Rule-Based Simplification of OCL Constraints

Martin Giese, Reiner Hähnle, and Daniel Larsson

Chalmers University of Technology
School of Computer Science and Engineering
41 296 Gothenburg, Sweden
{giese|reiner|danla}@cs.chalmers.se

Abstract. To help designers in writing OCL constraints, we have to construct systems that can *generate* some of these constraints. This might be done by instantiating templates, by combining prefabricated parts, or by more general computation. Such generated specifications will often contain redundancies that reduce their readability. In this paper, we explore the possibilities of simplifying OCL formulae 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 todays UML tools allow attaching OCL constraints to diagrams and allow to check 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.

One approach [2] is to hook into the design pattern instantiation mechanism provided by various case tools.[1] The idea is to let the user instantiate templates, pieces of class diagrams, which provide implementations for various design patters. As part of the instantiation, one can generate OCL constraints that capture certain properties of the pattern.[2] This approach is going to provide the motivating example for this paper, but other scenarios are conceivable, where constraints are somehow assembled from prefabricated parts, or otherwise generated by a program, see e.g. [8, 11, 7].

In any case, the (semi-)automatically generated constraints will often contain redundancies that make them hard to read for humans. The topic of this paper

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

[2] Design patterns in the usual sense of the word [6] 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.

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. Sect. 5 presents a prototypical implementation of our ideas within the KeY system [1]. Sect. 6 explores some connections to partial evaluation and Sect. 7 discusses related work. Finally, Sect. 8 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]. (The class diagram that is part of the design pattern is itself a kind of formal specification, however, there are many things that cannot be expressed using only "core" UML but require the use of OCL.)

As software development is becoming more and more structured, taking advantage of 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.

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 the pattern gets instantiated 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.

Let us look at a concrete example to make this more clear. The intention of the *Observer* design pattern (taken from [6]), 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. The solution is to have an abstract coupling between a subject and its observers: the subject only knows that it has a list of observers, conforming to the simple interface of `Observer`. When the state of a subject changes, `notify` gets called, which in turn causes a call of the `update` method of all current observers for this object.

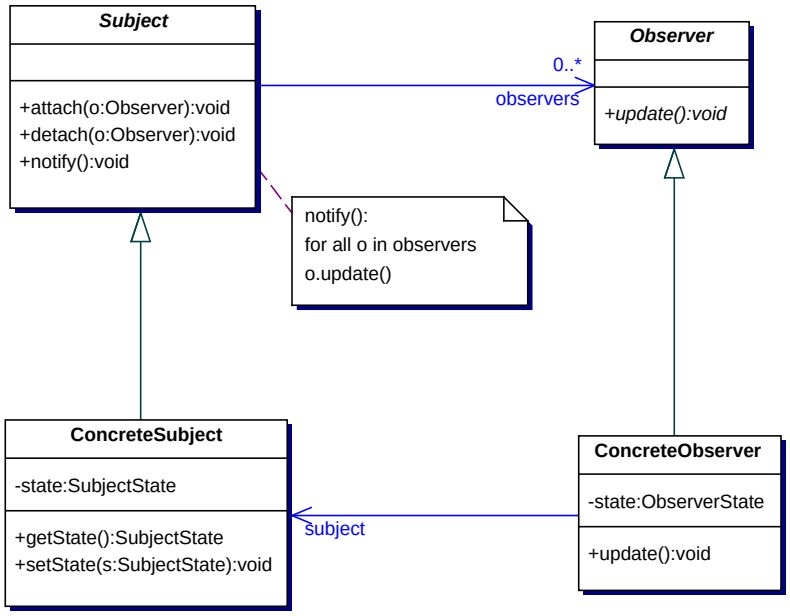

Fig. 1. The Observer pattern

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 components":

- *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.
- *Who triggers the update?* One can either let all state-setting methods in `ConcreteSubject` call `notify()` when they have changed the state, or one can leave this responsibility to the *client*. The advantage with the first approach is that clients do not have to remember to call `notify()`. The advantage with the second approach is that the client can wait to trigger the update until a number of state changes have been made, and can therefore avoid unnecessary intermediate updates.

