OCL EXCEPTION HANDLING

A Thesis

by

PRAMOD GURUNATH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2004

Major Subject: Computer Science

OCL EXCEPTION HANDLING

A Thesis

by

PRAMOD GURUNATH

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Richard Volz (Chair of Committee)	Karen L. Butler-Purry (Member)
William M. Lively (Member)	Valerie E. Taylor (Head of Department)

August 2004

Major Subject: Computer Science

ABSTRACT

OCL Exception Handling. (August 2004)

Pramod Gurunath, B.E., Bangalore University, India

Chair of Advisory Committee: Dr. Richard Volz

Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) specification and can be used to enforce constraints on the attributes or methods of a class. It would greatly help the software developers if such non-executable OCL constraints specified in a UML model could be enforced on the executable code generated from the model. This thesis discusses the concepts, ideas and the approach in transforming a model developed in the Rational Rose software with OCL constraints into Java code shells, complete with fragments of code to detect the run-time violations of the constraints. The implementation and testing of a prototype tool that incorporates these ideas is also discussed.

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Volz, for his unending support, guidance and encouragement in completing this thesis. I have learned a lot from him through the several interesting discussions we have had during this research.

I would also like to express my sincere thanks to my committee members, Dr. Lively and Dr. Butler for their valuable comments on my research.

I would also like to thank James C. Thompson of the United Space Alliance for his valuable input and suggestions.

Lastly, I would like to thank my family for their constant encouragement and belief in me throughout my graduate studies.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

Page

INTRODUCTION

The Unified Modeling Language (UML) [18] is a language for specifying, visualizing, constructing and documenting the artifacts of the software systems as well as for business modeling and other non-software systems. The Object Constraint Language (OCL) [17] is part of this UML specification and is a formal language used to express constraints. The constraints typically specify invariant conditions that must hold for the system being modeled. OCL is a pure expression language; when an OCL expression is evaluated, it just returns a value and the state of the model is unaltered. OCL is not a programming language and the OCL statements are not directly executable. As a result, the detection and handling of the run-time violations of these constraints is left to the software developer. Thus, an additional tool that accomplishes automated insertion of code that detects the constraint violation can be extremely helpful, especially when designing a large software system.

Our goal is to have a system that allows users to develop models in Rose, enter the OCL constraints in the model and get code shells that have the ability to detect the run-time violation of the constraints. This thesis will propose a mechanism for OCL exception handling and address various issues involved and the means by which the routines to detect constraint violation can be inserted into the code shells generated by Rational Rose [21]. Some of issues that we address are converting the non-executable OCL language statements to executable code; inserting this OCL-code at appropriate points in the code shells generated by Rational Rose code generator, and the code

---

This thesis follows the style of the *IEEE Transactions on Computers* journal.

patterns that should be used for the various constraints. The concepts and the system assume that the target language is Java. The insertion of the code, which corresponds to the OCL constraint statements, will be done as the post-processing step after the code generation from the Rose tool. The OCL constraint statements will be limited to the invariant, pre- and post-conditions. The code patterns that will be used for each of the constraint statements will be designed so that the constraint violation can be caught by exception handling techniques in Java.

The concepts and the system designed will be tested against an example model, which will be self-contained, and complete with respect to most of the popular UML constructs, such as associations, association class, inheritance and nested classes. Further, our focus will be to develop and demonstrate the fundamental principles involved in the context of most widely used OCL constraints, rather than doing a totally comprehensive engineering project. As a result, we concentrate only on the invariant and pre- and post condition OCL constraints.

BACKGROUND

The Unified Modeling Language [18] [6] is the most popular modeling language used to design software applications. UML helps users to express the requirements, analysis and design of software applications. It defines two important concepts: the notation and the meta-model. The notation is the syntax of the language and is the graphical layout in the model. A class diagram notation, for example, defines how items and concepts such as class, association and multiplicity are represented. The meta-model represents the relationship among the various notations. Class diagrams thus describe the types of objects in the system and the various kinds of relationships that exist among them. The interaction diagram helps a user to know the control and/or the data flow among the various entities of the system. There are several types of other diagrams [18]. Since constraints typically apply to classes and methods, we consider only the class diagrams that can be developed using the Rational Rose software tool.

