Simplifying Transformations of OCL Constraints

Martin Giese[1] and Daniel Larsson[2]

[1] Johann Radon Institute for Computational and Applied Mathematics
Altenbergerstr. 69, A-4040 Linz, Austria
martin.giese@oeaw.ac.at
[2] Chalmers University of Technology
Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden
danla@cs.chalmers.se

Abstract. With the advent of Model Driven Architecture, OCL constraints are no longer necessarily written by humans. They can be part of models that emerge from a chain of transformations. They might be the result of instantiating templates, of combining prefabricated parts, or of more general computation. Such generated specifications will often contain redundancies that reduce their readability. In this paper, we explore the possibilities of transforming OCL formulae to a simpler form through the repeated application of simple rules. We discuss the different kinds of rules that are needed, and we describe a prototypical implementation of the approach.

1 Introduction

The Object Constraint Language (OCL) [12] is designed with human authors and readers in mind. While some of today's UML tools allow attaching OCL constraints to diagrams and checking their syntax with a parser, there is practically no support for authoring OCL specifications. But writing good specifications is hard, and as the software to be specified becomes larger and more complex, designers will need tools that help them with that task.

OCL constraints might result from a transformation of a more abstract description of the system. For instance, constraints written at the analysis level might be transformed into design level constraints by some tool. Or a specification in some other graphical or logic-based formalism might be translated into OCL.

While tools performing such tasks have yet to be written, we already encounter tool-generated constraints in connection with an extension of the 'design pattern' instantiation mechanism provided by various case tools [2].[3] The idea is to let the user instantiate templates, pieces of class diagrams, which provide implementations for various design patterns. As part of the instantiation, one

[3] In the present work we employ Borland Together ControlCenter (TCC), see http://www.borland.com/together/index.html.

can generate OCL constraints that capture certain properties of the pattern.[4] Due to the availability of the tool as part of the KeY system [1], we will use the design pattern instantiation scenario in the motivating example for this paper. See e.g. [7, 8] for other work involving tool-generated OCL constraints.

Whichever scenario we pick, the (semi-)automatically generated constraints will often contain redundancies that make them hard to read for humans. The topic of this paper is how OCL constraints can be simplified with the goal of making them more readable. We will propose a rule-based method where various simple rules get applied exhaustively.

In Sect. 2, we describe the context of this work and give a motivating example. We show in Sect. 3 how a generated constraint can be simplified. We then analyze the required simplification steps in Sect. 4. In Sect. 5, a prototypical implementation of our ideas within the KeY system is presented. Sect. 6 discusses related work. Finally, Sect. 7 concludes the paper with some remarks about future work.

2 Motivation

The KeY tool [1] is a CASE tool in which formal methods are integrated with contemporary software development techniques. Besides the usual tasks of a CASE tool of creating UML models and creating implementations in Java, KeY allows the developer to add formal specifications to a model in the form of OCL constraints. One of the main goals of the KeY project is to spread the use of formal methods in software development, and a crucial step in the use of formal methods is the authoring of formal specifications like OCL constraints.

Unfortunately, it is not easy to write useful formal specifications. This is one of the major obstacles in getting developers to use formal methods in software development. One possible solution to this problem would be to, somehow, *automatically generate* formal specifications out of some prior information. Ideally, we would like to go directly from an informal specification to a formal one, but the possibilities to do so are very limited. However, an experienced developer can often recognize parts of an informal specification as instances of certain *design patterns* and, given a specific design pattern, it is possible to generate a formal specification that expresses useful requirements associated with that pattern [2].

As software development is becoming more and more structured, using patterns, frameworks, and so on, it is very natural that authoring of formal specifications also follows that line. It is a good way of re-using and taking advantage of experienced developers' knowledge.

[4] Design patterns in the usual sense of the word [5] provide a vocabulary for communicating design ideas. They are relatively abstract entities, consisting of textual descriptions of why, when, and how to use them, and the consequences of using them. What is called "design pattern" in CASE tools like TCC is just mechanical instantiation of templates. It is nevertheless useful, and it is such a template mechanism we use as an example in this work.

How can we obtain a formal specification for a design pattern? The problem is that in order to write a useful specification we need some information that is not available until the pattern gets instantiated, i.e. applied to a concrete design. Until then we do not know:

- The name space of the modeled domain, i.e. we do not know the names of the classes, fields, methods, associations, etc. in the design to which the pattern is being applied.
- How the developer will modify the structure of the pattern, i.e. adding or removing classes, fields, methods, associations, etc.
- What *flavor* of the pattern the developer will use. By flavor we here mean that different instances of a specific design pattern can have different requirements associated with it regarding some details.

