes of the AST (Lines 9–18): In these nodes, hypotheses resulting from different sub-trees are merged (Lines 13–16) – the most crucial part of the algorithm. In the simplest case, the hypotheses from the sub-trees are identical. Then, they can simply be adopted "as is" for the currently considered node. Otherwise, the classification may be downgraded depending on the sub-trees they originate from. While doing that, competing hypotheses for the same property and scope (as illustrated in Example 7) may occur. Then, the hypotheses are merged based on a set of *static rules* (which merge the hypotheses based on the expression type and operation name) as well as *design patterns* (which merge the hypotheses based on the specific context). More precisely:

- A selection of representative static rules are summarized in Table I. Here, the first column denotes the respective expression type according to the metamodel. In the second column, relevant components corresponding to sub-trees within the AST are listed. Details on the classification are provided in the third column, while the last column provides which scope-terms are to be extracted. A special case are disjunctions (e.g., `or` and `implies`): here a variable hypothesis is only adopted if it occurs in both sub-trees, otherwise it is adopted as ambiguous and additionally marked (as this may indicate an incomplete definition).
- As an example of a design pattern, consider the following issue: According to the static rule, the left- and right-hand side of an equality sign are treated interchangeably by default. However, if one side is a reference to a property in the post-state (PropertyCallExp), then the "=" is interpreted as an assignment operator ("= means assignment") and hypotheses from the other side are downgraded to ambiguous. Another special pattern is `property=property@pre`, i.e., an explicitly specified frame condition which leads to the classification unaffected. Moreover, in order to capture instantiation and destruction of objects, we keep track of a few key patterns, namely several operations called on `Class.allInstances()` (e.g. `includes()`, `excludes()`, and `size()`) and the OCL primitive `Object.oclIsNew()`. If these are recognized (Line 10), a special flag is activated that allows for adding a corresponding hypothesis on the class property `allInstances()`.

In a similar fashion, other cases are handled.

This algorithm is now demonstrated by means an example.

Example 8. *Consider again the AST in Fig. 3 which represents the postcondition of the operation `deleteReadMessages()` (cf. Example 6). Starting with the root node (=), first the design pattern "= means assignment" is recognized (Line 11), since the `source` is a reference to a property in the post-state. Consequently, all hypotheses from the `argument` sub-tree will later be downgraded. However, following the flow of the algorithm, the next step is to process the `source` sub-tree (Line 13–16). Here, an empty set of hypotheses is extracted from the leaf `self` (Lines 2/5) and a new hypothesis for `trash` is created: (`MessagingApp::trash`, `self`, variable) (Line 7). From the `argument` sub-tree of the root node, the properties `trash` and `inbox` are classified ambiguous, since they are marked `@pre`. The property `read` with scope-term `self.inbox@pre` is classified unaffected as it occurs within a `select` (static rule). When merging the hypotheses from both sub-trees at the root node level (Line 15), the competing hypotheses for `self::trash` (`source`: variable, `argument`: ambiguous) are merged to a single variable hypothesis (cf. Example 7). Due to the design pattern recognized at the very beginning, the hypotheses for `inbox` (only referenced in the pre-state) and `read` are to be downgraded to ambiguous. However, they are eventually adopted unchanged as they already are ambiguous or unaffected, respectively.*

[3] Note that this (Line 7) is indeed the only place where new hypotheses are created. At all other occasions, hypotheses are only updated or merged, as in Line 8 and later in Line 15.

[4] If only the pre-state or both the pre- and post-state of a property is referenced, it is indicated that changes to this property are intended, though, in the first case, the modifications are not explicitly defined or constrained. Consequently, we log the states in which a property is referenced.

Table I
CLASSIFICATION SCHEME FOR EXTRACTING HYPOTHESES FROM OCL EXPRESSIONS

Expression type	Components	Classification	Scope-terms
PropertyCallExp	source	Classify variable only in accordance to Principle 3	source::referredProperty
OperationCallExp	source, parameter	Adopt classifications from source "as is" except for the following cases: If the referred operation is arithmetic (e.g. +,−,*,/), then downgrade the classification to ambiguous. For the parameter sub-tree (if applicable), downgrade all classifications, except for logical operations (e.g. and, or, implies).	
IteratorExp	source, body, iterator	Adopt classifications from body "as is" if the iterator returns a Boolean (e.g., exists, forAll, one), but not if the iterator returns a collection (e.g., select, closure, any).	
IfExp	condition, thenExpression, elseExpression	Process subexpressions and lift their classifications (variable supersedes ambiguous supersedes unaffected)	If a property is only referenced in either of then- and else-branch, use the (negated) condition for the scope-term
VariableExp		Ignore, if contains parameter, result value, or iterator variables; else use for scope-term	self::property, Class.allInstances()::property
LiteralExp		Ignored, since expression only denotes a constant	–

