Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 189

Automatically Binding Variables of Invariants to Violating Elements in an OCL-Aligned XBase-Language

Sebastian Fiss[1], Max E. Kramer[1], Michael Langhammer[1]

Abstract: Constraints that have to hold for all models of a modeling language are often specified as invariants using the Object Constraint Language (OCL). If violations of such invariants shall be documented or resolved in a software system, the exact model elements that violate these conditions have to be computed. OCL validation engines provide, however, only a single context element at which a check for a violated invariant originated.Therefore, the computation of elements that caused an invariant violation is often specified in addition to the invariant declaration with redundant information. These redundancies can make it hard to develop and maintain systems that document or resolve invariant violations.
In this paper, we present an automated approach and tool for declaring and binding parameters of invariants to violating elements based on boolean invariant expressions that are similar to OCL invariants. The tool computes a transformed invariant that returns violating elements for each iterator variable of the invariant expression that matches an explicitly declared invariant parameter. The approach can be used for OCL invariants and all models of languages conforming to the Meta-Object Facility (MOF) standard. We have evaluated our invariant language and transformation tool by transforming 88 invariants of the Unified Modeling Language (UML).

Keywords: OCL, Xbase, XOCL4Inv

1 Introduction

When models are used to develop software systems, a metamodel can be used as a language specification that constrains all valid model instances. Not all types of constraints that have to be enforced can directly be expressed in a metamodel. For such constraints, specification languages can be used in addition to the metamodeling language. If a constraint has to hold for all models of a language, it is usually called an invariant. The Object Constraint Language (OCL) defined in the ISO 19507 standard is a popular language for defining such invariants for object-oriented software . OCL invariants are mainly used to validate model instances in order to ensure correctness prior to further processing or manipulation.

For many application scenarios, it is not sufficient to know whether an invariant holds for a given model: In order to document or resolve the problem indicated by an invariant violation, it is important to obtain the context of a violation [KPP09].In OCL, an invariant is, however, only a boolean constraint expression that may specify a name and a context type. Therefore, OCL-based validation engines provide only a single context element of this type to indicate an invariant violation. Many OCL invariants navigate and inspect several

[1] Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organization (IPD), Chair for Software Design and Quality (SDQ), Am Fasanengarten 5, 76131 Karlsruhe, Germany, sebastian.fiss@student.kit.edu, max.e.kramer@kit.edu, michael.langhammer@kit.edu

different elements and collections of elements related to the context element at which a check was initiated. Therefore, the context element of an OCL invariant often does not directly indicate where, how, and why a model violates the constraints. To achieve this, more specific elements that cause an invariant violation, e.g. by not satisfying one of several conditions defined in the invariant, have to be retrieved.

The retrieval of such model elements that cause an invariant violation is usually defined separately from the boolean invariant condition. As a result, model navigation statements and condition checks are repeated in the code for element retrieval and constraint validation. Although only a few statements may be redundant for a single invariant, the amount of duplicated code can grow to a considerable size for metamodels with hundreds of invariants, such as the Unified Modeling Language (UML) [ISO12a]. This code duplication can be a source for costly errors and can lead to unnecessary development and maintenance effort. It is common to all current approaches except EMF-IncQuery, which only support queries.

In this paper, we present an approach and tool[2]. to avoid this code duplication by computing elements that cause an invariant violation directly from an OCL-aligned invariant definition with explicit parameters. We propose a prototypical invariant language XOCL4Inv, which extends the expression language Xbase [Eff+12] and seamlessly integrates with the popular model transformation language Xtend[3]. As a result, our language is extensible and inherits the power and expressiveness of Xbase, e.g. lambda expressions and extension methods.

The language supports the definition of OCL-aligned constraints using boolean expressions and is in large parts syntactically and semantically equivalent to a subset of OCL. In order to relieve developers from writing separate code for element retrieval, it adds a possibility to define which elements should be retrieved in case of an invariant violation: Variables that are used to iterate over collections can be declared as invariant parameters, which is not possible in OCL. For every declared parameter, our tool computes a transformed invariant, which collects all those elements that are a) bound to the iterator variable corresponding to the parameter and b) causing the invariant violation. It is, however, also possible to directly specify queries, if this is preferred to invariant-to-query transformations.

Consider a simplified example invariant for a library metamodel, which specifies in the context of a reading room that all those books in a reading room that are used as reference copies have to have at least three copies:

```
self.books.forall[Book b | b.referenceCopy implies (b.copies >= 3)]
```

Our tool computes a transformed invariant, which collects all books that violate the constraint, i.e. that are used as reference copies but have less than three copies. It replaces the `forall` iterator with a `select` iterator and negates the condition:

```
self.books.select[Book b | !(b.referenceCopy implies (b.copies >= 3))]
```

For more complex invariants, e.g. with more iterators and parameters, it would be a waste of time to specify such transformed invariants manually as they can be computed automatically.