2.1 Example of Constraint Generation

Let us look at a concrete example to make this more clear. The intention of the *Observer* design pattern (taken from [5]), that is shown in Fig. 1, is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." This pattern is useful when one needs to maintain consistency between related objects, but one does not want to achieve this by making the classes tightly coupled.

In modern CASE tools such as TCC, one can perform machine-assisted application of design patterns. The user then has to supply a mapping from the name space of the pattern to the name space of the modeled domain. Optionally, the user may choose to modify the structure of the pattern. In the KeY tool, the user can also choose what flavor of the pattern he wants to use. In the context of the Observer pattern we can, for instance, find the following "flavor component":

- *Should the observers be allowed to observe more than one subject?* In other words, what should be the multiplicity of the `subject` association-end: `0..1` or `0..*`? In some situations, observers need information from more than one source, and then it might be a good idea to let them observe more than one subject. For example, a spreadsheet may depend on more than one data source.

An *instance* of the Observer pattern is shown in Fig. 2. This example is from the design of a system that handles statistical data. The statistics can be viewed graphically, both as a pie chart and as a bar chart. We can see that what is called `ConcreteSubject` in the pattern is here called `Statistics`, the role-name `statistics` corresponds to the role-name `subject` in the pattern, and so on. We can also see that we here have two concrete observers (`PieChart` and `BarChart`) in contrast to the single one in the pattern (`ConcreteObserver`), so the original structure of the pattern has been slightly modified.

What flavor of the pattern would be useful for this particular instance? Let us assume that the GUI observers only need information from one `Statistics` object, i.e. the `subject` association-end has multiplicity `0..1`.

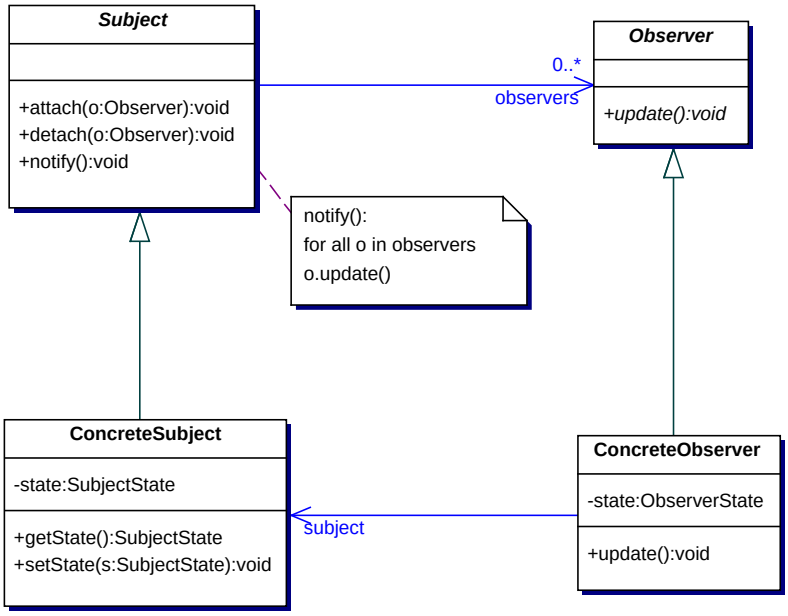

Fig. 1. The Observer pattern

Now, if we are going to write a formal specification for this design pattern, we need information that we do not obtain until the pattern is instantiated. A possible solution is to use *schemas*, as suggested in [2]. For each design pattern we want to specify, we design a schema from which we can generate formal specifications when the pattern is applied. A schema for (part of) the Observer pattern might look like this:

```
schema numOfSubjects(String flavor)
 ocl: context Observer inv:
       if flavor = 'one'
        then self.subject->size() <= 1
        else true
       endif
```

Here we have a parameterized version of one of the "flavor components", namely whether the `subject` association-end should have multiplicity `0..1` or `0..*`. The keyword `schema`, the name of the schema, optional flavor parameters, and the keyword `ocl:` are followed by the actual OCL constraint containing the flavor parameters. But there is a problem with this schema. There is no inheritance mechanism in the semantics of OCL, and this means that a formal specification will be generated for the abstract class `Observer` but not for the concrete ob-