Finally, the following preliminary hypotheses are extracted:

- *(MessagingApp::trash, self, variable),*
- *(MessagingApp::inbox, self, ambiguous),*
- *(Message::read, self.inbox@pre, unaffected).*

Overall, following the outlined principles and schemes, an initial, preliminary classification of properties with respect to the three categories (i.e., variable, ambiguous, and unaffected) is *automatically* conducted. In the next step, we will investigate whether there are further relevant properties that have not been classified so far, but to which a dependency exists via the models' invariants.

C. Extraction from Invariants

The classification of properties based on their occurrence in postconditions as described above is a promising starting point. However, there may also be properties which do not occur in the operation's postconditions, but are connected to other properties that were identified to be relevant for the operation (variable or ambiguous) in a different way.

Example 9. *Consider the constraint that the microphone shall be enabled during phone calls (expressed by the invariant microOnWhenInCall of the Microphone class in Fig. 1). Assume that the attribute inCall has been classified variable. 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. The finally extracted hypothesis reads (Microphone::enabled, self.phone.microphone, variable).*

Clearly, these dependencies have to be considered in order to obtain a complete set of properties for the frame conditions. Note that a direct connection between the corresponding classes via associations as in Example 9 is actually not required; dependencies can rather also be established with an *allInstances()*-call as illustrated in the following example.

Example 10. *The smartphone has an alert for new messages that is triggered if one of the messaging apps has new messages. This functionality is expressed by the invariant inboxAlert (associated to the Phone class in Fig. 1). Consequently, if the inbox of any MessagingApp or the read attribute of any Message is allowed to change, then also hasNewMessages is affected. Strictly speaking, this is true for all instances of Phone since the dependency is established via allInstances()-calls. This is expressed by the hypothesis (Phone::hasNewMessages, Phone.allInstances(), ambiguous).*

Hence, in order to reveal all such *implicit dependencies*, we perform a dependency analysis for the properties that have been determined to be relevant (variable or at least ambiguous) thus far. For a complete analysis, we have to consider all invariants of *all* classes. More precisely, the invariants are covered in a similar fashion like postconditions, i.e. with a slightly revised version of Algorithm 1. However, the gained hypotheses are not directly merged with the existing ones, but are used to discover dependencies to properties that were not considered relevant thus far. If such a property is referenced together with a variable or ambiguous property within the same invariant (more precisely: within the same branch with respect to disjunctions), there are two types of connections that lead to different classification results:

- if the property is classified variable within the invariant and at least one other variable property is referenced, a hypothesis with classification variable is added.
- if the property itself is only classified ambiguous within the invariant or no other property is referenced in a scope for which a hypothesis with classification variable exists, a hypothesis with classification ambiguous is added.

This analysis is applied iteratively until no more implicit dependencies are found. Overall, this gives a complete subset of properties for which there is (strong) evidence that they are relevant for the operation's frame conditions. Clearly, the invariant analysis may produce hypotheses that will be discarded later, but it reveals dependencies some of which are likely to be missed when generating frame conditions manually.

D. Disambiguation using Preconditions

The analysis of postconditions and invariants which has been performed so far, provided a set of hypotheses on properties that are likely to be relevant for the operation's frame conditions. In distinct cases, semantic information from the preconditions can be used to clarify ambiguous hypotheses.

Example 11. *Consider the operation* `emptyTrash()` *of class* `MessagingApp` *from Fig. 1. The postcondition requires that the trash is empty after the execution of the operation (*`post: trash.isEmpty()`*). In contrast, the precondition requires that the trash contains at least one message in the pre-state (*`pre: trash.size()>0`*). Consequently, the property* `trash` *is definitely meant to be variable and, if it was previously determined to be unaffected or ambiguous, the corresponding hypothesis has to be updated accordingly.*

However, in order to directly incorporate this information and to update hypotheses automatically, we would require a methodology to interpret the semantics of the pre- and postconditions with respect to aspects of properties like being undefined, being empty, ranges of values, etc. at an abstract level, i.e. without having any concrete instantiation of the model. As such a methodology is far out of reach, we rather again apply a slightly modified version of Algorithm 1 to the preconditions. However, the hypotheses gained from this are only used to check whether a property occurs in the precondition for which an ambiguous or unaffected hypothesis exists for the same scope. If so, the hypothesis is marked accordingly such that the modeler can take this knowledge into account when considering the ambiguous hypotheses. However, the preconditions are rather secondary, although they are not entirely insignificant, and the key parts of our approach are clearly the extraction of hypotheses from postconditions and invariants.

