Modular Architecture for a Toolset Supporting OCL

Heinrich Hussmann, Birgit Demuth, and Frank Finger

Dresden University of Technology, Department of Computer Science

Abstract. The practical application of the Object Constraint Language, which is part of the UML specification since version 1.1, depends crucially on the existence of adequate tool support. This paper discusses general design issues for OCL tools. It is argued that the nature of OCL will lead to a large variety of tools, applied in combination with a variety of different UML tools. Therefore, a flexible modular architecture for a UML/OCL toolset is proposed. The paper reports on the first results of an ongoing project which aims at the provision of such an OCL toolset for the public domain.

1 Introduction

Since version 1.1, the UML standard comprises a formal annotation language for UML models, the Object Constraint Language (OCL) [19]. This language is used in the UML standard for precisely defining the well-formedness rules of UML models on the metamodel level. Moreover, OCL is currently gaining popularity in the definition of other OMG standards. Besides this usage on a meta-level there is also high potential in using OCL in the actual development process of software to improve software quality.

- In the analysis phase, business rules can be expressed precisely. Usually, OCL invariants are attached to class diagrams and enable the specification of constraints which go far beyond the possibilities of "plain" UML (i.e. UML without OCL). Some of the most useful constructs of plain UML for the analysis phase can be seen just as abbreviations for simple OCL invariants (e.g. the construct of association multiplicities), and as soon as more complex constraints appear, OCL is the language of choice.
- In the design phase (mainly), constraints can be used to precisely specify pre- and postconditions for operations, and therefore provide a precise contract [13] for the implementor and user of the operations.
- In development tools, OCL can be used as a simple query and navigation language.
- At various other places in UML models, object constraints can be used for preciseness, e.g. in guards of statechart diagrams.

An example for a full UML-based development method which incorporates OCL usage is Catalysis [10].

Recently, at several places experiments have been started to introduce OCL into the practical object-oriented software development process (e.g. [2]). However, all such attempts are facing the problem that there is a definite lack of tool support for OCL. With a few exceptions [5], the tool industry seems to ignore OCL. There are at least two good reasons for this lack of commercial support: First, it has still to be proven that the theoretical potential of OCL leads to practical improvements in real software projects. Second, the needed functionality of OCL support tools is still rather unclear. So there is a need for significant further research before OCL can achieve a status of broad market acceptance and commercial tool support.

In this paper, we report on an attempt to enable serious practical experiments with OCL. We describe a software platform for OCL tool support which is designed for openness and modularity, and which is provided as OpenSource. The goal of this platform is to enable practical experiments with various variants of OCL tool support. In this paper, we analyze the requirements for this tool platform and describe the key design decisions.

This paper is structured as follows: In section 2, an overview of the full range of possibilities for OCL tool support is given, setting the scene for the analysis of the requirements for the tool platform. Section 3 describes the actual architecture chosen, based on key requirements derived from section 2. Several examples for possible configurations of the tool platform are given, and the current status of the implementation is described. Section 4, finally, summarizes our results and gives ideas for further investigations and projects.

2 Potential for OCL Tool Support

The specification of OCL constraints enhancing a UML model causes a significant amount of additional effort, so the crucial question is an economic one: How can we ensure that the additional development effort spent on adding all this detailed information really pays back? From this perspective, several kinds of tools are required:

- Tools that use the high precision present in OCL-based specifications for a thorough analysis of the UML/OCL model.
- Tools that help the modeler to ensure that the actual constraints together with other UML diagrams make up a sensible model of the problem domain.
- Tools that reduce the development, testing and maintenance effort by making use of the information given in OCL.
- Tools that enable a higher level of trust in the implementation of an OCL specification, and therefore are suitable e.g. for development of safety-critical applications.

When discussing tool support for OCL, an important difference between OCL and many other formal specification languages has to be pointed out: OCL constraints are *executable* in the sense that they can be evaluated mechanically for a given snapshot of a universe of objects. Nevertheless, OCL contains all language

constructs of classical first-order logic, like the classical quantifiers "forAll" and "exists", and these quantifiers can be applied in arbitrary nesting. In contrast to general predicate logic, however, OCL always ensures that these constructs are applied only to a finite set, so they can be checked mechanically by enumeration of the set. This property of OCL was less clear in early versions of the UML specification (due to constructs like `Integer.allInstances`). However, for UML 1.3 several changes have been applied which prohibit the usage of infinite sets, and which even ensure that the evaluation of a constraint on a snapshot always terminates. The executability of OCL is a key feature for effective and simple tool support.