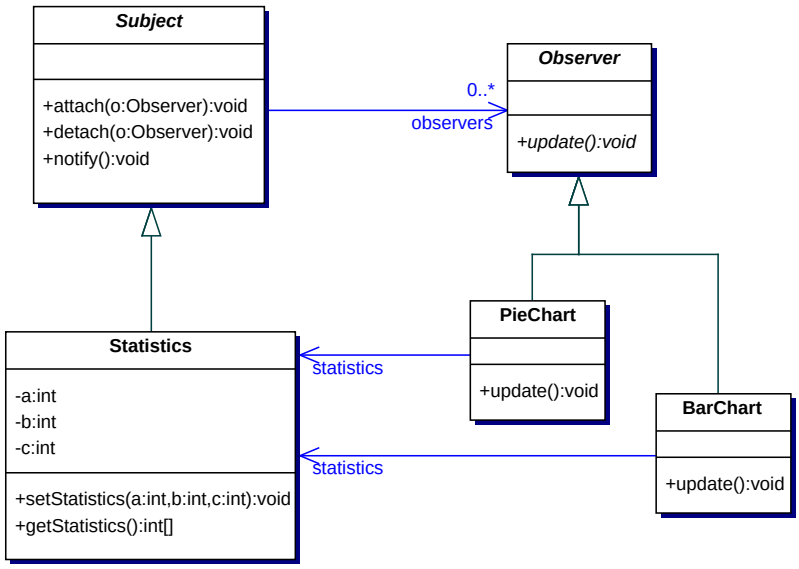

Fig. 2. An instance of the Observer pattern

server objects `PieChart` and `BarChart`. Writing a schema for `ConcreteObserver` instead, so that a specification is generated for each concrete observer in the model, does not solve the problem in general. If the developer introduces a hierarchy of observers including abstract super classes for subsets of the observers, then we have the same problem again. However, we can address the problem directly in our schema:

```
schema numOfSubjects(String flavor)
 ocl: Observer.allSubtypes()->
       forAll(s | s.allInstances()->
        forAll(i |
           if flavor = 'one'
            then i.subject->size() <= 1
            else true
           endif))
```

Now we quantify over all subtypes of `Observer`, and for each subtype we quantify over all instances of that subtype. This means that in essence we will get an invariant for each subtype. (The property `allSubtypes` is not pre-defined in OCL but can be expressed with the help of other operations. We just use it here to make the constraint more readable.) Here is the result of applying the Observer pattern to the model in Fig. 2 and using the schema above to generate a specification for the pattern instance:

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i |
          if 'one' = 'one'
           then i.statistics->size() <= 1
           else true
          endif))
```

`Observer` in the pattern is mapped onto `Observer` (could have another name) in the model, the `subject` association in the pattern is mapped onto `statistics`, and the parameter `flavor` is bound to the string literal `'one'`. As one can see, there is a potential for simplification here. Since we now have a concrete design, it should be possible to evaluate the expression `Observer.allSubtypes()`. It should also be possible to evaluate the if-then-else construct now that the `flavor` parameter is bound to a concrete value.

In general, when we write OCL constraint schemas for design patterns, they will be parameterized. We will have explicit parameters of the schema for different flavors of the pattern. The elements from the pattern's name space can also be viewed as formal parameters, since they have to be bound to concrete elements from the modeled domain. Moreover, we have to take into account possible structural modifications of the pattern. As we saw in the example, all this will lead to generated specifications containing redundant information. They become hard to read, hard to understand. The generated specifications need to be simplified.

3 Example

We shall now see how the previous example may be simplified through the application of several small simplification steps. The first step would be to recognize that `'one' = 'one'` is always true, and may therefore be replaced by `true`:

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i | if true
              then i.statistics->size() <= 1
              else true
             endif))
```

Next, an if-then-else construct with a condition known to be true may be replaced by its then-branch:

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i | i.statistics->size() <= 1))
```

A further simplification becomes possible if we take information about the model into account, namely the subtypes of `Observer` in this particular instance of the pattern. In Fig. 2, there are only two subtypes, so we can simplify the constraint as follows:

```
Set{PieChart,BarChart}->
 forAll(s | s.allInstances()->
  forAll(i | i.statistics->size() <= 1))
```

The outer `forAll` application now ranges over a finite set of which we know all elements. We can therefore transform it into a conjunction:

```
   PieChart.allInstances()->forAll(i | i.statistics->size() <= 1)
and BarChart.allInstances()->forAll(i | i.statistics->size() <= 1)
```

Finally, a property that should hold for all instances of a class is usually stated as an invariant. One could therefore split up this constraint and add an invariant to both of the observer classes:

```
context PieChart inv: statistics->size() <= 1
context BarChar  inv: statistics->size() <= 1
```

These constraints are certainly much simpler and more natural than the original general form from the schema. On the other hand, the meaning is guaranteed to be the same, as none of the small transformations changed it.

As a second example, let us assume that `flavor` was bound to `many`. We then get the constraint

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i | if 'many' = 'one'
             then i.statistics->size() <= 1
             else true
            endif))
```

The strings `'many'` and `'one'` are different, so the condition can be simplified to `false`:

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i | if false
             then i.statistics->size() <= 1
             else true
            endif))
```

In this case, only the else-branch needs to be kept:

```
Observer.allSubtypes()->
 forAll(s | s.allInstances()->
  forAll(i | true))
```

A `forAll` expression with body `true` always evaluates to `true`, so we can simplify this to

```
Observer.allSubtypes()->
 forAll(s | true)
```

and finally to

```
true
```

In this case, the constraint gets simplified away completely, as it does not say anything about the case where `flavor` is not `'one'`.

Another example of a step-wise simplification is given in [2]. There, OCL constraints from an instantiation of the Composite pattern are simplified in a similar way as was presented here.

4 Analysis

The previous section shows how OCL constraints can be simplified considerably through the repeated application of small, simple rules. All rules require only local transformation of the constraint, no global analysis is needed. This suggests implementing our simplification using a kind of rewriting rule engine. Such a rule engine repeatedly tries to apply transformation rules on subexpressions of the input until no more rules are applicable. This is a well-known principle and our work can profit from the extensive research on term rewriting systems (see e.g. [4]).

In this section, we are going to have a closer look at the different kinds of rules that are needed to simplify OCL constraints.

4.1 Primitive Types

The most fundamental primitive type in OCL is of course the `Boolean` type. For this, general logic simplification steps are needed, like for instance rewriting `false and e` to `false`, `true and e` to `e`, etc. One can give this kind of rules for all logical connectives, including the if-then-else construct. The rule that simplifies `e=e` to `true` also belongs to this category.[5]

A more difficult question is how to handle the other data types built into OCL. For instance, one surely wants to have simplification rules that rewrite `2+3` to `5`. Simplifying `0+x` to `x` is also useful. But should one have rules capable of simplifying `(x+y)*(x+y) - (x-y)*(x-y)` to `4*x*y`? It is known from computer algebra research that the simplification of algebraic expressions is a complicated affair. We think that it depends very much on the application field whether an OCL simplifier should be able to handle this kind of problem. If one thinks of design pattern instantiation, then it seems unlikely that algebraic simplification would be useful. We limit ourselves to evaluation of concrete expressions and simple laws on neutral elements, units, etc., until we come across an application that makes more powerful simplification necessary. This holds for all the primitive data types of OCL, i.e. integers, reals, and strings.

[5] We have so far ignored the difficulties of handling `undefined` using an appropriate three-valued semantics.

4.2 Collection Types

In Sect. 3 we saw an example of how a `forAll` expression over a finite set can be rewritten to a conjunction. Many interesting simplifications are possible for collections. Here are some examples:

- Operations with finite sets can be simplified: `Set{a,b}->exists(x|p(x))` can be written as `p(a) or p(b)`.
- Some operations can be completely evaluated for concrete sets. For instance, `Set{1,2,3}->sum()` can be simplified to `6`.
- Operations where the other parameters have a simple form can often by simplified: `s->forall(x|true)` can be rewritten to `true` and, if `s` is a `Bag`, `s->collect(x|x)` can be reduced to `s`.
- Special cases can be detected for some operations. For instance, one might rewrite `s->including(o)->includes(o)` to `true`.

As is the case for primitive types, no finite set of simplification rules can cover all cases. One should therefore pick a basic supply and extend it when applications make it necessary.

An common property of the collection operators in OCL is that they can all be expressed using the `iterate` construct. We can reduce the number of needed simplification rules for the various collection operators by translating them to an `iterate` expression and providing simplification rules only for that. For instance, the previously mentioned expression