IV. Evaluation

In the previous section, we proposed a methodology to extract hypotheses about the relevance of properties in order to aid designers in the generation of frame conditions for operations. To this end, we applied a heuristic approach that analyzes postconditions, invariants, and preconditions. In order to evaluate the quality of the obtained hypotheses, the proposed approach has been tested using several example models known from the literature. The results are summarized and discussed in this section.

A. Setup and Considered Examples

In order to perform the evaluation, we implemented our approach as a prototype using the Eclipse OCL framework and the Xtend language. We then applied our approach to several test models known from the literature – some of which have been explicitly constructed with respect to the frame problem and are, thus, relevant and representative examples. More precisely, the following models have been considered: (1) the running example from this paper (i.e., the *Smartphone* model from Fig. 1), (2) a model representing a toll collecting system (*TollCollect*, [19]), (3) a model representing a scheduling process, e.g., of a CPU (*Scheduler,* [20]), (4) a model representing the process of lending books (*Library*, [21]), (5) a model representing the civil status of human beings (*Civil Status*, [22]), (6) a model representing a traffic light preemption (*TrafficLight*, [23]), and (7) a model representing an access system at an (*Airport_CML*, [24]).

More details about the models as well complete lists of the extracted hypotheses have been compiled into a separate document which is available online for the reader's evaluation [25].

B. Obtained Results

Our results are summarized in Table II. For each model, we first provide statistical information of the models (size, number of constraints, number of properties). Then, for each operation of the model, details on the extracted hypotheses are summarized. The first column contains the name of the operation (denoted by *Operation*). In the second column, the total number of extracted hypotheses is provided. If a single number is shown here, hypotheses were only extracted from the operation's postconditions, i.e., no implicit dependencies to further properties were determined. If a sum $n + m$ is shown, then n hypotheses resulted from the analysis of the postconditions and m from implicit dependencies established by the invariants. In the remaining columns, the hypotheses are separated by their classification (variable, unaffected, and ambiguous). Whenever the algorithm provided a definite classification (i.e., variable or unaffected), we manually evaluated whether this classification was in accordance with our interpretation of the models (denoted by ✓) or not, i.e. whether the classification was false-positive or false-negative (denoted by f_p and f_n, respectively). Note that, whenever the proposed algorithm missed an implicit dependency, we interpreted this as a false-negative classification (although no hypothesis was actually extracted). In such cases, the number of missed dependencies is shown in parentheses.

C. Discussion

In order to assess the quality of the proposed approach, we addressed the following questions:

1) Does the algorithm extract wrong definite classifications and, if so, how often and why?
2) Does the algorithm extract all implicit dependencies and, if not, which are missed?
3) How many properties cannot be classified as variable or unaffected and, hence, have to be manually inspected by the modeler? What are the specific reasons for the ambiguous classification and which lessons can be learned?

In the following we will discuss the results in more detail with respect to these questions.

ad 1) First of all, it can be observed that no *false-positive* classifications are derived, i.e. no property is classified variable though it is actually intended to be unaffected. Consequently, definite variable classifications obtained by the algorithm seem to be safe and reliable. The same is true for definite unaffected classifications, since we do not observe *false-negative* classifications, i.e. possibly relevant properties which are not recognized as such, from actually extracted hypotheses. However, most properties are implicitly classified unaffected due to the restriction to the contractual scope of the operation; properties that are explicitly classified unaffected within actually extracted hypotheses only occur in rare cases: Besides the case discussed in Example 8, `MessagingApp::deleteReadMessages()`, only

