Validating UML Models and OCL Constraints

Mark Richters and Martin Gogolla

University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
{mr|gogolla}@informatik.uni-bremen.de,
WWW home page: http://www.db.informatik.uni-bremen.de

Abstract. The UML has been widely accepted as a standard for modeling software systems and is supported by a great number of CASE tools. However, UML tools often provide only little support for validating models early during the design stage. Also, there is generally no substantial support for constraints written in the Object Constraint Language (OCL). We present an approach for the validation of UML models and OCL constraints that is based on animation. The USE tool (UML-based Specification Environment) supports developers in this process. It has an animator for simulating UML models and an OCL interpreter for constraint checking. Snapshots of a running system can be created, inspected, and checked for conformance with the model. As a special case study, we have applied the tool to parts of the UML 1.3 metamodel and its well-formedness rules. The tool enabled a thorough and systematic check of the OCL well-formedness rules in the UML standard.

1 Introduction

The Unified Modeling Language (UML) [9] has been widely accepted as a standard for modeling software systems. A great number of CASE tools exists which facilitate drawing and documentation of UML diagrams. Many of the tools also offer automatic code generation and reverse engineering of existing software systems. However, often there is only little support for validating models during the design stage. Also, there is generally no substantial support for constraints written in the Object Constraint Language (OCL) [8, 13]. While it seems feasible to translate constraints into program code as part of the code generation process, we argue that a model and its constraints should be validated before coding starts. Mistakes in the design can thus be detected very early, and they can easily be corrected in time.

In this paper, we present an approach for the validation of UML models and OCL constraints that is based on animation. We have built a tool called USE (UML-based Specification Environment) for supporting developers in this process. The main components of this tool are an animator for simulating UML models and an OCL interpreter for constraint checking. A UML model is taken as a description of a system. System states are snapshots of a running system. They can be manipulated, inspected, and checked for conformance with the model. The tool implements and continues ideas and results from our previous work on formalizing the OCL and introducing a metamodel for OCL [11, 12].

Our validation tool can be generally applied to models from any domain. As a special case, we have applied it to parts of the UML 1.3 metamodel and its well-formedness rules. This is the first time (at least to our knowledge) that a tool enabled a thorough and systematic check of the OCL well-formedness rules in the UML standard. This also opens up a number of other useful applications. For example, the well-formedness of arbitrary models with respect to the UML standard can be automatically checked by validating them as instances of the UML metamodel.

There are currently only a few tools available which are specifically designed for analyzing UML models and OCL constraints. Probably, this is mostly due to the lack of a precise semantics of UML and OCL. A well-defined semantics is a prerequisite for building tools offering sophisticated analysis features. Work on precise semantics has been carried out in, e.g., [3, 11]. The following summarizes some work related to tools for checking UML designs. Alcoa is a tool for analyzing object models [5]. It does not use OCL but has its own input language, Alloy, which is based on Z. RTUML [7] focuses on real-time modeling aspects and offers a methodology for mechanized verification of design properties with PVS. An OCL compiler and code generator combined with an OCL runtime library implemented in Java has recently been developed at the Dresden University [4]. A beta release of a commercial tool offering animation of UML models and OCL support is ModelRun by BoldSoft [1].

The rest of this paper is structured as follows. In Sect. 2 we describe our approach to validating UML and OCL. Section 3 gives an overview of the USE architecture. A case study is used in Sect. 4 to demonstrate the key features of our tool with respect to the validation process. In Sect. 5, we report on applying the USE tool to the Core package of the UML metamodel. We close with a summary and draw some conclusions for future work.

2 The USE Approach to Validation

The goal of model validation is to achieve a good design before implementation starts. There are many different approaches to validation: simulation, rapid prototyping, etc. In this context, we consider validation by generating snapshots as prototypical instances of a model and comparing them against the specified model. This approach requires very little effort from developers since models can be directly used as input for validation. Moreover, snapshots provide immediate feedback and can be visualized using the standard notation of UML object diagrams – a notation most developers are familiar with.