The presented invariant transformation algorithm can be used for OCL invariants that are defined for models conforming to the Meta-Object Facility (MOF) ISO/IEC 19508:2014(E)

[2] The language and tool are available as open-source software on http://sdqweb.ipd.kit.edu/wiki/XOCL4Inv
[3] http://www.eclipse.org/xtend

standard. Our current invariant language XOCL4Inv and transformation prototype is based on the Eclipse Modeling Framework (EMF) and can transform invariants for metamodels that were built with the Essential MOF variant Ecore. We evaluated the correctness by transforming all 88 invariants of the UML metamodel of the Eclipse IDE[4] that contain collection iterators but do not use statements that cannot be transformed, such as `allInstances`.

The paper is structured as follows: In Section 2, we explain concepts and languages that are fundamental for our approach. In Section 3, we present our OCL-aligned invariant language XOCL4Inv. In Section 4, we explain the invariant transformation algorithm. In Section 5, we present our evaluation of the invariant language and transformation algorithm. In Section 6, we discuss related work and in Section 7 we draw some final conclusions.

2 Foundations

In this section, we explain the technologies and concepts that we use for our approach.

2.1 Model-Driven Software Development (MDSD)

In MDSD [SV06] models and code are used to develop a software system. An important point is that models are at least as important as the source code and not used for documentation purposes only. A common use case is to automatically create source code from the information in the models. The models are often created by domain experts to model a specific domain. To apply these models, Stahl et al. have described three requirements: First, DSLs are necessary to create the models. Second, model-to-code transformations languages are required to process them. Last, specific compilers, generators or transformers are necessary to create executable code from the models.

2.2 Object Constraint Language (OCL)

OCL [ISO12b] is a typed, declarative language that can be used to describe constraints that apply to model instances. OCL was initially developed for UML models, but can be used for arbitrary metamodels. Constraints that are specified with OCL are side-effect free. This means that the queried model instance is not changed by OCL. In MDSD, OCL is used to define constraints and invariants that cannot be easily expressed in a metamodel. Users of OCL can check whether a specific model instance fulfills the constraints that are defined for the metamodel. If one of these constraints is not fulfilled, the model instance is not valid.

2.3 Xtext and Xbase

Xtext [EV06] is a language development framework for creating DSLs. Users have to define the grammar of their DSL. From the grammar definition Xtext creates the lexer, parser and

[4] Eclipse UML metamodel – Rev. 57c76de64a8925e897c2a2ef0a898ea6c153816d – 2014-12-14 http://git.eclipse.org/c/uml2/org.eclipse.uml2.git/tree/plugins/org.eclipse.uml2.uml/model/UML.ecore

```
1 context ReadingRoom
2 invariant AtLeast3ReferenceCopies (Book b)
3 check self.books.forall[Book b | b.referenceCopy implies (b.copies >= 3)]
```

Listing 1: XOCL4Inv invariant definition with a simplified constraint for a library

the semantic analyzers for the new DSL. Xtext itself as well as the DSLs designed with it are integrated in the Eclipse IDE.

Xbase [Eff+12] is an expression language that can be used within any DSL that is created with Xtext. It is a partial programming language with a Java-like syntax. The goal of Xbase is to reduce the necessary effort to implement a DSL. Xbase expressions are similar to Java expressions and the type system is linked to the Java type system. Since it is an expression language, Xbase does not have the concept of statements in contrast to Java.

3 XOCL4Inv: an Xbase Extension for OCL-Aligned Invariants

We present a prototypical language for invariant specifications, which provides a syntax and look-and-feel that is very close to OCL. Unfortunately, OCL cannot directly be used to automatically retrieve elements that cause an invariant violation as described in Section 4 for two reasons: a) a mechanism for indicating which elements shall be retrieved is needed, and b) the language should be restricted to forbid the formulation of invariants for which the demanded elements cannot be retrieved. Both could also be achieved by extending an OCL grammar, editor, and validation engine. We decided, however, to develop a new language that can easily be extended and integrated with other languages for further research and that leverages the functional programming style and tool-support of Xtend. We first present the structure of our language and then we explain the relation to OCL.

3.1 Language Structure

In XOCL4Inv, model constraints can be declared using invariants. Listing 1 shows the running example. Within the invariant declaration, a context type has to be specified that conforms to a type from the constrained model (`ReadingRoom`, line 1). The constraint must hold for all model instances of the provided type. These context elements are bound by the tool as implicit first parameters for each invariant declaration. Furthermore, a unique name is used as an identifier for the invariant (`AtLeast3ReferenceCopies`, line 2).

In addition to the context element, XOCL4Inv allows the declaration of optional invariant parameters (`Book b`, line 2). Each parameter has a unique name and specifies an element type. These parameters are used to indicate which elements of a certain type need to be bound from the invariant upon its violation. XOCL4Inv allows the specification of multiple invariant parameters which get bound to independent sets of constraint-violating elements.