Table II
Results of the Evaluation

Operation	#Hypotheses	#Variable		#Unaffected		#Ambiguous
		✓	f_p	✓	f_n	
Smartphone (running example), 5 Classes, 10 Attributes, 6 Associations, 9 Operations, 3 Invariants						
Phone::topup()	1	1				
Speaker::setVolume()	1	1				
CallingApp::placeCall()	1+1	2				
CallingApp::talk()	2	1				1
CallingApp::closeCall()	1+1	2				
MusicApp::playNextSong()	5	2				3
MessagingApp::sendMessage()	2	1				1
MessagingApp::deleteReadMessages()	3+1	1		1		2
MessagingApp::emptyTrash()	2	2				
TollCollect [19], 2 Classes, 4 Attributes, 2 Associations, 8 Operations, 3 Invariants						
Truck::init()	2+1	2				1
Truck::enter()	2	2				
Truck::move()	3	3				
Truck::pay()	1	1				
Truck::bye()	2	1				1
Point::init()	1+1	1				1
Point::northConnect()	1	1				
Point::southConnect()	1	1				
Scheduler [20], 2 Classes, 1 Attribute, 3 Associations, 4 Operations, 1 Invariant						
Scheduler::Init()	3	3				
Scheduler::New()	3	1		2		
Scheduler::Ready()	3	3				
Scheduler::Swap()	4	3				1
Library [21], 3 Classes, 7 Attributes, 2 Associations, 7 Operations, 10 Invariants						
User::init()	2+1	2				1
User::borrow()	1	1			(1)	
User::return()	2	2				
Copy::init()	3+1	3				1
Copy::borrow()	1	1			(1)	
Copy::return()	2	2				
Book::init()	3+1	3				1
Civil Status [22], 1 Class, 4 Attributes, 1 Association, 4 Operations, 5 Invariants						
Person::birth()	4+1	4			(2)	1
Person::marry()	6	1				5
Person::divorce()	6	1				5
Person::death()	8	1				7
TrafficLight [23], 1 Class, 3 Attributes, 0 Associations, 2 Operations, 1 Invariant						
TrafficLight::init()	3	3				
TrafficLight::switch()	2+1	2				1
Airport_CML [24], 2 Classes, 2 Attributes, 0 Associations, 4 Operations, 1 Invariant						
Airport::Init()	2	2				
Airport::GivePermission()	1+1	1				1
Airport::RecordLanding()	1+1	1				1
Airport::RecordTakeOff()	2	2				

in one more case, namely `Scheduler::new()`, a property mentioned in the postconditions is automatically classified unaffected. In the latter model, frame conditions are explicitly modeled via terms like `property=property@pre` in the postconditions. This pattern is recognized and leads to this definite classification.

ad 2): In three cases, the algorithm misses implicit dependencies. This behaviour can be explained by the restrictive strategy that is applied during the invariant scan and which discards possible dependencies if the properties are spread over different branches of a disjunction. Using a more liberal strategy could avoid this, but will likely produce more unnecessary ambiguities and make the determination of scope-terms infeasible. However, the actually missed dependencies are not severe and do not concern properties that are in fact relevant. In all cases, still all relevant properties are identified and corresponding hypotheses are extracted.

ad 3): In general, the number of *ambiguous* classifications is rather small, except for the *Civil Status* and *Smartphone* example.

- In the *Civil Status* example, there is a large amount of case distinctions (males marry females and vice versa) which are expressed by constructs such as
  `gender = #female implies ...`
  `gender = #male implies ...`
  As the algorithm treats these implications like usual disjunctions, many ambiguities could have been avoided by using `if .. then .. else .. endif` constructs.
- For the *Smartphone*, the large number of ambiguities can be explained with the fact that this model was explicitly constructed to demonstrate ambiguities and contains many navigation chains that are treated as ambiguous by the algorithm.
- The ambiguities in the *TollCollect* and *Library* models result from invariants that express uniqueness constraints. They constrain a certain property as a *key property*,

i.e., with pairwise different values for different objects. One ambiguity in `Truck::bye()` results from a design pattern that recognizes return values of an operation.

- In the *Scheduler* model, frame conditions have been defined explicitly. However, one case has been missed and is only covered implicitly in the `Scheduler::Swap()` operation: it is not determinately defined which process from the ready queue is indeed activated, if the queue is not empty. As this leads to different classifications in the `then`- and `else`-part, an ambiguous hypothesis is finally extracted.
- In the *TrafficLight* example, one signal is not defined explicitly in the postconditions, but its correct value is enforced by an invariant. The algorithm finds this dependency, but as the invariant is constructed as a large disjunction of conjunctions (sum of products), it does not provide a definite classification here.
- Finally, in the *Airport_CML* model an invariant establishes a subset-dependency between two sets of aircrafts which results in an ambiguous hypothesis in two cases. In one of these, knowledge from the preconditions can be used to clarify the ambiguity.