Overview of OCL

The Rational Rose software [21] is a commercially available tool that fully supports UML. Using Rose, we can develop class diagrams using various object-oriented features. A Rose class diagram, however, is not complete with respect to all the features of UML. For example, we cannot enter constraints on the classes and their methods. Since constraints are part of the complete description of an object, without the constraints, the description would be incomplete. The constraints could be on the attributes of the object or on its methods. In order to write unambiguous constraints, the

formal languages have been developed. A formal language that can specify constraints is the Object Constraint Language [32], which is part of the UML specification. [17] has a good coverage of OCL and its constructs. According to the OCL specification, OCL is:

- A pure expression language
- Not a programming language
- Typed language

OCL can be used for a number of purposes [17]:

- To specify invariants on classes and types in the class model.
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on operations and methods
- As a navigation and description language
- To specify constraints on the operations.

As mentioned in the previous section, we restrict ourselves to the invariant and the pre- and post condition constraints of the OCL specification.

Invariants are constraints that can be enforced to specify that a particular attribute, or function of a collection of attributes, obeys the constraint at all times. For example, we can specify that in case of a class `Company`, the attribute `numberOfEmployees` will always exceed 50 as:

```
context Company inv:
self.numberOfEmployees > 50
```

Every OCL expression is written in the context of an instance of a type. The reserved keyword *self* is used to refer to the contextual instance. In the above invariant, *self* is referring to an instance of the class `Company`. The `context` keyword introduces the context of the expression, i.e., the class or the method on which the constraint applies.

Pre- and post conditions can be specified on methods in a similar manner. For the class `Person`, whose annual `income` has to be found, we can specify, using the keyword `result`:

```
context Person::getIncome() : Integer
post: result = monthlySalary * 12
```

OCL has a number of basic types defined that can be used in any OCL expression. These basic types are Integer (for integer types), Real (for float or real types), Boolean (for Boolean – true or false) and String (for the string literal). In addition, OCL defines some predefined operations on these basic types. For example, for the type Integer, a modulus, minimum and maximum operators are defined in addition to the addition, subtraction, multiplication and division operations. For the type String, there are concatenation and substring functionalities among others. The complete list of operations can be found at [17]. OCL is a typed language. Enumeration types are also supported in OCL. A sequence of elements can be specified using the *Collections* type. *Collections* supports ordered and unordered sets. They are primarily used to access array-type attributes in the model. A number of pre-defined operations are also defined for the *Collections*. An example is given later (See page 47).

We can define sub expressions in OCL using the *let* expression. For example,

```
let income : Integer = self.job.monthlySalary * 12 in
```

defines an integer *income* that represents the annual salary. Defining sub expressions is useful when dealing with a large OCL constraint. A number of other features are explained in detail in the OCL specification [17].

OCL is thus powerful enough to enable us define constraints using the OCL features and predefined types and their operations. Since OCL has the provision to access the various constructs of UML, such as classes, attributes and methods, it is fully compliant with UML and has methods for easy navigation to association classes and subtypes.

RELATED WORK

Extensive work has been carried out in the field of UML and OCL. There are a number of tools that support UML. However, there are not many tools that support both UML and OCL. Since OCL cannot exist without a model, it has to occur within UML. We first explore the different tools that support UML. We then look at the previous work performed to support OCL.

UML tools

In object oriented software development, the design phase is one of the most important phases. It is during this phase that the architecture of the software is brought to life, by clearly depicting a number of classes and their interactions. UML is extensively used for this process and a number of tools are commercially available. Tools that we evaluated were:

- Rational Rose software [21]: This is a popular and most widely used UML-based tool. It supports a number of features in UML, including use cases, class diagrams, interaction diagrams and the sequence diagrams. The model file can be saved either in the ASCII format (called the Rose Petal format, *.ptl) or as an application-specific format (called the Model file, *.mdl). It supports object-oriented languages like C++ and Java. Rose has the facility to generate code shells (i.e., files that contain only the entire specification of the classes except the body of the methods) that represent the classes and their associations. However, Rose does not support OCL. It has no facility to input OCL constraints and

instrument the code shells with executable code representing the OCL constraints.