Let us briefly discuss the most important kinds of tools supporting OCL.

Syntactical analysis The simplest form of support is of course parsing OCL expressions. This form of tool support, however, is able to find only very basic OCL errors.

Typechecking After some amendments to the standard, there is now a relatively stable type system for OCL which enables mechanical static typechecking of OCL constraints, much in the same way as typechecking of a typed programming language. However, OCL constraints always make reference to an underlying class model. So an OCL typechecker has to have access to the underlying UML model information. Practical experience shows that this relatively simple tool feature already significantly improves the quality of OCL specifications.

Logical consistency checking Since OCL is a logic language, it is possible to write down sets of constraints which are contradictory in themselves. Although it would be very helpful to have a tool checking statically for such inconsistencies, there are many open questions still to be solved before such tools can be built. In general, it is even likely that consistency is undecidable, so one may want to develop appropriate criteria for decidable subcases. Moreover, links to recent research prototypes for symbolic constraint solving (e.g. [9]) may lead to interesting results.

Dynamic invariant validation From the point of view of the tool builder, an invariant is a Boolean-valued function which can be evaluated on a given snapshot of the system. So it makes sense to provide tools which actually do this evaluation during the evolution of a system. There are several approaches to such a dynamic verification: To integrate assertion tests into standard generation of code (fragments); to generate specific code doing automatic invariant checking, as it is possible for database integrity constraints [6]; or to provide a simulation tool for construction of sample system states [5], [17].

Dynamic pre/postcondition validation In a very similar way to the treatment of invariants, pre- and postconditions can be evaluated at runtime of a system constructed from the specification. In this case, the most appropriate approach seems to be to integrate assertion statements into the generation of code (fragments).

Test automation An automated test tool may make use of the OCL pre- and postconditions to achieve an automatic check of test results against the specification. Moreover, it may be possible to derive test cases from an analysis

of the functional specification written in OCL. This class of tools may be the most interesting one from an economic point of view, since it helps to save some of the usually high costs for quality assurance. Also maintenance costs can be reduced by such tools since automatic regression testing against the formal specification is enabled, and the semantic consistency of specification and code is enforced.

Code verification and synthesis In the long run, OCL may also form a basis for code verification and synthesis. First projects in this direction have been started (e.g. the Karlsruhe KeY project [8]), but this is clearly the most ambitious kind of tool which may be appropriate only for special, safety-critical development projects.

From the list above, it becomes obvious that there is a large variety of different tools, all of which rely on a rather small common functionality. This observation is reflected in the toolset architecture described below.

3 Toolset Architecture

3.1 Requirements

From an analysis of potential OCL tools, as summarized above, a number of requirements can be derived:

Requirement 1: The architecture shall enable interworking with various CASE tools and repositories, regarding the access to model information for type-checking. A simple and flexible interface is required which supports the construction of stand-alone experimental tools (working e.g. on a file representation of the model) as well as a tight integration into CASE tools, for more user-friendly versions of tools.

Requirement 2: Syntax analysis and type checking of OCL constraints is the functionality which is common to all tool variants. So a simple interface to this functionality is needed in order to enable integration into various OCL tools.

Requirement 3: The tool platform has to provide a simple and easily reusable interface for accessing the actual constraint information (the abstract syntax of the constraints) from different kinds of tools. Ultimately, the solution should be compliant with an OCL metamodel [16], which is still under discussion.

Requirement 4: Different tools want different levels of abstraction in accessing the representation of OCL constraints. For example, a tool generating programming language code may want to expand automatically all `select` operations into the generic `iterate` mechanism. In contrast, a tool generating SQL integrity conditions may want to keep the `select` operations since they can be mapped easily and directly to SQL [6].

3.2 Key Design Decisions

In the following, a modular architecture for an OCL toolset satisfying the above listed requirements is presented. The architecture is designed based on our experience with a prototype implementation [14]. We decided to develop the OCL toolset in Java because of the high popularity of Java as implementation language in the Open Source Community and therewith the availability of useful tools like parser generators and the possibility to integrate the OCL toolset with free CASE tools such as Argo/UML.

A further decision was that an OCL toolset should be fully compliant to the UML/OCL specification version 1.3 [15]. Unfortunately, this specification contains some ambiguities and contradictions. These have to be solved in a sensible way, and aspects where the implementation deviates from the specification have to be documented in detail.

The first requirement from above means that a toolset should be adaptable to different environments. This adaptability can be achieved if all external interfaces are designed and documented carefully. Dependencies between the OCL toolset and the environment have to be minimized.