```
Set{a,b}->exists(x | p(x))
```

can be written as

```
Set{a,b}->iterate(x ; acc:Boolean = false | acc or p(x) )
```

The iteration over the finite set can then be unrolled to

```
(false or p(a)) or p(b)
```

which is in turn simplified to

```
p(a) or p(b)
```

It turns out that most of the simplifications one might think of for `forAll`, `exists`, `collect`, etc., can actually be handled in this way. If one has m simplification rules for n operators, one can effectively replace $m \cdot n$ rules by $m + n$.

The drawback of this approach is what happens when the expression *cannot* be simplified further after translation to the `iterate` form: in that case, the latter form is certainly harder to read than the original. Our current solution to this problem is to provide a number of rules for the inverse transformation, i.e. to transform `iterate` expressions of certain forms to `forAll`, `exists`, etc. These rules are applied as a final step after all other simplifications. In other words, simplification proceeds in two phases. In the first phase, everything is translated to the `iterate` form and simplified as much as possible. In the second phase, remaining `iterate` expressions are translated back to the various simpler operators.

This approach leads to an interesting theoretical question, namely which properties the rule sets for the two phases should have to make the overall behaviour equivalent to that of a single phase with $m \cdot n$ rules.

4.3 Model Dependent Simplifications

The previously discussed simplification rules are not very specific to OCL. They would make sense in any formal language that provides the same data types. A peculiarity of OCL is that OCL constraints are always attached to UML diagrams. They cannot occur in isolation. Accordingly, we can identify simplification rules that depend on the model.

For instance, some of the properties (operations) defined by OCL refer to the operations and attributes available for a type, rather than to a state of the modeled system.[6] If the concrete model is known, these can often be evaluated. For instance, in our example, `PieChart.supertypes()` can be simplified to `Set{Observer}`. In contrast to the simplifications proposed in the previous sections, this requires knowledge of the model. Similarly, expressions involving the `attributes`, `operations`, etc., properties defined for `OclType` will usually be simplifiable once the model is known.

The use of information from the UML model is not limited to 'meta'-properties: another possibility might be to use the multiplicities of associations. For instance, if the association `assoc` has a multiplicity of n, then `o.assoc->size()` can be simplified to n.

The interesting issue here is how to organize the implementation of such simplifications. An implementation that uses a rule engine with a fixed set of syntactic rewrite rules would have to generate a considerable number of rules from the model. For instance, there would be a rule for each of the `OclType` properties for each class in the model. Even worse, simplification rules that involve two types, for instance for expressions involving `oclIsKindOf`, might need one rule for every *pair* of classes, so the rule set would grow quadratically in the size of the model. At the same time, most of these rules would not be needed for any particular simplification.

To avoid this waste of resources, a practical solution requires a rule engine that can obtain information from the model to determine the applicability and result of some of the rules. This is the approach we have chosen, using the rule engine's 'meta constructs' as described in the next section.

We have not yet discussed the final step in Sect. 3, where a single constraint is split up and distributed among the invariants of several classes. This could be done as a post-processing step, but we chose to incorporate it into our rule based mechanism. The simplification rules are not applied on raw constraints, but on lists of constraints with contexts. This allows us to formulate rules that add constraints to different classes.

[6] In other words, these properties return information about the state of the *metamodel*. Up to OCL 1.5, these were predefined on the type `OclType`. In OCL 2.0, they were removed to avoid inconsistencies between OCL and the UML metamodel. Our discussion is based on the properties as defined in OCL 1.5.

5 Implementation

We have implemented a prototype of a rule-based OCL simplifier and integrated it with the pattern-instantiation mechanism in KeY. It is now possible in KeY to generate OCL specifications for instances of certain design patterns, with the help of schemas, and then to use the simplifier to simplify the generated specifications. To parse the OCL expressions that need to be simplified, we use a parser and typechecker that has been developed at Chalmers University and is described in [9]. When implementing the OCL simplifier, we used the fact that we already had a rule-engine available: the theorem prover in the KeY tool. This theorem prover is based on *taclets* [3], which are a kind of generalized term rewriting rules that can be used to describe the rules of a logic calculus.

5.1 Taclets

Although the taclet concept was designed with theorem proving in mind, the design is so general that it is possible to use taclets for other purposes as well. After a few extensions of the implementation of the KeY taclet engine, we were able to use taclets to perform OCL simplification.[7] A rewrite taclet for OCL simplification can, for instance, look like this:

```
find(#e and true) replacewith(#e)
```

Here, `#e` is a *schema variable*, i.e. it stands for an arbitrary expression. This taclet is *applicable* to an OCL expression `exp` if the find-part of the taclet matches `exp` (i.e. if we can instantiate the schema variable so that `exp` and the find-part become identical). If we *apply* the taclet to `exp`, then `exp` will be replaced by the instantiated replacewith-part of the taclet. The schema variables used in a taclet must first be declared, meaning that they are given a type. In this way we can ensure that taclets are only applicable to expressions with matching types. New rules can easily be defined in a text file, using the notation above, and are then parsed into the KeY system. Our approach is to write a set of simplification rules, in the form of taclets, and then apply them to the generated OCL specifications.

Each OCL taclet contains a find-expression and a replacewith-expression, both consisting of OCL syntax extended with schema variables, *meta constructs*, and the *substitution operator*. Meta constructs are references to procedures that transform a given OCL expression into another one when a taclet is being applied. They are only allowed to appear in the replacewith-part and are used to extract information from the UML model, e.g. the subtyping hierarchy of classes. Most taclets do not need any meta constructs. The meaning of the substitution operator will be explained in the context of an example below. Here are some examples of taclets needed for OCL simplification:

[7] There exists a number of model transformation languages within the MDA framework, and one of them could probably have been used to express the OCL simplification. However, when our project started no tool support was available for these languages, and we therefore went for the taclet solution.

```
equals {find(#e = #e) replacewith(true)}

and_false {find(#e and false) replacewith(false)}

if_true {find(if true then #e1 else #e2 endif)
         replacewith(#e1)}
```

In the examples, all schema variables are prefixed with a '`#`' sign to distinguish them from the keywords in the syntax. It should be pointed out that in the current implementation, one cannot use proper OCL syntax in the taclets like in the examples. A special, taclet-tailored syntax has to be used instead. This is due to the difficulties in integrating the parser for the taclet language with an OCL parser. Of course, this technicality will be visible only to the author of the simplification rules, and not to the user of the simplifier.

5.2 Collections

In order to simplify OCL expressions, one has to have a way of dealing with OCL collections. The constructors for OCL collections (`Set{...}`, etc.) can enumerate any number of elements, i.e. they can be viewed as operators having a *variable arity*. Now, operators with variable arity are not very easy to handle in an efficient way when one wants to apply rules to them. Our solution to this is to represent collection literals in structures that resemble the list in functional programming languages. These structures are built using two constructors: `insert` that takes two arguments—the "first" element in the collection and the rest of the collection—and `empty` that represents the empty collection.

To be more precise, we have two collection constructors for each collection type: `insert_set` and `empty_set`, `insert_bag` and `empty_bag`, and so on. In that way we do not lose the type information. Using these collection constructors, it is easy to perform various operations on OCL collections. Instead of having to deal with variable arity, we use *induction* when designing our simplification rules: we have one base case rule for the empty collection, and one induction step rule, just as one defines functions operating on lists in functional programming languages. As an example for this representation, `Set{a, b, c}` becomes

```
insert_set(a, insert_set(b, insert_set(c, empty_set)))
```

We can now define taclets to transform a universal quantification over a concrete, finite set to a conjunction. In other words, we want to transform an expression like

```
insert_set(a, insert_set(b, insert_set(c, empty_set)))
  ->forAll(x | e(x))
```

to

```
e(a) and e(b) and e(c)
```

Below we see the two taclets needed to perform this transformation, one rule for the base case and one for the induction step:[8]

```
forAll2Conjunction_base {
   find(empty_set->forAll(#x | #exp))
   replacewith(true)}