- CrazyBeans [3]: This is an open-source project by a small community of developers. It is essentially a parser that can process a Petal file (*.ptl) generated from Rose. Also, this parser can be integrated with another tool called the NSUML [16]. With the help of the NSUML tool, Java code shells can be generated. The advantage here is that since it is open source, we have the source code and hence would be possible to modify the tool. However, when we performed some testing, we found that the code generation was incorrect for several of the UML features like the association class and associations. Further, the code generated could not be compiled due to the presence of syntax errors and hence was found to be buggy. Moreover, the Petal file format is not officially published by Rational Rose.
- Eclipse [5]: Eclipse is an IBM-driven open source development platform. Initially, it was conceived as a Java Integrated Development Environment (IDE). It has now expanded to include the Eclipse Modeling Framework (EMF), which supports UML. EMF is the basic framework for UML and supports Java code generation. However, it does not support all the features that Rose offers. Plug-ins are available that provide support for OCL within the EMF, but the code shells generated from EMF are not instrumented to detect the constraint violations.

- ArgoUML [25]: ArgoUML supports almost all features mentioned in the UML specification. It is open source and is implemented in Java. It tries to emulate the commercial Rose software in many ways. However, there are several features missing [25]. An important addition in ArgoUML is that the Dresden parser, part of the Dresden compiler is integrated. This enables the user to enter OCL constraints into a designated field and check for its validity. It, however, has no provision to either generate the Java code corresponding to the OCL constraints or its insertion into the Java code shells generated by ArgoUML's code generator The OCL equivalent Java code can be generated using a separate tool called the Dresden Injector (see *OCL tools* section).

A number of tools thus support UML. However, the Rational Rose tool, in addition to being fully compliant with UML, has tools for code generation (Java/C++), model import/export and has good technical support. Further, in the enterprise sector, Rose has been the most widely accepted UML tool. Hence, we decided to use Rose as our UML tool. In particular, we use the Rational Rose Enterprise edition, release version 2002.05.20.

OCL tools

There exist a few OCL-related tools that have been developed quite recently. These tools address how we can parse the OCL constraints, transform them to code and

integrate it with the code shells generated by the UML tools. Some of the important ones are:

- The IBM OCL Parser [9]: This tool parses the OCL grammar and prints out the different tokens. It is a non-commercial and un-maintained product, which is no longer under development. It does not have a code generator.
- The Dresden OCL Compiler [22]: This tool has a parser and a code generator parses OCL and produces Java-equivalent code. Some demo-applications are available that take in OCL constraints and output the equivalent Java code. This tool is not being actively developed [8], but is better than the IBM OCL parser. However, the tool does not accept the Rose file format as the input; instead, it has a hard-coded model on which this has been tested. It does claim support for the XMI format [19].
- The Dresden OCL Injector [22]: This tool uses the Dresden OCL Parser. It reads in the Java code generated from the ArgoUML code generator, extracts the OCL constraints and modifies the code so that the constraints are enforced. In order to enforce the OCL constrains on the code, the injector tool first obtains the Java code equivalent for the OCL using the Dresden OCL compiler and inserts this code into the code generated by ArgoUML. The manner in which this insertion (or injection) takes place is by using “wrapping the methods” [33] code pattern. In the “wrapping the methods” code pattern, the existing methods’ signature and calls are changed so that it these new methods incorporate the constraint code and the call to the original function. However, using the injected code, constraint

violations cannot be effectively detected since it has no provision for throwing exceptions, which would have greatly helped to handle the constraint violations effectively. All it does is print out a message if a constraint is violated. Further, the code injected by the Dresden injector is obscure and hence a programmer desiring to make changes to the injected code would thus find it difficult.

- A tool by Verheecke et al. [29]: In [29], Verheeke et al. exploit the object-oriented paradigm by representing constraints (in OCL) as explicit classes in the implementation. The various points in the code where the constraints must be checked are automatically detected by the tool. These *insertion points* boil down to a set of methods that need to be checked for constraint violation. The user can then specify the *checkpoint* (where in the methods, the constraints need to be checked) and the *action* (action to take when constraints are violated). However, there is no explicit mention of their support for the Rational Rose tool.
- OCL Environment [24]: This is an integrated development environment for OCL. It integrates with a model specified in the XMI [19] format. The constraints are entered into separate constraint files. It claims support for generating the Java code for the model along with the code that can check for constraint violations. However, we could not find documentation on the code patterns used to insert the Java code corresponding to OCL statements into the Java code shells. It also does not state if it supports models in Rational Rose with OCL constraints embedded in them.