With the adaptibility comes the extensibility of the design. As discussed in section 2, a large variety of OCL tools is imaginable that the toolset architecture should support. While it is very hard or even impossible to achieve this fully for very sophisticated tools like consistency checkers, the architecture should at least offer the possibility to use the toolset as the first stage of such a tool. As a result, the new tool can benefit from the adaptability of the toolset to different environments, and reuses existing functionality.

The reuse of functionality can be achieved by the design of small, configurable modules with clearly defined responsibilities. Information can be passed between modules using a *blackboard strategy* [3] with each module being implemented as a separate traversal of the abstract syntax tree of the OCL expression. Additional dependencies, such as type information offered by the type checker module, are restricted to Java interfaces to allow different implementations (*strategy pattern* [7]).

3.3 Modules and the Abstract Syntax Tree

According to the above listed requirements and design decisions as well as common compiler implementation principles, the toolset architecture consists of the modules presented below. Figure 1 gives an overview of the basic modules of the OCL toolset. Following classical techniques from compiler design, the essential internal interface of the OCL toolset is the abstract syntax tree.

Abstract Syntax Tree Abstract syntax tree classes are created out of a grammar description using the the parser generator SableCC [18]. The parser generator creates abstract classes for productions and a concrete subclass for each of the alternatives of the production. These classes, generated for a straightforward translation of the OCL grammar in [15] into the specification format of SableCC,

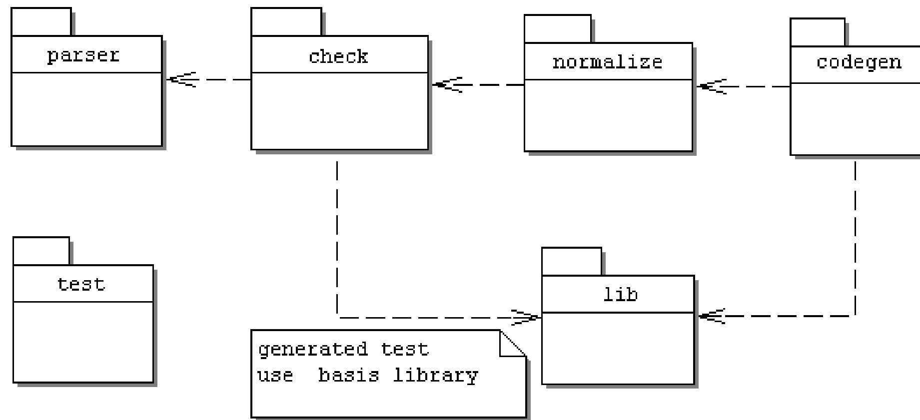

Fig. 1. Modules of the OCL toolset

are used as the primary information exchange data structure between modules. Figure 2 shows a screenshot displaying the abstract syntax tree of the prototype implementation. The current user interface is more targeted towards test and demonstration of the core functionality of the toolset. It is easy to replace this user interface by other interfaces, which are for instance more integrated with other UML tools.

The decision to use an abstract syntax tree as internal storage form of the OCL toolset was taken for very pragmatic reasons. The SableCC system provides quite a number of helpful mechanisms to deal with abstract syntax trees effectively, which could be reused. Moreover, in the current situation, the evolving OCL metamodel [16] is not yet an alternative since it is not fully stabilized (and will probably not become stable before UML 2.0). The current decision allows a very loose coupling between the OCL toolset and its environment and is based on current standards. As soon as the metamodel is stable, it will be easily possible to introduce an additional layer of abstraction on top of the syntax tree, which will be based on an OCL metamodel. In fact, a metamodel view was already used in some of the typechecking algorithms (see below).

Parser The parser transforms the input OCL expression into an abstract syntax tree. Of course it is straightforward to use a SableCC-generated parser for this task, but it can be fulfilled by an arbitrary parser that creates the appropriate instances of the SableCC-generated abstract syntax tree classes.

Semantical Analysis The abstract syntax tree classes can be seen as a representation of a static UML metamodel[1]. By formulating invariants that constrain this model, consistency rules (well-formedness rules) on the abstract syntax tree

[1] This metamodel is not the one proposed in [16], but the UML model corresponding to the classes generated by SableCC.