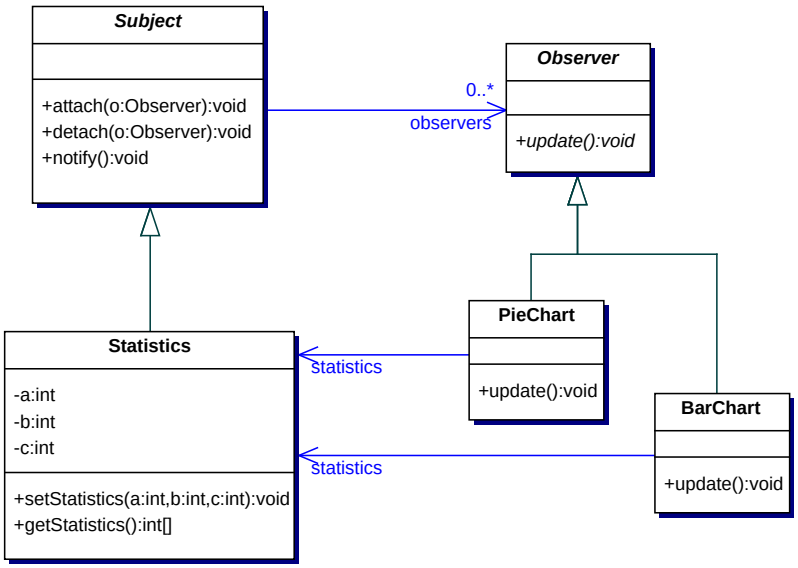

Fig. 2. An instance of the Observer pattern

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? It seems reasonable to assume that the GUI observers only need information from one `Statistics` object, i.e. the `subject` association-end has multiplicity `0..1`.

Who should trigger the update? Since in this very simple example we just have one state-setting method in `Statistics`, namely `setStatistics()`, it is probably a good idea to perform the call to `notify()` at the end of that method.

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 observer 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 the meaning.

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 sub expressions of the input until no more rules are applicable. This is a well-known principle and we can base our work on the extensive research on term rewriting systems (see e.g. [5]).

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 into this category.[3]

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.

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 `s->collect(x|x)` 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 interesting point with the collection operators in OCL is that they can all be expressed using the `iterate` construct. It might be worthwhile to 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

[3] We omit from our analysis OCL's `undefined` value that can occur in any expression. It would be perfectly possible to extend the rules, but `undefined` is rarely used in practice due to its complex semantics.

```
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. We are currently investigating an approach where the simplification would backtrack over such a transformation step if the result is not ultimately simpler than the original. This approach is different from the usual one in term rewriting [5], where one also searches for a minimal form with respect to some ordering, but where every step is required to reduce the term. If we require decreasing terms, we either need the complete set of $m \cdot n$ rules, or we need to make the form using `iterate` smaller in the term ordering. The first option would make the rule set harder to maintain and the second would be contrary to the purpose of making the simplified constraints easier to understand for humans.

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. UML diagrams provide the name space for OCL expressions, i.e. the set of available attributes, queries and role names.

OCL admits properties that are defined on the UML metamodel.[4] 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.

[4] 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.

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->count()` 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.

We have not yet looked at the final step in Sect. 3, where a single constraint is split up and distributed among the invariants of several classes. The easiest way to do this is probably to view it as a separate post-processing step. On the other hand, it would also be possible to apply our simplification rules not on raw constraints, but on lists of constraints with contexts. In that case, invariants for different classes could be generated within the same framework.

5 Implementation

We are currently in the process of implementing a rule-based OCL simplifier that will be integrated with the pattern-instantiation mechanism in KeY. It is already possible in KeY to generate OCL specifications for instances of certain design patterns, with the help of schemas. The job of the simplifier would then be to simplify the generated specifications that often contain redundant information. Now, we already have 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.

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. Of special interest for us are the so-called rewrite taclets, which can be used to define rewrite rules in a simple way. 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 arbitrary expressions. 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. 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 consists of 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, and they are only allowed to appear in the replacewith-part. They are mainly used to extract information from the UML model, e.g. the subtype 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.

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: `cons` that takes two arguments—the first element in the list and the rest of the list—and `nil` that represents the empty list.

To be more precise, we have two list constructors for each collection type: `set_cons` and `set_nil`, `bag_cons` and `bag_nil`, and so on. In that way we do not loose the type information. Using these list 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 list, 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

```
set_cons(a, set_cons(b, set_cons(c, set_nil)))
```

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

```
set_cons(a, set_cons(b, set_cons(c, set_nil)))->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:

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

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

In the examples, all schema variables are prefixed with a '`#`' sign to distinguish them from the keywords in the syntax. Here we can also see the syntax for *substitution*, `{x e}exp`, which means that all free occurrences of `x` in `exp` are replaced with `e`. Here are some more examples of taclets needed for OCL simplification:

```
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)}
```

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.