Finally, the language allows the declaration of a constraint (line 3). A constraint is a boolean expression that has to hold for every model instance of the context element type, which

OCL	Xbase	Expression of extension method for XOCL4Inv
`iterate`	`fold`	-
`forAll`	`forall`	-
`forAll(a,b)`	-	`coll.product(coll).forall(predicate)`
`exists`	`exists`	-
`exists(a,b)`	-	`coll.product(coll).exists(predicate)`
`select`	`filter`	-
`reject`	-	`coll.filter[e|!predicate.apply(e)]`
`collect`	`flatten ∘ map`	-
`collectNested`	`map`	-
`isUnique`	-	`coll.groupBy[function.apply(it)].values .forall[it.size == 1]`
`sortedBy`	`sortBy`	-
`any`	`findFirst`	-
`one`	-	`coll.filter(predicate).size == 1`

Table 1: OCL iterators and corresponding Xbase or XOCL4Inv extension methods

is referenced using the keyword `self` within the expression. The constraint is an ordinary expression of the expression language Xbase used in the DSL bench Xtext.It can contain iterators to specify expressions that are iteratively evaluated for a multi-valued property of a metaclass or another collection. For each iterator, an iterator variable can be used to reference the individual element for each evaluation of the iterator expression. If such an iterator variable is explicitly declared as an invariant parameter, then the algorithm described in Section 4 can be used to transform the invariant in order to collect the violating elements.

3.2 OCL Alignment

Invariant declarations in XOCL4Inv are very similar in their structure to OCL. Both languages allow the specification of an invariant name, a context element, and a boolean constraint. Additionally, XOCL4Inv allows the optional specification of invariant parameters to indicate which model elements shall be retrieved for an invariant violation. Since XOCL4Inv constraints are formulated in Xbase, model elements, attributes, references, operations, collection types and primitive types can be used. Most constructs, such as enumerations, null values, and arithmetic and logical expressions exist in Xbase and OCL.

Furthermore, OCL provides methods marked with the prefix *ocl*, for instance `oclAsType` or `oclIsTypeOf`. These methods either rarely occur within invariant constraints, e.g. `oclIsInState`, or have an equivalent Xbase method, for instance type casts or `instanceof`-checks.To provide equivalent functionality for the remaining OCL operations that are commonly used, we defined extension methods which add custom behavior to existing types. Most of these extension methods operate on multi-valued properties. In Table 1, we show which operations for OCL iterators are already available in Xbase and which had to be added to XOCL4Inv. Other common collection operations that do not iterate over collections are shown in Table 2.

OCL	Xbase	Expression of extension method for XOCL4Inv
includes	contains	-
includesAll	containsAll	-
excludes	-	!coll.contains(object)
excludesAll	-	objects.forall[!coll.contains(it)]
isEmpty	empty	-
notEmpty	-	!coll.empty
size	size	-

Table 2: OCL operations without iterator variables and corresponding Xbase or XOCL4Inv methods

For every OCL operation, we either show a corresponding Xbase operation or an expression implemented in an equivalent XOCL4Inv extension method with the same name.

Like in OCL, constraints formulated in XOCL4Inv have to be side-effect free. Xbase in general does not have this restriction, so the language has to be restricted in order to prevent modifications of the model state through constraint checking. Side-effect free methods can be marked with a `@Pure` annotation. Additionally, library methods that cannot be annotated can be added to a user-defined whitelist for pure methods. The tools checks whether constraints only call methods that are marked accordingly or that are whitelisted. The whitelist and static analysis for side-effect free methods should, however, be improved in future work in order to reduce the amount of false positives.

4 Binding Variables of Violating Elements to Parameters

In this section, we explain how invariants formulated in XOCL4Inv are used to compute constraint-violating elements based on invariant parameters. First, we present our algorithm for transforming invariants to queries on a high level and introduce a running example. Then, we explain the individual steps and transformation rules in detail. Finally, we illustrate how the algorithm works by discussing the transformation of the running example.

4.1 Transformation Overview

Invariants in XOCL4Inv contain a boolean constraint for which named invariant parameters may be specified. For each of these parameters, our automated approach finds the corresponding model elements that violate the constraint, and binds these elements to the parameters. More precisely, our algorithm finds the unique multi-valued collection property that is iterated with an iterator variable and that matches the invariant parameter's name and type. For this collection, only those elements that are responsible for the invariant violation are bound. The tool transforms the invariant into a query that collects the constraint-violating elements and binds them to the parameters by executing several transformation steps.

First, the constraint expression is parsed into a custom expression tree. Then, the specified invariant parameters are matched to expression nodes for iterate operations. For every

```
1 context Library
2 invariant AtLeast3OpenReferenceCopies (Book b, List<Edition> editions)
3 check self.books.select[Book b|!b.stack.closed]
4   .map[it.editions.filter[it.referenceCopy]]
5   .forall[List<Edition> editions|editions.reduce[e1,e2|e1.copies + e2.copies] >= 3]
```

Listing 2: Complete example invariant ensuring at least three copies for open reference books

invariant parameter that needs to be bound, transformation rules are applied to the iterator node and its parent nodes in a copy of the expression tree. The resulting transformed expression tree represents the desired query. In a last step, this expression tree is converted back into a query expression, which is used to bind the computed elements to the parameters.