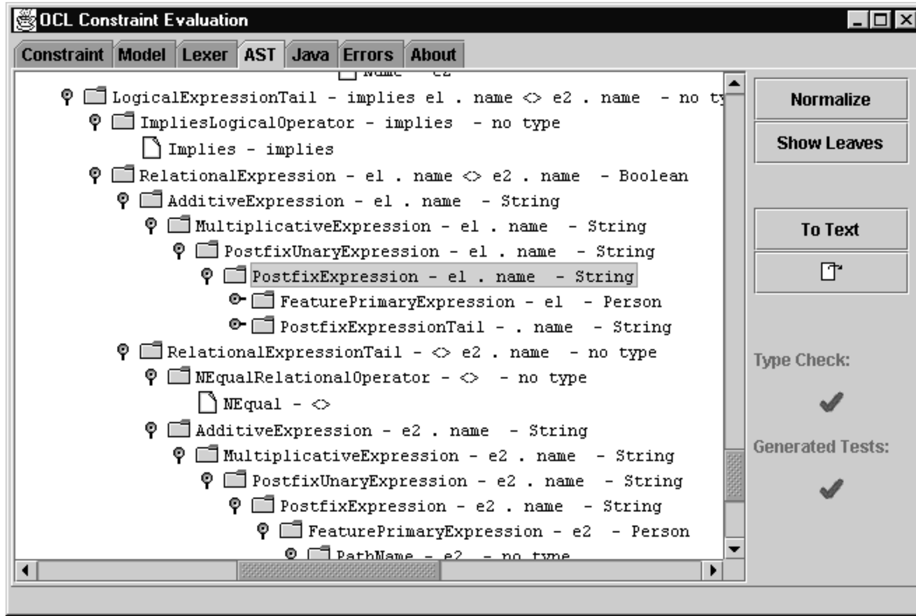

Fig. 2. User interface of the OCL toolset prototype

can be defined. The Java code generator of the OCL toolset can be used to automatically transform these constraints into Java code which can then be used as a part of semantical analysis. In fact, part of the typechecking in the prototype was "bootstrapped" already this way.

Type checking is implemented as a module that, in addition to checking semantical correctness with reference to the OCL type system, offers type information about syntax tree nodes and variables towards other modules. To minimize inter-module dependencies, this information is made available through a clean Java interface. It is important to note that OCL type checking is not possible without information about the UML model the OCL constraint is part of. Such information is not available within the OCL toolset, but has to be extracted from the toolset's environment. An external interface for this purpose is described in subsection 3.4.

Normalisation In order to be able to support a variety of tools, it is desirable to avoid that every tool has to implement the execution of any OCL expression completely. This can be achieved by defining a *normal form* of OCL terms, such that all terms can be mapped into a simpler subset of the OCL language. Such a normal form could for example disallow multiple iterators for the collection property `forAll`, since they can be replaced by nested iteration.

As it was mentioned in requirement 4 above, different normal forms are preferable for different purposes: For consistency checking it might be helpful to normalise collection properties like `forAll` and `exists` to `iterate`, but a

Java code generator might produce less efficient code after this modification. Customized normalisation is made possible in our tool architecture by defining and implementing small normalisation steps, like "remove multiple iterators" or "expand shorthand for `collect`". The implementors of OCL tools based on our platform are free to combine these steps as desired, or even to add further normalisation steps.

While the normalisation module does not define an additional internal interface, other modules depend on properties of the abstract syntax tree achieved by normalisation. These dependencies can be made explicit by managing a list of asserted invariants for the syntax tree. Normalisation steps can add invariants to this list, and modules dependent on certain invariants can assure that these have been asserted before.

Code Generation Code generator modules transform the normalised syntax tree into a target language. How this is done is to a large extent dependent on the target language. For the implementation of a Java code generator, a combination of a class library and a comparatively simple syntax-directed translation has proven to be sufficient. The class library offers Java-representations for the predefined OCL types, and the code generator can make straightforward translations of OCL property accesses into Java method calls for most cases.

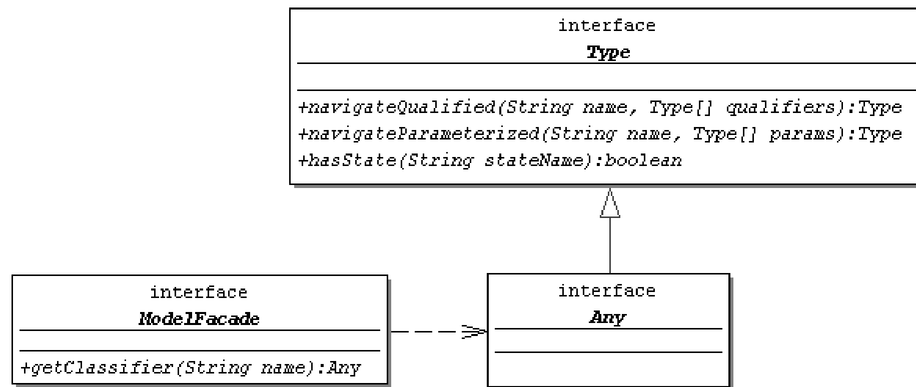

Fig. 3. Model information source interfaces

3.4 Interfaces of Tool Modules to the Environment

An OCL toolset needs at least two interfaces for communication with its environment, as stated above in requirements 1 and 2.