The result of validating a model can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not fulfill the constraints, this may indicate that the constraints are too strong or the model is not adequate in general. Therefore, the design must be revisited, e.g., by relaxing the constraints to include these cases. On the other hand, constraints may be too weak, therefore allowing undesirable system states. In this case, the constraints must be changed to be more restrictive. Still, one has to be careful about the fact that a situation in which undesirable snapshots are detected during validation and desired snapshots pass all constraints does not allow a general statement about the correctness of a specification in a formal sense. It only says that the model is correct with respect to the analyzed system states. However, some advantages of validation in contrast to a formal verification are the possibility to validate non-formal requirements, and that it can easily be applied by average modelers without training in formal methods.

The diagram in Fig. 1 illustrates the basic use cases for validating a model with USE. First, a model specification can be checked by the validation system. The *check specification* use case includes a syntax, type and semantic check. The syntax check verifies a specification against the grammar of the specification language which is basically a superset of the OCL grammar defined in [8, 13] extended with language constructs for defining the structure of a model. The type check makes sure that every OCL expression can be correctly typed. Finally, a semantic check verifies a number of context-sensitive conditions. Among these conditions are the well-formedness rules defined as part of the UML Semantics [10]. An example for such a well-formedness rule is the requirement that a generalization hierarchy must not contain cycles.

When a specification has passed all checks, a developer may start producing and *changing system states*. A system state can be changed by issuing commands for creating and destroying objects, inserting and removing links between objects, and setting attribute values of objects. The developer can *check a system state* at any time. A system state check includes two phases. First, all model-inherent constraints must be verified. A model-inherent constraint is a constraint which is inherent to the semantics of all UML models. For example, the set of links between objects is verified against the multiplicity specifications of the association ends. The number of objects participating in an association must conform to the multiplicities defined at the association ends. Second, if the developer has defined explicit OCL constraints, all the constraint expressions are evaluated. If any of the constraints is false or has an undefined result, the system state is considered illegal.

The *inspect system state* use case describes facilities for getting information about a system state. This is very important for helping a user to understand the effects of commands resulting in system state changes. Furthermore, when a constraint fails and a system state is found to be invalid, the developer has to find the reason for the failure. Inspecting a system state involves the inspection of individual objects, their attribute values and links. Another powerful way for inspection is the use of OCL as a query language. For example, consider a model where each object of class A must have at least one link to an object of class B, i.e., the association end at class B has multiplicity `1..*`. If this multiplicity constraint is violated in some system state, the objects of class A which do *not* have a link to a B object can easily be found by the expression `A.allInstances->select(a | a.b->size = 0)`.