The presented approach has a few limitations: Currently, only invariant parameters that match an iterator variable can be specified. Other attributes and members of model elements can be used to formulate invariants but they cannot be bound to parameters. The effect of variables and members can also be expressed with iterators. Therefore, this is not a limitation of the expressiveness but an inconvenience. Nevertheless, we plan to support invariant parameters that match members or variables similar to the `let`-statement in OCL in future work. Furthermore, the algorithm has to apply transformation rules to the iterator node and all direct and indirect parent nodes. Only operations that correspond to a node for which a transformation rule is defined are therefore allowed after a parameterized iterate expression. Currently, these operations are `not, and, or, select, map, forall`, and `exists`. No such limitations exist for child nodes, i.e. the partial expressions prior to a matched iterator can be arbitrary Xbase expressions.

4.2 Running Example

In Listing 2, we present a complete version of the library invariant, which we already used to motivate our approach and to explain our language. This complete invariant illustrates more transformation rules (see Section 4.6) and applies to the more precise metamodel presented in Figure 1. In contrast to the metamodel used in the simplified invariant, books do not belong to a fixed reading room but to the library. They are stored in a stack which may be open to the public or not. Furthermore, the flag for reference copies and the attribute for number of copies are no longer specified for a book but for a specific edition of a book.

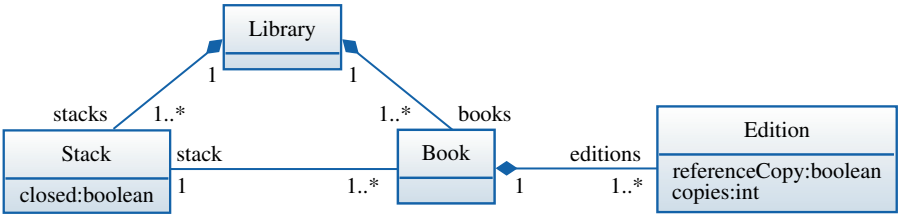

Figure 1: Library metamodel for the complete version of the example invariant

```
1 context Library
2 invariant Books4AtLeast3OpenReferenceCopies (Book b)
3 query self.books.select[Book b|!b.stack.closed &&
4    !(b.editions.filter[it.referenceCopy].reduce[e1,e2|e1.copies + e2.copies] >= 3)]]
```

Listing 3: Query for the complete example returning open reference books with less than three copies

The constraint of the extended example specifies that for every book in an open stack the sum of copies for all editions must total to more than three (line 3–5). If the constraint is violated, the responsible elements have to be computed. A trivial solution would be to return the library context element (line 1). This solution ignores, however, the collection and properties that are inspected during a check and does not determine a precise cause for an invariant violation. The directly responsible elements are those lists of editions for which the sum of copies does not satisfy the constraint. With our approach, these elements could be retrieved by specifying an invariant parameter `List<Edition> editions`. For our running example, we choose the invariant parameter `Book b` (line 2) to obtain those books of the library that have such a list. These indirectly responsible books can be retrieved by a query that is automatically derived from the invariant and shown in Listing 3.

Both example versions illustrate cases in which invariant parameters are needed in addition to contexts. First, there are cases in which an invariant only has to hold for instances with incoming references from the context element: The simplified invariant does not have to hold for books that are not in the reading room. Second, there are cases where a single context as in OCL is not enough because several elements may lead to a violation: Violations of the complete invariant can be resolved by manipulating books or lists of editions.

4.3 Obtaining a Custom Expression Tree Suited for our Transformation

Currently, the XOCL4Inv grammar specifies that invariant constraints can be arbitrary Xbase expressions. Therefore, we obtain an Abstract Syntax Tree (AST) of XExpressions from the parser generated with Xtext. The transformation algorithm defines rules based on much finer distinctions. For instance, all method calls result in a XMemberFeatureCall in Xbase, but method calls have to be transformed in a method-specific way. Calls to the methods `select` or `forall`, for example, have to be transformed differently. Therefore, we use a custom expression tree that differentiates between node types that have to be transformed differently and unifies node types that can be treated identically. This makes it possible to define transformation rules exactly for these node types and to focus on properties that are relevant for the transformation. The metamodel for the nodes of the custom expression tree is shown in Figure 2. A constraint in Xbase syntax is converted into an expression tree that consists of these custom nodes. The nodes' metaclasses are listed in Table 3, along with the XExpression and example expressions from which they are parsed.