The first interface allows the environment to pass OCL expressions into the toolset. Constraints are here represented as simple String objects.

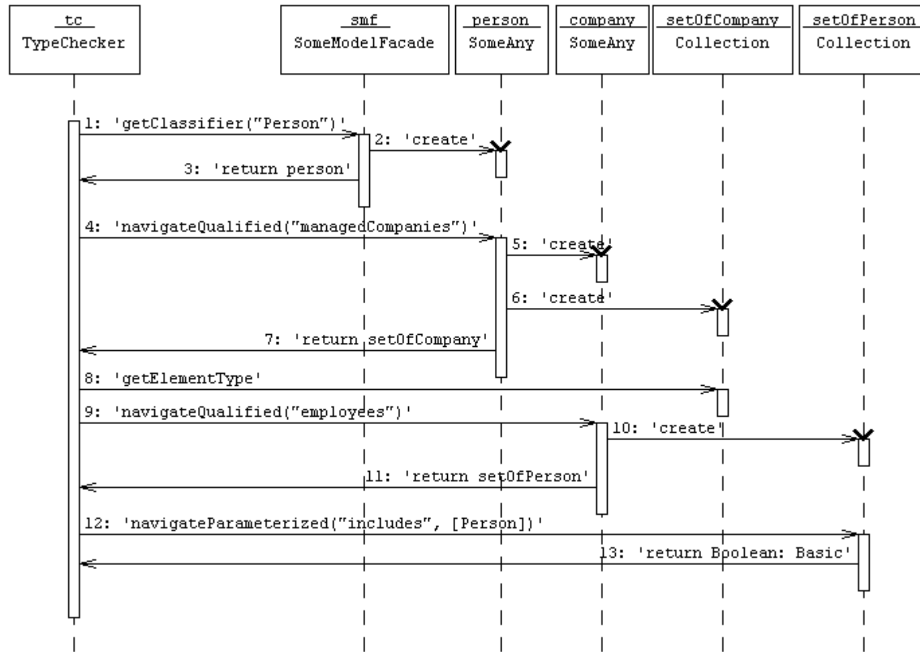

Fig. 4. Communication between type checker and model information source

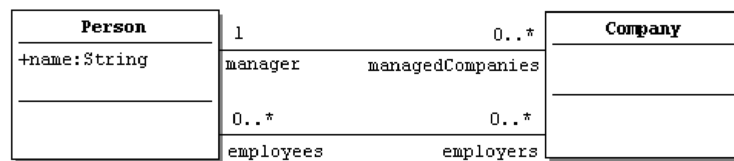

Fig. 5. Model for example constraints

The second external interface is used by the type checker of the toolset to access model information. Since OCL expressions are dependent only on a small part of the UML meta model (classifiers, behavioural and static features, associations, and states), the necessary queries can be restricted to the small and elegant interfaces `ModelFacade` and `Any` shown in figure 3. These have to be implemented for a toolset environment. Figure 4 shows how the type checker and implementations of these interfaces cooperate to examine the following constraint:

```
context Person inv:
managedCompanies->forAll(employees->includes(self))
```

All example constraints refer to the model in figure 5.

Several experimental implementations of the `ModelFacade` interface have been realised already. For instance, there is an implementation which reads the model information out of a file in the XML-based UML exchange format XMI (compatible with the XMI export facility of Rational Rose), and an implementation which takes the model information directly from the repository of the OpenSource CASE tool Argo/UML. Another implementation extracts model information from Java classes that are enriched with Javadoc comments which show the element type of collections.

Additional external interfaces may be added by other modules. For example, for a code generator it makes sense to offer an interface where produced code can be queried. Such an interface should not only make the generated code itself available, but also offer information such as the constrained class and operation or, for procedural target languages, the variable that holds the result of the evaluation.