forAll2Conjunction_step {
   find(insert_set(#head, #tail)->forAll(#x | #exp))
   replacewith({#x #head}#exp and #tail->forAll(#x | #exp))}
```

Here we can see the syntax for *substitution*, `{x e}exp`, which causes all free occurrences of `x` in `exp` to be replaced with `e` once the taclet is applied.

5.3 Type Inference

Another thing we must handle in our implementation is a certain degree of type inference. The type of an OCL expression is in most cases given directly by the top operator of the expression, but in some cases one has to infer the type of the expression from the types of its subexpressions. For example, the type of an expression with `forAll()` as top operator is always `Boolean`, while the type of the expression `if b then e1 else e2 endif` is the least common supertype of the types of `e1` and `e2`. One way to implement a type system that handles this kind of type inference would be to design a general type inference algorithm, e.g. using unification, like the ones found in functional programming languages like ML and Haskell. However, combining such an algorithm with subtyping is a delicate matter. Moreover, there are relatively few, builtin OCL operations that need special treatment, and their number is fixed. Instead of using a general type inference algorithm, we have therefore chosen to hard-code the necessary inference directly in the representation of these OCL operations.

5.4 Status of Implementation

In order to perform OCL simplification using taclets we have extended the implementation of the KeY tool so that we can now, for instance, perform the simplification steps described in Sect. 3, and also what is needed for the problem described in [2]. We can perform basic simplifications, like `x and true` to `x`, but also more advanced tasks like extracting information from the model using meta constructs. We can also handle bound variables and express the substitution of such variables, which is needed to handle `forAll`, `iterate`, etc. Moreover, we have extended the pretty-printing module of KeY so that the simplified OCL expression can be displayed in proper OCL syntax. What remains to be done is to complete our set of taclets. So far we have only written simplification taclets for a few design pattern schemas.

[8] As mentioned in Sect. 4.2, this simplification would actually be performed via the `iterate` representation. To make our presentation simpler, we here give rules that simplify `forAll`-expressions directly.

Since it is possible to use the prover of the KeY system "stand-alone", without the CASE tool component, we expect to be able to produce a stand-alone version of our OCL simplifier as well.

6 Related Work

The idea to attach schematic OCL constraints to design patterns was first discussed in [2]. The need for simplification was recognized there, but not systematically investigated. This is done in the present paper, together with an implementation approach.

The authors have explored the idea of applying partial evaluation [10] techniques to simplification, but the approach turned out to be rather unfruitful, as discussed in Sect. 6 of [6].

7 Conclusion

We presented an approach to perform OCL simplification through repeated application of simple rules. Simplification of OCL constraints is often needed when the constraints have been automatically generated by instantiation of templates, by combination of constraint fragments, or by some other technique. On a higher level, we think that tool support for the generation of formal specifications is an important step on the way to make formal methods more accessible to software developers. In this paper we concentrated on how to simplify OCL constraints generated in the context of design pattern instantiation, i.e. constraints expressing requirements associated with the patterns.

We identified various kinds of rules that are needed for OCL simplification and pointed out differences to usual term rewriting systems. We also compared template instantiation and simplification of OCL constraints to program specialization.

Moreover, we implemented a prototype of an OCL simplifier by re-using the rule application mechanism of the theorem prover in the KeY tool. We described some of the technical issues that need to be solved in such an implementation.

An important body of future work will be to add simplification rules for the various operators and data types built into OCL. In connection with this, we will need to evaluate our approach in some significant case studies. The well-studied theory of rewrite systems [4] can be applied to show termination, uniqueness of simplification results, etc. The presence of variable binding operators (`forAll`, `iterate`, etc.) also makes the work on higher-order rewriting [11] relevant in this context.

An interesting direction for future research is to perform simplification under side conditions. For instance, one might have information that is separate from an OCL constraint, but lets one decide which branch of an if-then-else construct needs to be kept. This would be useful for the work presented in [7].

We believe that future software engineering tools will in an increasing degree generate models and OCL constraints, in addition to today's manual authoring. Simplifying these specifications for improved readability will be indispensable.

Acknowledgment

The authors are thankful to Philipp Rümmer for his useful comments on a draft of this paper.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1), 2005.
2. Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, pages 389–404. Springer, September 2000.
3. Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1):17–53, 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
4. Nachum Dershowitz and David A. Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-based simplification of OCL constraints. In Octavian Patrascoiu et al., editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 84–98, 2004.
7. Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. of UML2004, Lisbon*, volume 3273 of *LNCS*, pages 197–211. Springer, 2004.
8. Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of Joint European Conferences on Theory and Practice of Software, ETAPS, Grenoble*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.
9. Kristofer Johannisson. Disambiguating implicit constructions in OCL. In *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, 2004.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
11. Tobias Nipkow and Christian Prehofer. Higher-order rewriting and equational reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations*, volume 8 of *Applied Logic Series*, pages 399–430. Kluwer, 1998.
12. J. Warmer and A. Kleppe. *The Object Constraint Language*. Object Technology. Addison-Wesley, second edition, 2003.