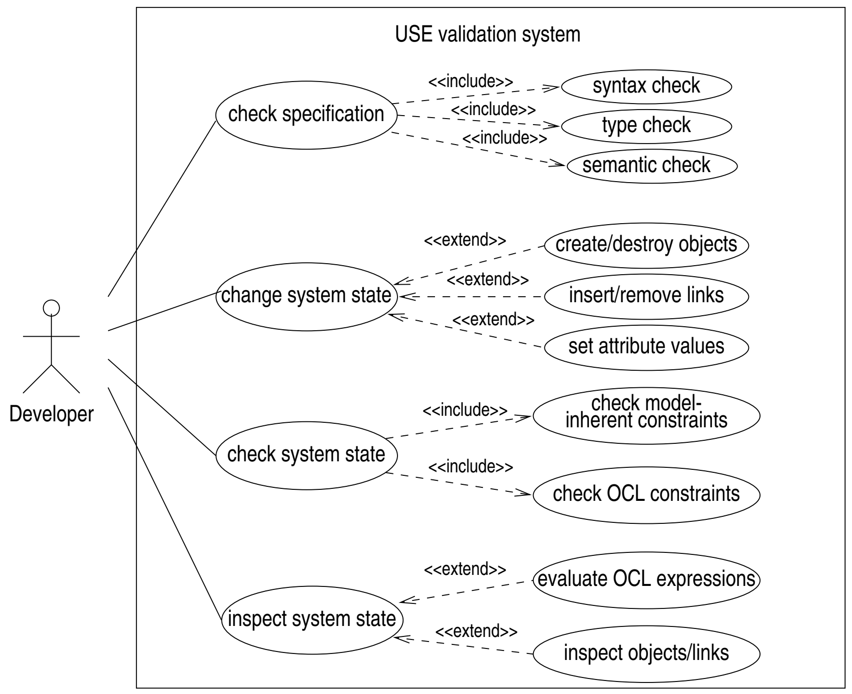


Fig. 1. Use case diagram showing basic functionality of USE

3 Architecture of USE

A high-level overview of the USE architecture is given in Fig. 2. We distinguish between a *Description Layer* at the top, and an *Interaction Layer* below. The description layer is responsible for processing a model specification. The main component is a *Parser* for reading *Specifications* in USE syntax and generating an abstract syntax representation of a model. A USE specification defines the structural building blocks of a model like classes and associations. Furthermore, OCL expressions may be used to define constraints and operations without side-effects.

The output of the parser is an abstract representation of a specification containing a *Model* and *OCL* expressions. The representation of the model is done with a subset of the Core package of the UML metamodel [10, p. 2-13]. The subset excludes all model elements which are not required during the analysis and early design phase of the software development process. For example, model elements like Permission, Component, and Node seem to be more adequately applied in extended design and implementation models. The abstract representation of OCL expressions closely follows the OCL metamodel we have presented in [12].

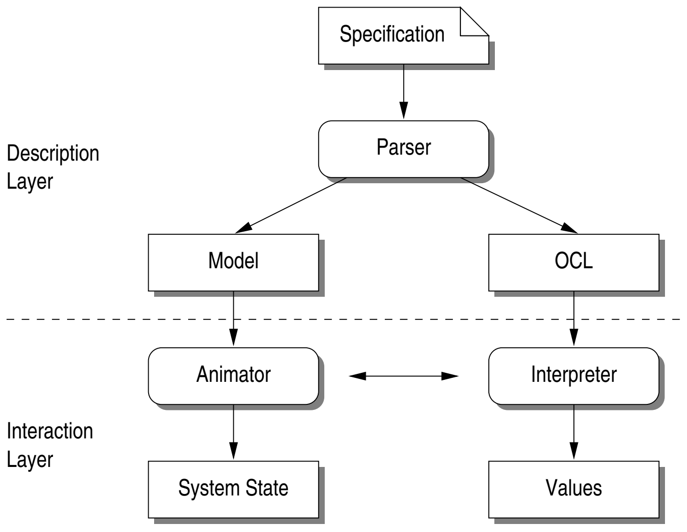


Fig. 2. Overview of the USE architecture

The *Interaction Layer* provides access to the dynamic behavior and static properties of a model. The main task of the *Animator* component is the instantiation and manipulation of *System States*. A system state is a snapshot of the specified system at a particular point in time. The system state contains a set of objects and a set of association links connecting objects. As a system evolves, a sequence of system states is produced. Each system state must be well-formed, i.e., it must conform to the model's structural description, and it must fulfill all OCL constraints. Furthermore, a transition from one system state to the next must conform to the dynamic behavior specification. Specifying and checking state transitions is not yet available in USE.

The *Interpreter* component is responsible for evaluating OCL expressions. An expression may be part of a constraint restricting the set of possible system states. In order to validate a system state, the animator component delegates the task of evaluating all constraints to the interpreter. The interpreter is also used for querying a system state. A user may query a system state by issuing expressions that help inspecting the set of currently existing objects and their properties.

The Model and OCL branches in Fig. 2 are tightly related to each other. For example, a model depends on OCL since operations of classes defined in a model may use OCL expressions in their bodies. A dependency in the other direction exists because the context of OCL constraints is given by model elements. However, it is in general possible to define models which do not use OCL at all, or there may be OCL expressions which do not require a user model. For example, the realization of various general purpose algorithms with OCL (like sorting, determining the transitive closure of a relation [6], etc.) is an interesting task on its own and can be done without the need for any particular model.

Animator and interpreter closely work together. The animator asks the interpreter for evaluating OCL expressions. On the other hand, the Interpreter needs information about the current system state, e.g., when evaluating an expression which refers to the attribute value of an object.

4 Example Case Study

In this section, we will demonstrate the validation of a UML model by means of a small case study. We will start with presenting a class diagram of a company model together with a few constraints. The model will then be specified in the textual USE notation. This specification serves as input to the validation tool. In an interactive session, a sequence of system states will be produced by creating objects and links between them. Finally, we will check a system state against the specification and show how the tool supports exploring the system state and helps in finding the reason for a constraint violation.