3.5 Java Assertion Generation

As a first complete configuration of the modules of the described toolset, a compiler has been realised which translates OCL constraints into Java assertion code [14]. According to the classification given in section 2, this OCL tool has the functionalities of dynamic invariant validation and dynamic pre/postcondition validation. It can also be used for test automatisation. The approach to generate code which is executed at runtime as part of an actual implementation of the UML model is less straightforward that e.g. an OCL interpreter and model animator [5]. However, we believe that for the application of OCL in larger projects, the compilation into assertions is much more important, whereas the interpretative approach is more suited to education in OCL.

The prototype compiler uses the XMI-based implementation of the `ModelFacade` interface. Therefore it can be used to generate Java assertion code out of two files: a text file containing OCL constraints and a UML/XMI file (containing the class diagram), which is created by export from standard UML CASE tools. Internally to the compiler, the normalisation module is used to transform the input constraint into a sublanguage of OCL that avoids the use of multiple iterators, iterating properties without declarator, multiple use of the same variable

name, and use of the default navigation context[2]. Figure 6 shows a normalisation example.

```
input expression
context Company inv:
employees->forAll(e1, e2 | e1 <> e2 implies e1.name <> e2.name)
```

```
normalised expression
context Company inv tudOclInv0:
let tudOclLet0 : Set(Person) = self.employees in
tudOclLet0 -> forAll (
  e1 : Person | tudOclLet0 -> forAll (
    e2 : Person | e1 <> e2 implies e1.name <> e2.name
  )
)
```

Fig. 6. Example for normalisation

The Java code generator module then follows syntax-driven rules to produce Java code, a sample of which is shown in figure 7. The classes used in this code, like `OclAnyImpl` or `OclSet`, are defined in a class library, which defines among other things implementations for the standard collection data types of OCL. As usual with compilers, the generated code is not meant for human readers but for execution by machine. It uses a number of auxiliary variables to break down the code in relatively small pieces (a standard compiler construction technique). After executing this code, the result variable (`tudOclNode8`) contains a value which indicates whether the examined object fulfills the constraint. In order to access the actual snapshot of the model at runtime, again a simple and elegant Java interface has been defined (using the *Factory Method* design pattern) that makes it easy to adapt the assertion code to any chosen representation of UML constructs in Java. This feature is particularly important for the representation of associations, where many significantly different design choices exist.

In order to make practical use of our prototype OCL compiler, a separate tool is required which takes this code and inserts it as the body of a new method `assertOcl()` into a Java source code file. Figure 8 shows the effect of this tool for the file `Company.java` from the running example. Using such generated tests, component testing can be greatly simplified. For instance, in the *JUnit* testing framework [11], assertions are made that usually compare the result of a method call with the desired result. Using the automatic generation of assertions, it is sufficient to just make calls to the methods that are being tested. The validation of the result is automatised based on an OCL post-condition for the method which was formulated during modelling.

Figure 9 gives an overview of the interaction of client and compiler modules for Java assertion generation.

[2] Not using the default navigation context means that every navigation expression has to begin with a bound name, like `self` or an iterator name, or a literal.

```
final OclAnyImpl tudOclNode0=Ocl.toOclAnyImpl( Ocl.getFor(this) );
final OclSet tudOclNode1=
  Ocl.toOclSet(tudOclNode0.getFeature("employees"));
final OclIterator tudOclIter0=tudOclNode1.getIterator();
final OclBooleanEvaluatable tudOclEval0=new OclBooleanEvaluatable() {
  public OclBoolean evaluate() {
    final OclIterator tudOclIter1=tudOclNode1.getIterator();
    final OclBooleanEvaluatable tudOclEval1=new OclBooleanEvaluatable(){
      public OclBoolean evaluate() {
        final OclBoolean tudOclNode2=
          Ocl.toOclAnyImpl(tudOclIter0.getValue()).
          isNotEqualTo(Ocl.toOclAnyImpl(tudOclIter1.getValue()));
        final OclString tudOclNode3=
          Ocl.toOclString(Ocl.toOclAnyImpl(tudOclIter0.getValue()).
          getFeature("name"));
        final OclString tudOclNode4=
          Ocl.toOclString(Ocl.toOclAnyImpl(tudOclIter1.getValue()).
          getFeature("name"));
        final OclBoolean tudOclNode5=
          tudOclNode3.isNotEqualTo(tudOclNode4);
        final OclBoolean tudOclNode6=tudOclNode2.implies(tudOclNode5);
        return tudOclNode6;
      }
    };
    final OclBoolean tudOclNode7=
      tudOclNode1.forAll(tudOclIter1, tudOclEval1);
    return tudOclNode7;
  }
};
final OclBoolean tudOclNode8=tudOclNode1.forAll(tudOclIter0,tudOclEval0);
```