In order to perform OCL simplification using taclets we have to extend the implementation of the KeY tool, and this is work in progress. The current version can, 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 bound variables, which is needed to handle `forAll`, `iterate`, etc. However, a number of things remain to be done:

- Implement pretty-printing of OCL expressions.
- Read the input using an OCL parser. The OCL expressions to be simplified need to be parsed and type-checked. A new OCL parser that can generate a suitable abstract syntax tree is currently under development at Chalmers University, see [9].
- Integrate it with the pattern mechanism in the KeY tool.
- Handle types. So far we ignore OCL's type system.
- Complete our set of taclets. So far we have only written a few taclets to be able to simplify some test expressions.

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 Simplification and Partial Evaluation

OCL is nearer to a programming language than most other specification formalisms, like for instance classical first order logic (there are even considerations for a directly executable version). The point is that OCL constraints can (at least in principle) be effectively evaluated, given a snapshot of the system in question. This is mostly due to the fact that there is no quantification over infinite sets in OCL. The collection data types allow only finite collections, and any iteration or quantification ranges over some finite set.

If one interprets OCL constraints as executable programs, then the process of first instantiating parts of an OCL schema and then simplifying the result becomes an instance of what is known as *program specialization* or *partial evaluation* (see e.g. [10]). This connection suggests that results and techniques from partial evaluation might be adapted and used in our context.

There are, on the other hand, also significant differences between the two topics. First, there are no potentially non-terminating constructs, like while-loops or recursion, in OCL.[5] This eliminates many of the more difficult aspects of partial evaluation. Second, we want to simplify OCL constraints not for efficiency or space reasons, but to improve their readability for humans. This distinction can make a big difference in practice.

There are nonetheless several techniques known from partial evaluation that we eventually want to apply in our OCL specializer:

- For both partial evaluation and human-oriented OCL simplification, there are cases where it is hard for a computer program to decide whether a certain rule should be applied or not. In this case, a semi-automatic solution might be sensible, where the user is queried for some of the applicable rules.
- One technique employed in partial evaluation is to do a *binding time analysis* before any actual simplification. This analysis identifies expressions and programs that can be statically evaluated if certain arguments are fixed. Among other purposes, this information can help in writing programs in a way that makes it easy to specialize. The same approach would certainly be helpful for OCL specialization. One could for example envisage an editor which highlights constructs that can be evaluated statically.

It remains to be seen which other results from the research on partial evaluation can be applied to a specification language like OCL.

7 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

[5] Recursive definitions have recently been proposed as an extension, but we will ignore this here.

implementation approach. Closest to our work are the *specification patterns* for temporal properties in the *Bandera* project [4]. These are essentially grammars for structured English that are designed in such a way that they can be directly translated into low-level logical expressions. In contrast to this grammar-based approach, one could call our approach language-based, because it uses operations, parameters, and (in the near futures) types.

8 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.

In the near future, we plan to extend the implementation to make it aware of OCL's type system. We also want to connect it to an OCL parser and integrate it with the pattern mechanism in KeY. In a longer perspective we want to look into the area of partial evaluation and see which techniques from that area might be applicable.

Another 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 interesting for the work presented in [7].

We believe that future tools for formal methods will need to help developers by generating formal specifications from other information. Simplifying these specifications for improved readability will be indispensable.

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 System Modeling*, 2004. To appear.

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 writing theorem provers. *Revista De La Real Academia De Ciencias Exactas, Fisicas Y Naturales*, 2004. To appear.
4. James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In Klaus Havelund, John Penix, and Willem Visser, editors, *7th International SPIN Workshop Stanford*, volume 1885 of *LNCS*. Springer, 2000.
5. 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.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In Thomas Baar and Alfred Strohmeier, editors, *Proc. of UML2004, Lisbon*. 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. Boris Roussev. Generating OCL specifications and class diagrams from use cases: A newtonian approach. In *36th Annual Hawaii International Conference on System Sciences*, page 321b, 2003.
12. J. Warmer and A. Kleppe. *The Object Constraint Language*. Object Technology. Addison-Wesley, second edition, 2003.