Figure 3 shows a UML class diagram of our example model. Employees have name and salary attributes and work in departments. A department controls projects on which any number of employees can work. Both department and projects have attributes specifying the available budget.

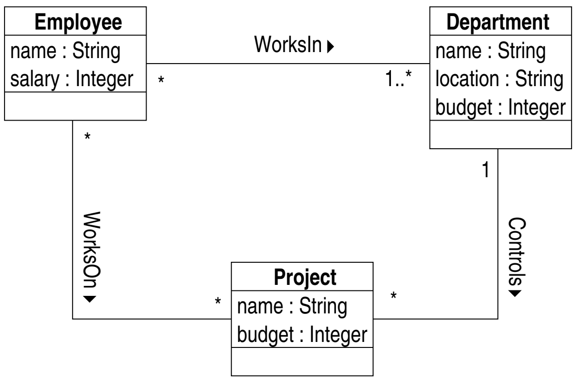


Fig. 3. Class diagram of example model

The result of translating the class diagram into the textual USE notation is shown in Fig. 4. The specification contains definitions for each of the classes and associations. The definition of a class includes its attributes, an association defines references to the participating classes for each association end. Multiplicity ranges are specified in brackets. Not used in this example but also supported by the USE language are UML features like generalization, operations, different association types, role names, etc.

```
model Company

class Employee
attributes
  name : String;
  salary : Integer;
end

class Department
attributes
  name : String;
  location : String;
  budget : Integer;
end

class Project
attributes
  name : String;
  budget : Integer;
end

association WorksIn between
  Employee[*];
  Department[1..*];
end

association WorksOn between
  Employee[*];
  Project[*];
end

association Controls between
  Department[1];
  Project[*];
end
```

Fig. 4. USE specification of the example model

In order to make the example more interesting, we add some constraints which cannot be expressed graphically with the class diagram. The following five conditions have to be satisfied by a system implementing the given model.

[1] The salary and budget attributes are always positive.
[2] A department has at least as many employees as projects.
[3] An employee working on more projects than another employee gets a higher salary.
[4] The budget of a project must not exceed the budget of the controlling department.
[5] Employees working on a project must work in the controlling department.

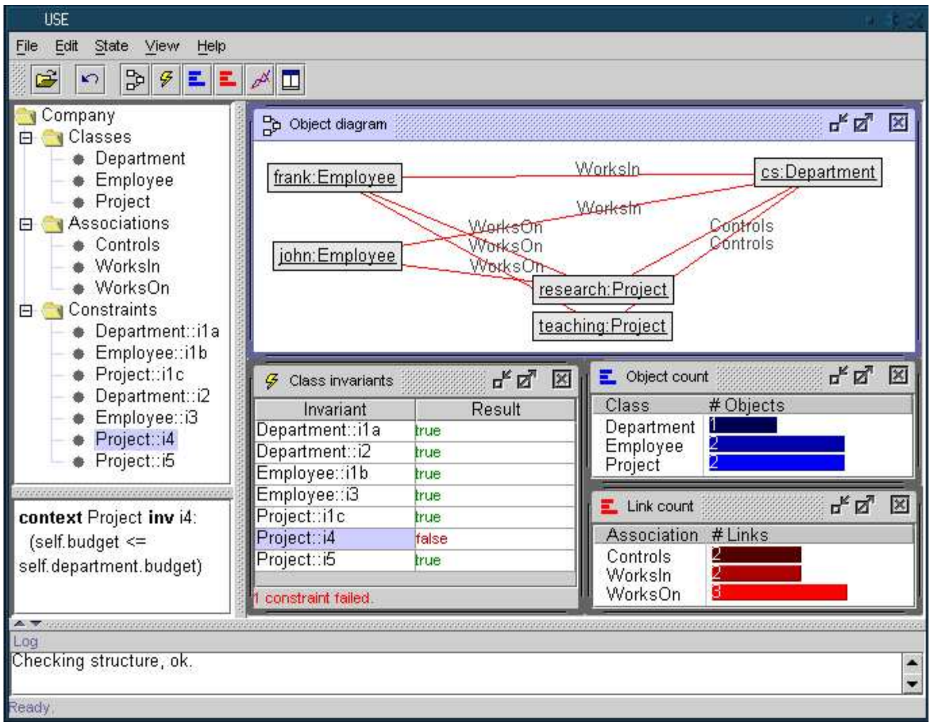
For each of these constraints we have specified OCL expressions that are used as invariants on the classes. We continue the specification begun in Fig. 4 with a section defining the set of constraints shown in Fig. 5. Each invariant is named for allowing an easy reference to the list above. Note that constraint [1] actually maps to three OCL invariants (i1a, i1b, i1c) since it states a condition on each of the three classes.