The last benefit of a custom tree model are references to parent and child nodes, which are essential for the traversal of the expression tree during the transformation process. These

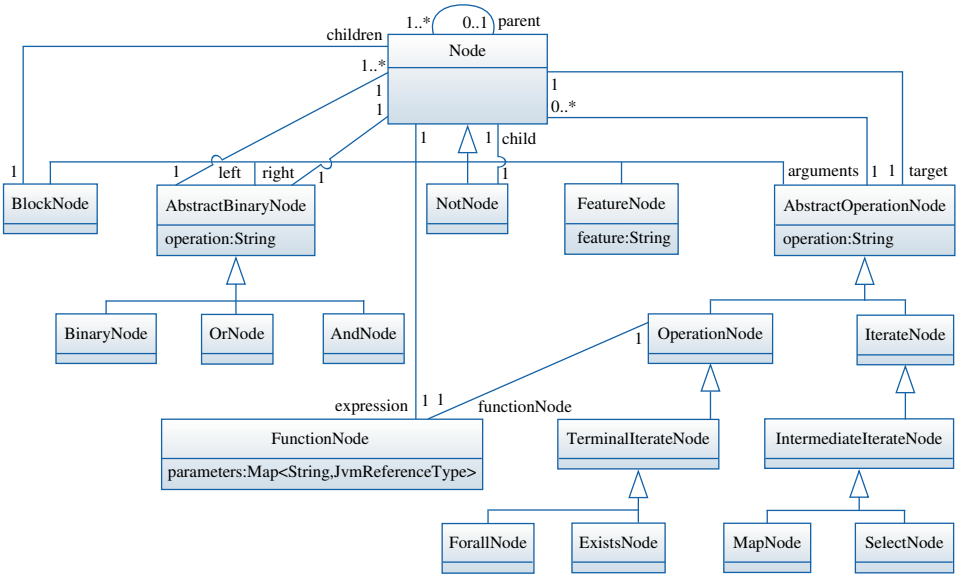

Figure 2: Custom node metamodel for expression trees

references are not contained in XExpressions and make it easier to create, copy, substitute, and modify nodes to apply individual transformation rules.

Currently, the following expressions cannot be transformed because we did not yet define custom node types and transformation rules: type casts, control structures, and variable declarations. Our prototype provides extension methods to transform equivalent constraints that use them instead of the unsupported expressions. In future work, these node types and transformation rules will be added and these extension methods will no longer be needed.

The expression tree for the running example invariant is presented in Figure 3. To obtain the pretty-printed expression shown in Listing 2, an in-order traversal is performed on the tree.

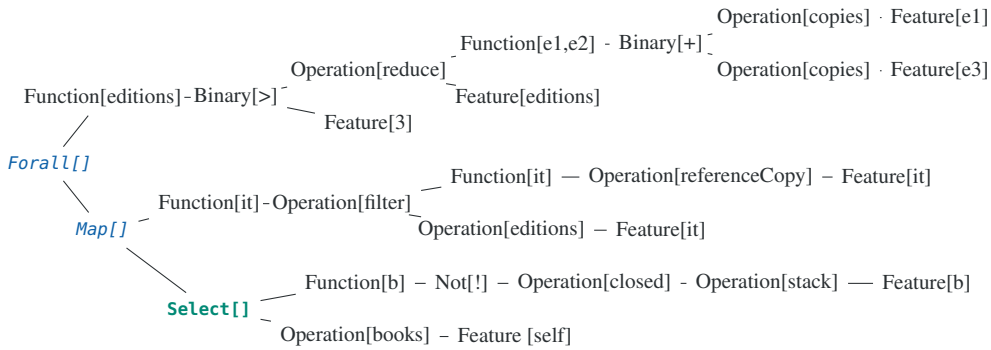

Figure 3: The custom expression tree that is obtained for the complete example invariant

Node type	Example expressions	Corresponding XExpression
ForallNode	`forall`	XMemberFeatureCall
ExistsNode	`exists`	XMemberFeatureCall
SelectNode	`select`	XMemberFeatureCall
MapNode	`map`	XMemberFeatureCall
OperationNode	`self.getBooks(), edition.copies`	XMemberFeatureCall
AndNode	`&&`	XBinaryOperation
OrNode	`||`	XBinaryOperation
BinaryNode	`<, +, /, ...`	XBinaryOperation
NotNode	`!`	XUnaryOperation
FeatureNode	`self, editions, b, it, 3`	XFeatureCall or Literal
FunctionNode	`[a|expression(a)]`	XClosure
BlockNode	`{...}`	XBlockExpression

Table 3: The classification of nodes that are used to build the expression tree

4.4 Matching Parameters to Iterator Nodes

In order to transform the invariant expressions for each specified invariant parameter, the algorithm first matches every parameter to its corresponding iterator node. More precisely, the expression tree is traversed with in-order depth-first search to find all nodes of type IterateNode. If the lambda function of an iterator node specifies an iterator that has the same name as the invariant parameter, the node is a name match candidate. In order to provide only unambiguous matches, both invariant parameter names and iterator variable names have to be unique within the complete invariant constraint.

A name match candidate is only a parameter match if the type of the iterator variable is assignment-compatible to the type of the invariant parameter. This ensures that the resulting query retrieves elements that can be bound to the statically typed invariant parameter. In the running example (Listing 2), the name of the invariant parameter `b` (line 2) matches the iterator variable of the `select` operation (line 3). Both have the same type and therefore the algorithm can proceed. In general, the algorithm finds a matching iterator node for each invariant parameter and transforms a separate invariant copy into a query to retrieve the violations. The required transformation rules are presented in the next section.