- Cybernetic OCL Compiler [23]: The Cybernetic intelligence has developed a tool that can read a Rose model with OCL constraints embedded in them. It can parse and check for OCL's validity and the model's well-formedness. However, it currently does not support code generation for the constraints.

More analysis of OCL tools is presented in [26]. We observe from [26] that there are not many tools that can check model's consistency and generate code for the OCL constraint statements.

Other related tools

Since Rose is an established and reliable CASE [20] tool, we consider class diagrams from Rose alone, as mentioned before. However, since it is propriety software, we cannot interpret the Model file it generates. The XML Metadata Interchange (XMI) [19] is a specification that allows the easy interchange of metadata between modeling tools. The tool that transforms a Rose model into XMI is the Unisys XMI plug-in [27]. This plug-in outputs an XML file that can then be read by a parser to process the model file.

In order to instrument the Java code shells generated from Rational Rose, we should be able to parse the Java files and know their structure. There are several Java parsers available. Following are the most popular parsers:

- JavaCC [30]: The Java Compiler Compiler is an open source generic parser implementation on the Java platform. The grammar for the parser to be generated

is input in a special formatted file called the *jj* file. It can parse any language whose BNF grammar is specified satisfies the LALR [1] conditions. When we use the grammar for the Java language [10], it generates a Java parser that can parse an input Java source file. Since it is just a parser and not a code generator, it simply outputs the same Java source code in a formatted form. However, this is exactly what is needed as a basis for the work described here. We will refer to such a parser as the Java parser from now on. In the ensuring discussing, we will describe how this Java parser is modified to insert the appropriate OCL-Java code for detecting and reporting constraint violations.

- JTB [31]: The Java Tree Builder is a syntax tree builder that is used along with JavaCC. It takes the JavaCC grammar file and modifies it in such a way that the parser created follows the Visitor [7] pattern. Thus, each production is a node in the syntax tree. The syntax tree can be navigated easily with the help of the visitor pattern.

There exist a few other tools that operate only on the Java files (might be source files or the compiled class files) to perform run-time constraint checking. These tools do not consider the model file that might have been used to create these Java files.

- Design by Contract tools (DBC) [15]: In Design by Contract, typically, the designer develops the class structure for a system using some set of UML tools, but it is the developer who writes the code that is expected to also insert the constraint statements (which are in a special syntax, not OCL). The constraints

are placed in a comments field immediately before a class or method declaration. The JMSAssert [14] tool of DBC processes the Java files that contain these constraints. The constraint expressions are specified in a special syntax and not OCL. The JMSAssert creates a separate set of helper files and links to the Java Virtual Machine (JVM) to check for constraint violations. This implies that there is no source code alteration other than the comments containing the constraints, and the tool is dependent on the JVM. This tool is currently available on the Windows platform.

Another DBC tool is the iContract [12]. It is similar to JMSAssert, except that the original source files are modified and is platform independent. The constraint expressions follow the Java language syntax, with few extensions based on OCL. However, the constraints do not support all the features or syntax of OCL. Jass [28] is yet another tool, similar to iContract, but supports some extra features, such as specifying loop invariants. The constraint expressions in Jass are specified in a special syntax and not OCL.

One of the disadvantages of all these tools is that the designer cannot specify the constraints during the design phase and that the constraints cannot be specified in pure OCL syntax.

- Constraint checking using Java Class files: The Handshake [4] and the jContractor [11] tools operate on the Java class files to enable detection of constraint violation. The constraints are specified in a separate file. The invariants, pre and post conditions converted to byte code and are then inserted

into the original byte code of the source. The advantage here is that, in the original byte code, the Java compiler would already have calculated the return points and the return value, so insertion of constraint checking code would be easy. However, even in this case, there is no relation to the model and the constraints cannot be specified in OCL.