We can run the USE tool with the specification and start with an empty system state where no objects and no association links exist. As a next step, we are going to populate the system with objects and link them together. There are three kinds of commands which allow us to modify a system state: (1) creating and destroying objects, (2) changing attribute values, and (3) inserting and

```
constraints

context Department
   inv i1a: self.budget >= 0
   inv i2: self.employee→size >= self.project→size

context Employee
   inv i1b: self.salary >= 0
   inv i3: Employee.allInstances→forAll(e1, e2 |
            e1.project→size > e2.project→size implies e1.salary > e2.salary)

context Project
   inv i1c: self.budget >= 0
   inv i4: self.budget <= self.department.budget
   inv i5: self.department.employee→includesAll(self.employee)
```

Fig. 5. USE specification of OCL constraints

deleting association links. Figure 6 shows a screenshot of USE visualizing a system state after several objects and links have been created.



context Project inv i4:
(self.budget <=
self.department.budget)

Invariant	Result
Department::i1a	true
Department::i2	true
Employee::i1b	true
Employee::i3	true
Project::i1c	true
Project::i4	false
Project::i5	true

1 constraint failed.

Class	# Objects
Department	1
Employee	2
Project	2

Association	# Links
Controls	2
WorksIn	2
WorksOn	3

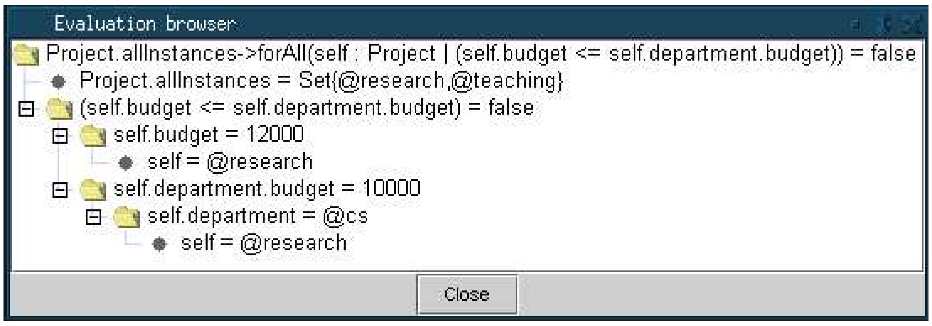
```
Log
Checking structure, ok.
Ready.
```

Fig. 6. USE screenshot

On the left side, the user interface provides a tree view of the classes, associations, and constraints in the model. The pane below shows the definition of the currently selected component (the invariant `Project::i4`). The pane on the right contains several different views of the current system state. These views are automatically updated as the system changes. A user can choose from a number of different available views each focusing on a special aspect of a system state. In this example, there are views showing an object diagram, a list of class invariants with their results, and two views displaying the number of objects and links.

The highlighted constraint `Project::i4` in the class invariant view has the result false indicating that the system state does not conform to the specification. The plain information that an invariant has been violated is usually not very helpful in finding the reason for the problem. The invariant from the example is quite short (see Fig. 5), and we can infer from the specification that there must be at least one project with a budget exceeding the budget of its controlling department. We could proceed by inspecting all projects until we find one violating the constraint. For larger systems, this quickly becomes a laborious task. Our tool therefore offers special support for analyzing OCL expressions.