Fig. 7. Generated Java code for the example of figure 6

```
import tudresden.ocl.lib.*;
import java.util.*;

public class Company {

  public int numberOfEmployees;

  protected Person manager;
  protected Vector employees;

  public void assertOcl() {
      tudOclAssert0();
  }

  private void tudOclAssert0() {
    // generated Java code is inserted here

    if ( ! tudOclNode8.isTrue() ) {
      throw new RuntimeException("constraint violated");
    }
  }
}
```

Fig. 8. Java class with assert method (generated code is omitted)

3.6 SQL Integrity Constraint Generation

Another configuration of the modules of the presented OCL toolset architecture is currently under development, which aims towards an integration with database schema generation tools. The intention is to automatically generate SQL integrity constraints [12] as part of a relational database schema. The integrity constraints are derived from OCL expressions that are specified in form of business rules on UML models. Basic mappings from OCL invariants to SQL constraints (`create assertion` et al.) are given in [6]. For this purpose, the parser and type checker can be reused without change. However, SQL as declarative language requires in contrast to Java other properties of the abstract syntax tree. For example, it makes no sense to normalise the above given example (see figure 6) in the same way as presented above. In this case, it would be better to normalise the OCL expression into an abstract syntax tree representing the equivalent OCL constraint:

```
context Company inv: employees->isUnique(name)
```

This expression can be easily transformed into a SQL unique key constraint. Therefore, specific normalisation steps have to be developed and added to the normalisation module. A further development step is then the implementation of the code generation interface for the generation of SQL statements. Because of the large number of relational database system vendors that offer different

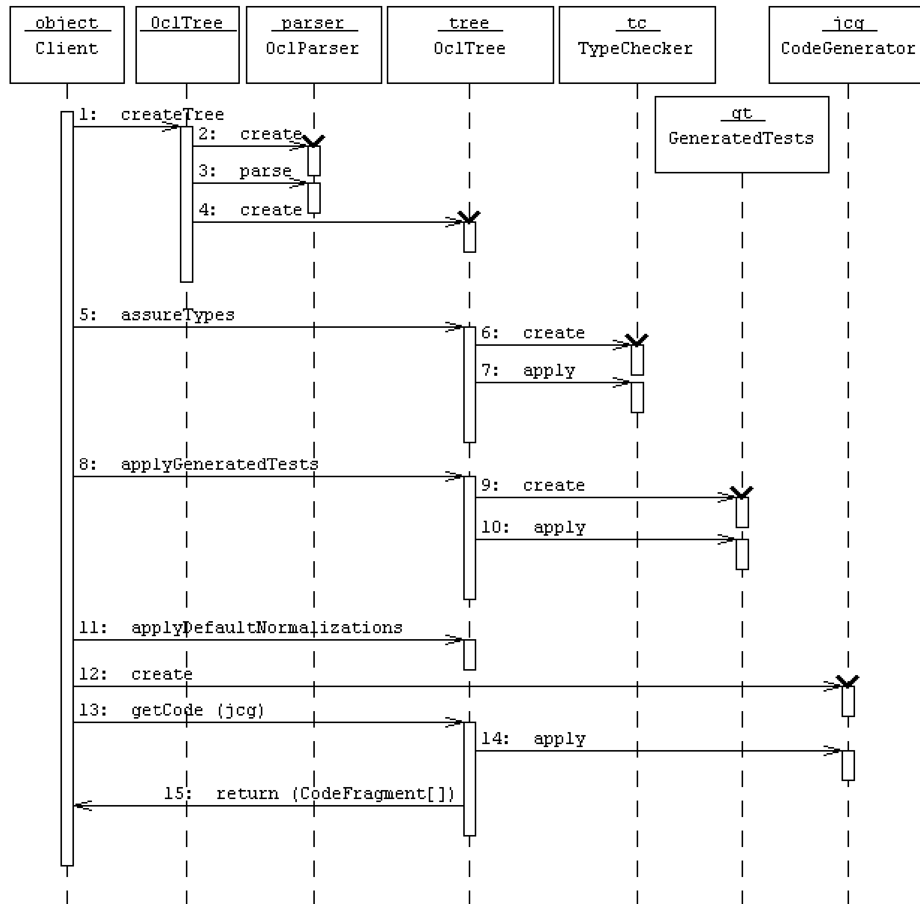

Fig. 9. Inter-module communication for Java code generation

implementations of the SQL standard, a flexible approach for code generation is needed. One way to achieve this is the separation of the SQL code generator from the mapping rules by their description in XML. The structure of such a document is predefined by a document type definition (DTD) and can be seen as a catalog that contains SQL code templates related to the grammar rules of the OCL specification. Using this approach, the generated SQL code is to a large extent independent of the compiled code and can easily be adapted to different SQL dialects using XML-Editors. The current OCL prototype toolset supports both SQL-92 [12] and Oracle/SQL code generation. Although this development is not yet completed, the design of it already gives some proof for the adequateness of the chosen modular structure of our OCL toolset.