4.5 Transforming Iterator Nodes to Queries

Once the expression tree is generated and the matching iterator node is found for a specified invariant parameter, the tool transforms a copy of the expression tree into a tree for a query that selects the desired elements. This transformation is executed independently for every specified unique invariant parameter. The root of this tree is a SelectNode which selects the invariant-violating elements from the invariant context.

Given the constraint expression tree and an iterator node matching an invariant parameter, the algorithm recursively applies transformation rules. It starts top-down at the root node

and transforms child nodes until the iterator node is converted into the desired SelectNode. The query expression is finally obtained by performing an in-order traversal on the resulting query expression tree. The tool uses the transformed expression to bind the elements responsible for a specific constraint violation to the invariant parameter.

In the next paragraph, we explain the individual transformation rules for all transformable node types. The algorithm transforms the parent nodes recursively before transforming the actual node. Therefore, the transformation rules are not isolated but take the transformation result of the parent node into account.

For NotNodes rules of standard predicate logic are applied: Negated conjunctions (AndNode) and disjunctions (OrNode) are transformed by applying DeMorgan's laws. The nodes are replaced with their negated counterparts by pushing the negation inwards. A negated universal quantification (ForallNode) is replaced with an existential quantification (ExistsNode) for the negated predicate, and vice versa.

The ForallNode specifies that all elements in the target collection have to satisfy a given predicate. Therefore, the resulting query selects all elements that do not satisfy the predicate and thus violate the constraint.

```
coll.forall[e | predicate(e)]
-----------------------------
coll.select[e | !predicate(e)]
```

The ExistsNode specifies a predicate that has to be satisfied by at least one element in the target collection. If the constraint is violated, then all elements in the target collection are responsible as none of them satisfies the predicate. But if one element satisfies the predicate, then no elements have to be retrieved even if some of them may not satisfy the predicate.

```
coll.exists[e | predicate(e)]
-----------------------------
coll.select[!coll.exists[e | predicate(e)]]
```

A SelectNode only occurs with a parent node or as the result of a prior transformation. First, the parent node is transformed by applying the appropriate transformation rule to it. The result is a SelectNode for the parent. Then, the predicate of this parent SelectNode is conjunct with the predicate of the current SelectNode and the iterator variables are substituted accordingly to form a single resulting SelectNode.

```
coll.select[e | predicate(e)].select[p | parentPredicate(p)]
------------------------------------------------------------
coll.select[e | predicate(e) && parentPredicate(e)]
```

A MapNode applies a function to each element of the target collection. On this mapped collection, further iterate operations may be used. First, these operations are transformed into a SelectNode. Then, the mapping is inlined into the SelectNode: The MapNode is replaced by the SelectNode and all occurrences of the iterator variable are replaced with an application of the function that was specified in the MapNode.

```
coll.map[e | function(e)].select[p | predicate(p)]
--------------------------------------------------
self.select[e | predicate(function(e))]
```

An AndNode combines an expression that contains the unique iterator variable matching the invariant parameter with another expression. For the resulting query, elements referenced

by this iterator variable have to be retrieved if the expression with the matched variable evaluates to false. Whether the other expression without the matched variable also evaluates to false has no influence on the elements to be retrieved. Therefore, the transformation algorithm removes the expression without the matched variable and only transforms the expression with the matched variable. The order of the expressions does not matter. A swapped invariant `otherExpression && self...e...` is transformed the same way.

```
coll.forall/exists[e | predicate(e)] && otherExpression
-------------------------------------------------------
coll.select[e | predicate(e)]
```

An OrNode combines a parameterized expression and another predicate similar to an AndNode. But in contrast to the transformation for the conjunction, the other predicate of the disjunction cannot be ignored. If the expression evaluates to false but the other predicate holds, then the constraint is not violated. Therefore, the retrieved elements of the child expression may only be selected in the query if the other predicate is violated.

```
coll.forall/exists[e | predicate(e)] || otherPredicate
------------------------------------------------------
coll.select[e | predicate(e) && !otherPredicate]
```

4.6 Transformation Example

To illustrate the transformation we come back to the running example shown in Listing 2 and 3. In order to transform the SelectNode printed in **bold** in Figure 3 with the invariant parameter `Book b`, the algorithm recursively transforms the parent nodes printed in *italics*. It starts at the ForallNode and transforms it into:

```
...select[editions|!(editions.reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

The algorithm continues with the transformation of the MapNode. The previously obtained SelectNode is substituted with the following expression:

```
...select[!(it.editions.filter[it.referenceCopy]
 .reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

Last, the SelectNode with the invariant parameter is transformed by incorporating the parent node's predicate and substituting the iterator variable:

```
...select[Book b | !b.stack.closed &&!(b.editions.filter[it.referenceCopy]
 .reduce[e1,e2 | e1.copies + e2.copies] >= 3)]
```

The final result is the query presented in Listing 3. It retrieves all books that transitively violate the constraint and is bound to the invariant parameter `Book b` by our tool.