The details of evaluating an OCL expression can be examined by means of an evaluation browser which provides a tree view of the evaluation process. Figure 7 displays such a browser for the failing invariant. The root node shows the complete OCL expression defined as the body of the invariant (actually, the original expression is first expanded into a self-contained expression which does not require a context). Child nodes represent sub-expressions which are part of their parent node's expression. Evaluating the `forAll` expression requires the evaluation of the source collection (`Project.allInstances`) and the argument expression (`self.budget <= self.department.budget`). By looking at the current binding of `self`, we can conclude that it is the budget of the "research" project which invalidates the whole invariant.



```
Project.allInstances->forAll(self : Project | (self.budget <= self.department.budget)) = false
  Project.allInstances = Set{@research,@teaching}
  (self.budget <= self.department.budget) = false
    self.budget = 12000
      self = @research
    self.department.budget = 10000
      self.department = @cs
        self = @research
```

Fig. 7. OCL evaluation browser

5 Validating the UML Metamodel

The USE validation tool can be generally applied to models from any domain. As a special case, we have applied it to the Core package of the UML 1.3 metamodel and its well-formedness rules. The tool enabled a systematic check of the OCL well-formedness rules in the UML standard. This section describes the procedure for checking the metamodel and some results.

In the first step, we had to translate the class diagrams defining the UML Core ([10, Sect. 2.5.2]) into the textual USE notation. Next, all well-formedness rules as well as additional operations in Sect. 2.5.3 of [10] were added. Some minor syntactical changes required by the USE syntax were necessary following these rules:

1. If an association does not have a name, add one.
2. Append an underscore to identifiers which are reserved keywords in USE (e.g., `association`).
3. Append a pair of parentheses '()' to calls of additional (user-defined) operations when they have no arguments.
4. Replace all implicit occurrences of collect by an explicit invocation.
5. Replace all implicit occurrences of collection flattening by using a predefined operation `flatten`.
6. Replace occurrences of Boolean enumeration types with the OCL type Boolean.

The final specification[1] has 31 classes and 24 associations. There are 43 well-formedness rules and 28 additional operations resulting in a total of 71 OCL expressions. The expressions in the UML document had a number of errors which could quickly be located by analyzing the error messages signaled by the USE parser. Some errors could easily be corrected, others indicated more serious problems with the constraints. We classified the problems into the following categories (with increasing severity) and give an example for each category. A class name together with a number in brackets refers to the respective well-formedness rule in [10].

E1: *Syntax errors*
- Example: wrong spelling of keywords and standard operation names (`Association[3]`, `AssociationEnd[1]`)

E2: *Minor inconsistencies*
- Example: there is no operation `max` defined on Multiplicity (`AssociationEnd[2]`)

E3: *Type checking errors*
- Example: union of sets with incompatible element types (`Classifier[4]`, `Classifier[5]`)
- Example: implicit collect expression returns a bag not a set (`ModelElement::supplier()`)

[1] Available at `http://www.db.informatik.uni-bremen.de/~mr`

E4: *General problems*
  – Example: the operation `contents()` in class Namespace has syntax errors and an identical description as `allContents()`. It remains unclear how these operations should look like.

The results from analyzing the OCL expressions are summarized in Table 1. We found that there were errors in 39 out of 71 expressions. Some expressions contained two or more errors belonging to different categories. Approximately, every second erroneous expression had errors of category E1 which could be fixed without much effort. The other errors generally required more work and detailed knowledge of the metamodel.

Table 1. Results from analyzing OCL expressions in the UML Core

	Classes	Associations	Invariants	Operations	Errors	E1	E2	E3	E4
Count	31	24	43	28	39	20	7	13	5

It is not very surprising that a tool-based mechanical check of OCL expressions greatly helps in finding frequently occurring errors such as spelling mistakes. The fact that OCL provides strong typing also helps in getting complex expressions right. Another general observation that we have made is related to the style of the OCL syntax. In some cases, a single notation is used for many different things. This makes it sometimes quite difficult to understand an expression and requires a lot of context knowledge. From a human's point of view this complicated the task of reading, understanding and checking OCL expressions. Consider, for example, the definition of the operation allParents in class GeneralizableElement:

```
allParents : Set(GeneralizableElement);
allParents = self.parent->union(self.parent.allParents)
```