4 Summary and Outlook

The purpose of this paper was not just to describe a particular implementation of an OCL tool. Instead, a rather general discussion of the potential tool support for OCL has been given, and the described design contains a number of ideas which may be transferable also to other tool environments. Moreover, the described tool platform may be interesting for a wide audience, since the full source code is freely available under GNU Library General Public License [14].

The current status of realisation comprises a complete and stable implementation of all the modules which were mentioned in figure 1. The compiler from OCL constraints to Java assertions is available and thoroughly tested. The compiler from OCL to SQL integrity constraints is currently under development. There are already several other (academic) projects which have decided to take the tool platform described here as the basis for their development of OCL tools.

The design of the toolset is oriented towards an easy integration into all kinds of other (Java-based) environments. It was an encouraging experience regarding the toolset design that a first prototype of the integration into the OpenSource UML tool Argo/UML was produced within just a few hours of development time (as could be observed "live" by the participants of an OCL workshop taking place in Canterbury/UK in March 2000). The core parts of our OCL toolset will be fully integrated into future releases of Argo/UML. Moreover, experiments for integration with other Java-based CASE tools, e.g. Together, are going on in cooperation with other research institutions.

Further work from our side will concentrate on additional modules for the toolset which enable practical experimentation with fully automatic tool support for OCL. Among the future plans is to develop an OCL interpreter based on our toolset and to connect it with a CASE tool in order to automatically check meta-level OCL constraints on UML models. Such a tool may provide significant help in stabilizing the formal OCL parts of the current OMG standard for UML. Another goal for the near future is to provide adequate tool support for automatic testing based on OCL, and to carry out pilot studies for the use of such tools in small but realistic development projects.

From the viewpoint of tool developers, we can summarize that effective support for the OCL part of UML is possible, and that the language is in principle well designed to achieve a high level of automatisation within such tools.

Acknowledgment: The authors would like to thank Ralf Wiebicke and Sten Loecher for their contributions to the prototype implementation.

References

1. Argo/UML Page, http://www.ArgoUML.com
2. Baar, Th.: Experiences with the UML/OCL-Approach to Precise Software Modeling: A Report from Practice, Unpublished report 2000, see http://i12www.ira.uka.de/~projekt/publicat.htm
3. Buschmann, F. et al.: Pattern-Oriented Software Architecture - A System of Patterns. John Wiley and Sons Ltd, Chichester, UK, 1996
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999
5. BoldSoft, Object Constraint Language Support Information, http://www.boldsoft.com/products/bold/ocl.htm
6. Demuth, B., Hussmann, H.: Using OCL Constraints for Relational Database Design. in: UML'99 The Unified Modeling Language, Second Int. Conference Fort Collins, CO, USA, October 1999, Springer, 1999
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, 1995
8. Haehnle, R., Menzel, W., Schmitt, P.H. KeY - Integrated Deductive Software Design, see http://i12www.ira.uka.de/key
9. Jackson, D. et al. (MIT Software Design Group): Alcoa. http://sdg.lcs.mit.edu/alcoa/
10. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML - The Catalysis Approach, Addison-Wesley 1999.
11. JUnit, http://www.xprogramming.com/software.htm
12. Melton, J., Simon, A.: Understanding the New SQL: A Complete Guide. Morgan Kaufmann, 1993
13. Meyer, N.: Applying "Design by Contract", IEEE Computer, 25(10), Oktober 1992, pp. 40-51.
14. OCL Page, Dresden University of Technology, http://dresden-ocl.sourceforge.net/
15. OMG UML v. 1.3 specification, http://www.omg.org/cgi-bin/doc?ad/99-06-08
16. Richters, M., Gogolla, M.: A Metamodel for the UML Object Constraint Language OCL. in: UML'99 The Unified Modeling Language, Second Int. Conference Fort Collins, CO, USA, October 1999, Springer, 1999
17. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. in: UML2000 The Unified Modeling Language, contained in this proceedings, Springer, 2000
18. SableCC Homepage, http://www.sable.mcgill.ca/sablecc/
19. Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999