5 Evaluation and Discussion

We evaluated the correctness of the language and of the invariant transformation in two stages: In the first stage, we used synthetic test cases to ensure that invariants formulated in our language produce the same results as the equivalent OCL invariants and that the transformation algorithm produces queries that retrieve the correct elements. In the second stage, we used 88 out of 444 real invariants of the Eclipse metamodel for the UML to check that the language and the transformation algorithm fulfill these properties on them as well.

5.1 Evaluation of the Invariant Language

Our XOCL4Inv language is strongly aligned to OCL, which is commonly used for the specification of metamodel constraints. Therefore, it is necessary that OCL invariants can be expressed in a similar way in XOCL4Inv and produce identical results. We created 19 synthetic language test cases to compare OCL and XOCL4Inv expressions for every operation defined in tables 1 and 2. Each test case provides the same input models to an OCL expression and an XOCL4Inv expression containing Xbase or extension methods and verifies that they retrieve the same results.

In addition to these basic synthetic tests, we also evaluated XOCL4Inv using real invariants from the UML metamodel. It contains 444 constraints, which we categorized individually. 24 constraints have only a textual description, leaving 420 invariants in OCL notation. 175 of these compare attributes or properties of the context element and could be expressed in XOCL4Inv. Further 79 invariants compare sizes of collections from multi-valued properties of the context element. These constraints use the Xbase `size` or `empty` operations. In general, it is not possible to determine the elements that cause an invariant violation for these cardinality constraints because they may be part of the collection or not.

Most importantly, 88 constraints of the UML metamodel contain iterators that can be transformed, i.e. expressions containing `forall`, `exists`, `select`, or `map`. We formulated each of these invariants in XOCL4Inv, as described in the next subsection. A last set of 78 invariants cannot be assigned to any of the previous categories the invariants contain nested combinations of multiple operations or calls to unsupported operations, such as `allInstances`. These nested operations may contain nested invariant parameters, i.e. parameterized iterate operations within the predicate of other iterate operations, which are currently not supported by our prototype. Our approach is able to transform only the 88 invariants that contain non-nested iterate operations that specify an iterator variable. We are currently working on nested expressions by transforming non-nested and nested expressions separately and combining them afterwards.

5.2 Evaluation of the Binding Transformation

In order to test the correctness of the transformation, we checked for 19 additional synthetic transformation test cases that the tool generates the correct query for a given invariant with a parameter and that this query retrieves the correct model elements. For each test, input-output pairs verify that the retrieved elements are equal to the expected ones. The synthetic tests cover all node-operation combinations and the following numbered expression categories:

- 1-2 Unchained parameterized `forall` and `exists` invariants
- 3-4 A `select` with the invariant parameter followed by `forall` (3) or `exists` (4)
- 5-6 A parameterized `map` followed by `forall` or `exists`
- 7-8 Both `&&` and `||` combining a parameterized `forall` with a second predicate
- 9-16 The negation of 1-8
- 17-19 The operations `forall` and `exists` as well as conjunctions and disjunctions in combination with parent nodes

The transformative approach is evaluated on all 88 invariants of the UML metamodel that contain iterators that can be transformed or rewritten to be transformed. For each invariant, we provide an equivalent formulation in XOCL4Inv and manually checked that the generated query equals the expected outcome of the transformation algorithm.

5.3 Discussion

The language and transformation evaluation results for the UML metamodel are promising but have to be complemented by evaluations on further metamodels and further invariants. We demonstrated successfully that we can express and transform 88 invariants of a popular UML metamodel, but it is unclear whether all other Ecore metamodels and OCL invariants can be processed. Therefore, additional invariants should be used to confirm that the language and transformation also work for OCL expressions that are not used in the UML invariants and for structural patterns that do not occur in the UML metamodel.

Furthermore, the transformation evaluation for the UML invariants is based on a manual inspection of the obtained query. In future work this inspection should be automated in order to ensure systematically that the obtained query retrieves the correct elements for several input models. It is, however, an open question whether some of these input models and the sets of elements to be returned for violations can be generated or whether they have to be created manually and can only be automatically compared.

If the transformation is extended in order to transform further expressions, this extension should be evaluated using some of the 78 invariants of the UML metamodel that can currently not be transformed, but also using further invariants of other metamodels. A formal proof of correctness could be done for each transformation rule, but as it mainly realizes well-known predicate logic the benefit of such proofs is disputable.

6 Related Work

Sigma [KC12] is a hybrid model transformation library or internal domain-specific language for Scala. It supports declarative transformation rules and imperative validation and transformation code. Sigma groups constraints in validation contexts and provides facilities to define severity levels for invariant violations, error messages and repair actions. If model elements that caused an invariant validation are used in such repair actions, then the computation of these elements has to be explicitly defined in addition to the definition of the invariant check [cf. KCF14, p.1613, ll.19–22]. Parts of the checking of an invariant can, of course, be factored out and be reused for retrieving model elements. For most invariants this is, however, much more verbose than specifying constraint parameters in our approach.