The syntax of the expression `self.parent` is the same for referring to an attribute, an operation, or a role name of an associated class. Furthermore, `parent.allParents` may again be an attribute reference, an operation call, or a navigation by role name. Additionally, it may be an implicit collect expression written in shorthand notation. To find out which case is actually present, one has to look at the attributes, the operations and associations of all the referenced classes. However, this is still not enough since all these features might be defined in superclasses so that the generalization hierarchy also has to be taken into account. We therefore "re-engineered" most expressions to an explicit form making the intended meaning much clearer. The example from above was augmented with an explicit collect, a flattening operation, and a conversion of the result to a set which is required by the declaration.

```
allParents = self.parent()->union(
  self.parent()->collect(g | g.allParents())->flatten)->asSet
```

We found the USE tool to be very beneficial for understanding and analyzing the well-formedness rules of the UML metamodel. A number of errors in the OCL expressions could be quickly located and corrected. For future work, we plan to extend the analysis to the complete UML metamodel including all of its well-formedness rules and making it available in USE. This might not only be useful for improving the state of the standard but also implies another very nice application: in principle, any UML model can then be checked for conformance to the UML standard. A model conforms to the UML standard if it can be represented as an instance of the UML metamodel. The general idea is to (1) import a UML model (preferably in XMI representation), (2) traverse the model and execute a sequence of USE commands for instantiating the model elements as objects of the UML metamodel, and (3) check all constraints on the resulting snapshot. All these steps can be done mechanically. If the last step fails, the model is not conform to the UML standard.

6 Conclusion

In this paper, we have presented a tool-based approach to validating UML models and OCL constraints. The ideas presented here have been implemented in the USE tool. The functionality of USE has been shown by means of use cases and a small example case study. We have also applied the tool for checking a part of the UML metamodel which makes extensive use of OCL constraints. As a result we could identify a number of errors in the standard document. Using the metamodel as a specification of arbitrary UML models, the tool enables a mechanical check for conformance of these models with the standard.

The OCL parser and interpreter that is part of USE implements most of the core features of OCL like expression syntax, strong type checking, and evaluation of expressions. We have implemented almost all of the more than 100 standard operations on predefined OCL types. Features we are currently working on include the syntax of path expressions and qualifiers, type checking of empty collection literals, and syntax and semantics of pre- and postconditions. For validating postconditions it might also be desirable to have some kind of an action language to specify side effects of operations.

There are several possible future extensions which would fit within the USE framework. First, it would be nice if the animation could be automated to some extent by deriving test cases from the model. Another extension could apply the validation techniques of USE to implementations of a model. Program code could be generated which mirrors the state of a system at runtime. The state traces can be observed and analyzed in parallel with USE. With this approach there is no need for transforming OCL expressions into program code since the interpretation of expressions is already part of USE. Also very useful would be an analysis of OCL constraints with respect to properties like consistency. However, there is currently no clear definition of what it means for a set of OCL constraints to be consistent. A discussion of this can be found in the OCL Semantics FAQ [2].

Acknowledgments

We thank Frank Finger, Heinrich Hussmann, and Jos Warmer for fruitful discussions.

References

1. BoldSoft. Modelrun, 2000. Internet: `http://www.boldsoft.com/products/modelrun/index.html`.
2. Tony Clark, Stuart Kent, Jos Warmer, et al. OCL Semantics FAQ, Workshop on the Object Constraint Language (OCL) Computing Laboratory, University of Kent, Canterbury, UK. Internet: `http://www.cs.ukc.ac.uk/research/sse/oclws2k/index.html`, March 2000.
3. Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 140–155. Springer, 1999.
4. Frank Finger. Design and implementation of a modular OCL compiler. Diplomarbeit, Dresden University of Technology, Department of Computer Science, Software Engineering Group, Germany, March 2000.
5. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proc. International Conference on Software Engineering, Limerick, Ireland, June 2000*, 2000. (to appear).
6. Luis Mandel and María Victoria Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.
7. Darmalingum Muthiayen. *Real-Time Reactive System Development - A Formal Approach Based on UML and PVS.* PhD thesis, Department of Computer Science at Concordia University, Montreal, Canada, January 2000.
8. OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [9], chapter 7.
9. OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999.* Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1999.
10. OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [9], chapter 2.
11. Mark Richters and Martin Gogolla. On formalizing the UML Object Constraint Language OCL. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.
12. Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 156–171. Springer, 1999.
13. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.