FOCUS OF RESEARCH

As can be seen, most of the prior work mainly focuses on either the UML or OCL, but not both. The Dresden Injector tool attempts to integrate the two concepts, but it does not achieve this effectively. It does not support Rational Rose directly (instead, it gets the model information through reverse engineering the Java code shells) and the code pattern for detecting the violation of the constraint is obscure. Further, upon constraint violation, it just prints a message and does not throw any exceptions.

Using Rational Rose, the class diagrams can be developed with the desired level of detail and a target language (we assume Java for our research) can also be specified. There is no existing tool that can help a designer to insert the OCL constraints in the Rose model. Also, there is no tool that can insert the Java code generated from the OCL constraints (embedded in the Rose model) into the code shells using *simple, effective* and *maintainable* code patterns that can generate and throw exceptions upon detecting constraint violations. There have been no concrete documents that describe the approach, design or implementation of such a tool.

Our focus is to develop the concept and a system with the help of which, one can transform the model developed in Rational Rose software with OCL constraints into Java code shells, complete with fragments of code to detect the run-time violations of the constraints. The thesis discusses the various designs that we developed, our design decisions and analysis. The implementation of such a tool and some test cases are also presented.

APPROACH

In our research, we assume that the designer of the software system is using Rational Rose to model the system's various classes and their interactions. Our aim is that the resultant code that maps onto the classes should contain the code that detects constraint violation. We assume that the code is in the Java language. By making use of the previous work done on UML and OCL, we have come up with several concepts and designs for achieving our objective. Each of these concepts is explained below, with possible trade-offs. In all cases, we assume that the class diagrams would be developed in Rational Rose and that the Rose code generator can be used to created code shells that would implement the structure of a system being designed (minus, of course, the OCL).

Approach one

The first problem to be addressed is that Rational Rose does not have a provision to enter the OCL constraint statements. However, Rose does have a number of pre-defined fields that we can force into a mechanism for entering OCL statements. In particular, each class, attribute and method has an associated "Documentation" field that holds text descriptions. We can use this field to specify the OCL. Alternatively, any field could be used as well as long as it can be extracted from parsing the Petal file. Thus, we could also use the "Preconditions" and "Postconditions" field in Rose to specify the pre and post-conditions for methods respectively. We note, though, that Rose itself makes no use of these fields, and outputs nothing corresponding to them while generating Java code shells.

Once we have a model with OCL constraints in place, we can save this as the ASCII Petal (*.ptl) format. In addition, we can generate the Java code shells from Rose using its code generator. From the Petal format, we can extract the OCL constraint statements using the CrazyBeans parser. We can then pass these OCL statements to an OCL compiler that outputs the Java code equivalent of the OCL statements (which, from now on, is referred to as OCL-Java). With both the Java code shells and the OCL-Java in place, we should be able to determine the right points in the Java code shells where we have to insert the OCL-Java. However, in order to determine the insertion point in the Java code shell for a particular OCL-Java, we need to know where in the model the OCL constraint was specified. The only way to determine this is that we have to parse the petal file to find the OCL constraint, and once we get this, process the Java code shells to determine which and where in the Java code shell file the OCL-Java has to be inserted. This whole process is depicted in figure 1.