The Epsilon Validation Language (EVL) [KPP09] of the Epsilon framework is similar to OCL but overcomes several shortcomings of it. Similar to Sigma, it supports the definition of fix procedures for invariants. These fixes are tightly coupled to a constraint, so that it is not possible to write several fixes for a single constraint without repeating it. In contrast to

our approach, parameters that are defined in invariant checks cannot be reused directly in fixes. They have to be defined and computed again in fixes [cf. KPP09, p.215, ll.47–63].

Furthermore, both Sigma and EVL do not separate the definition of invariant checks from fixes. This may be crucial if different fixes are to be defined for different editors, transformations, development projects, or customers while some of the corresponding invariants may be defined for a metamodel regardless of its usage. We are planning to support such a reuse of invariants in the domain-specific language for model consistency [Kra15], which will integrate the invariants language presented in this paper.

EMF-IncQuery [Ber+12; Ujh+15] is a framework for declarative model queries. It performs incremental graph pattern matching based on Rete networks. IncQuery provides a live validation service that can report constraints validations directly after the modification that lead to it. An annotation can be used to turn an ordinary graph pattern into constraints and to define severity levels or error messages for it. Parameters of a constraint pattern can be designated as keys to identify a violation which is a pattern match. These constraint keys are equivalent to the invariant parameters of the presented approach: They also provide elements that lead to a violation based on explicit constraint parameters and do not force developers to repeat parts of the constraint checking logic in order to obtain these elements. The main difference to our approach is, however, the relation to OCL: If elements that cause an invariant violation shall be computed for pre-existing OCL invariants, the transition to our approach based on an OCL-aligned language should require less effort than implementing these invariants again with IncQuery. In projects where such invariants exists and where developers are already familiar with OCL but not with IncQuery, our approach may be more appropriate. To further ease such a use for legacy OCL invariants we are currently working on a automated conversion from OCL to XOCL4Inv. There is a translation from OCL queries to graph patterns that can be queried using EMF-IncQuery [**bergmann2014a**]. With this approach, it would be possible to modify the patterns that result from an OCL invariant in order to obtain wanted elements that violate the invariant. The goal of the translation was, however, better performance. Therefore, a conceptual mapping from the resulting patterns to the initial OCL invariant may not always be straightforward, in contrast to our approach.

7 Conclusion

In this paper, we have presented an OCL-aligned language for invariants with explicit parameters and an algorithm to bind elements causing an invariant violation to these parameters. First, we have explained why an automated computation of such elements from a redundancy-free invariant definition is important to document or resolve invariant violations without unnecessary effort. Then, we have introduced the XOCL4Inv language that combines the style of OCL invariants with the extensibility and power of the expression language Xbase. We have presented an algorithm that obtains queries that compute elements causing an invariant violation by recursively applying transformation rules for invariant expressions. Last, we have discussed how we evaluated the correctness of our language and

transformation algorithm with a prototypical implementation and invariants taken from a popular UML metamodel.

In future work, we are going to evaluate the language and algorithm with invariants of additional case studies. We are also planning to add transformation rules for invariant expressions and parameters that cannot yet be processed. Finally, we are going to reduce the amount of false alarms for side-effects in invariants.

References

[Ber+12] G. Bergmann et al. “Change-driven model transformations”. In: *Software & Systems Modeling* 11.3 (2012), pp. 431–461.

[Eff+12] S. Efftinge et al. “Xbase: Implementing Domain-specific Languages for Java”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE ’12. ACM, 2012, pp. 112–121.

[EV06] S. Efftinge and M. Völter. “oAW xText: A framework for textual DSLs”. In: *Eclipsecon Summit Europe 2006*. 2006.

[ISO12a] ISO/IEC 19505-2:2012(E). *Information technology – Object Management Group Unified Modeling Language (OMG UML), Superstructure*. International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–758.

[ISO12b] ISO/IEC 19507:2012(E). *Information technology – Object Management Group Object Constraint Language (OCL)*. International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–234.

[ISO14] ISO/IEC 19508:2014(E). *Information technology – Object Management Group Meta Object Facility (MOF) Core*. International Organization for Standardization, Geneva, Switzerland, 2014.

[KC12] F. Křikava and P. Collet. “On the Use of an Internal DSL for Enriching EMF Models”. In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL ’12. ACM, 2012, pp. 25–30.

[KCF14] F. Křikava, P. Collet, and R. B. France. “Manipulating Models Using Internal Domain-specific Languages”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. ACM, 2014, pp. 1612–1614.

[KPP09] D. S. Kolovos, R. F. Paige, and F. A. Polack. “On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages”. In: *Rigorous Methods for Software Construction and Analysis*. Vol. 5115. LNCS. Springer Berlin Heidelberg, 2009, pp. 204–218.

[Kra15] M. E. Kramer. “A Generative Approach to Change-Driven Consistency in Multi-View Modeling”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA ’15. 20th International Doctoral Symposium on Components and Architecture (WCOP ’15). ACM, 2015, pp. 129–134.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[Ujh+15] Z. Ujhelyi et al. “EMF-IncQuery: An integrated development environment for live model queries”. In: *Science of Computer Programming* 98, Part 1 (2015), pp. 80–99.