Overall, these results clearly confirm the potential of the proposed algorithm and the underlying methodology. In almost all cases, very powerful hypotheses can be extracted out of which only a tiny fraction are false-positives/false-negatives. Considering that, thus far, modelers are supposed to generate *all* frame conditions manually, the proposed methodology boils down these efforts to a brief consideration of hypotheses out of which just a few ambiguous properties remain to be inspected in detail.

V. Conclusions

In this work, we considered the frame problem for behavioral UML/OCL models. Thus far, only workarounds for the definition of corresponding frame conditions exist: they are either to be performed entirely manually – a time-consuming task – or they are implicitly derived from the postconditions using simple heuristics like "and nothing else changes" which often does not lead to satisfactory results. To this end, we proposed an automatic approach that considers the whole contractual scope of the operations, i.e., postconditions, invariants, and preconditions, and extracts hypotheses addressing the question which properties are to be considered for the frame conditions.

An evaluation confirmed that the methodology provides very good results. In fact, it safely filters out rather definite cases and pinpoints the modeler to a small subset of properties which require a more detailed inspection. Many of these can be decided on after a brief consideration. For the remaining properties, the modeler can also be aided by automated methods, e.g., consistency checkers. These can be employed to generate execution scenarios with this particular property being variable or unaffected. These validations may even pinpoint the modeler to possible inconsistencies in the design. For the first time, this provides a systematic approach on how to deal with the frame problem in behavioral models.

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] P. Niemann, F. Hilken, M. Gogolla, and R. Wille, "Assisted generation of frame conditions for formal models," in *DATE*, 2015, pp. 309–312.

[2] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[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] Z. W. Pylyshyn, "The Robot's Dilemma: The Frame Problem in Artificial Intelligence," *Artif. Intell.*, vol. 36, no. 1, pp. 131–137, 1988.

[5] D. C. Dennett, "Cognitive Wheels: The Frame Problem of AI," *Language and Thought*, vol. 3, 2005.

[6] 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.

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

[8] B. Meyer, "Eiffel: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.

[9] J. M. Spivey and J. Abrial, *The Z Notation: A Reference Manual, 2nd Edition*. Prentice Hall Hemel Hempstead, 1992.

[10] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*. Springer, 1999, pp. 175–188.

[11] C. B. Jones, *Systematic Software Development using VDM*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[12] J. Woodcock, A. Cavalcanti, J. S. Fitzgerald, P. G. Larsen, A. Miyazawa, and S. Perry, "Features of CML: A formal modelling language for Systems of Systems," in *7th International Conference on System of Systems Engineering*. IEEE, 2012, pp. 445–450.

[13] T. Baar, "OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem," in *Satellite Events at the MoDELS 2005 Conference*, ser. LNCS, J. Bruel, Ed., vol. 3844. Springer, 2005, pp. 20–31.

[14] 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.

[15] 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.

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

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

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

[19] M. Gogolla, L. Hamann, F. Hilken, M. Sedlmeier, and Q. D. Nguyen, "Behavior Modeling with Interaction Diagrams in a UML and OCL Tool," in *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, ser. BM-FA '14. ACM, 2014, pp. 4:1–4:12.

[20] P. A. P. Salas and B. K. Aichernig, "Automatic Test Case Generation for OCL: a Mutation Approach," *UNU-IIST Report*, no. 321, 2005.

[21] M. Gogolla, "Teaching Touchy Transformations," in *MODELS Educators' Symposium (EDUSYMP'2008)*, M. Smialek, Ed., 2008, pp. 13–25.

[22] 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.

[23] "USE Examples Repository." [Online]. Available: http://sourceforge.net/projects/useocl/

[24] Q. Charatan and A. Kans, *Formal Software Development: From VDM to Java*. Palgrave Macmillan, 2004. [Online]. Available: http://symphonytool.org/examples/Airport/index.html

[25] P. Niemann, F. Hilken, M. Gogolla, and R. Wille, "Extracting Frame Conditions from Operation Contracts: Additional Material," University of Bremen, Tech. Rep., 2015. [Online]. Available: http://www.informatik.uni-bremen.de/agra/doc/misc/NHGW2015addon.pdf