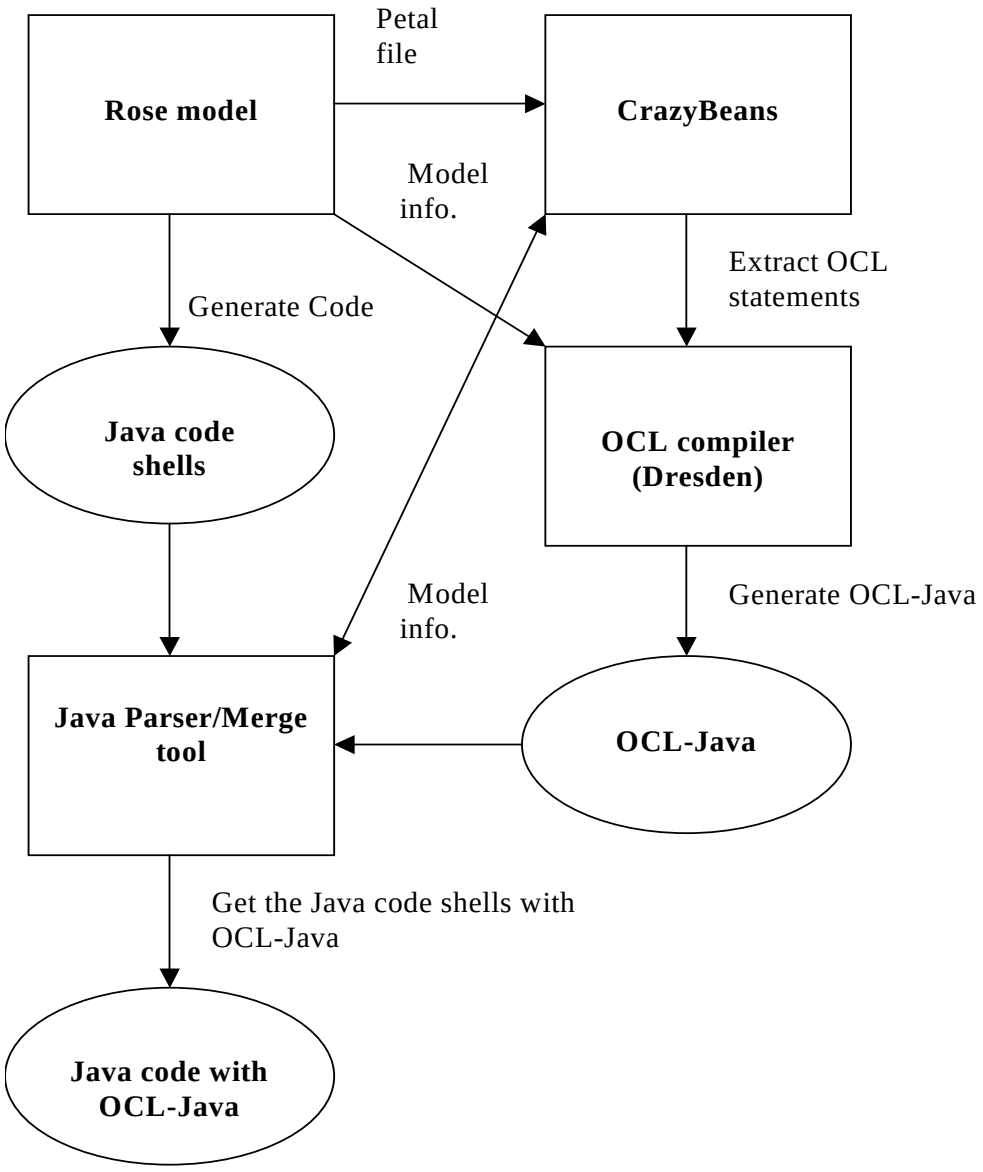


Figure 1 Approach one

There is one serious drawback in this method. In the Java parsing stage using the Java parser, we would have to know the relationship between the Java code shells and the OCL-Java. In other words, we should be able to determine where in the code shells we should insert the OCL-Java. For each OCL statement, we would have a

corresponding OCL-Java code fragment and hence we have to figure out this relationship for each of the fragment. Since the Java code shells themselves do not have this information, we have to parse the Petal file again and find the insertion point in Java code shell, as mentioned before. This would prove to be a costly process since this involves parsing the Petal file and searching in the code shells for the correct insertion point, for every OCL statement.

Approach two

In this approach, the class diagrams would be developed in Rose and OCL statements would be inserted in Rose, similar to approach one. However, we eliminate the drawback in approach one. Instead of trying to determine the correct points of OCL-Java insertion into the Rose-generated Java code shells, we can perform the OCL-Java insertion along with the Java code shell generation. This implies that we have the model information while performing the OCL-Java insertion, as desired. Since Rose's code generator is propriety, we can use the code generator of CrazyBeans. CrazyBeans is open source and we could modify it to emit Java code shells, complete with OCL-Java. This would require the CrazyBeans to call on the functionality of the OCL compiler. This process is depicted in figure 2.

The advantage in this method is that all the three functionalities – extracting OCL statements, generating Java code shells and inserting OCL-Java into the code shells – can be performed by the modified CrazyBeans. This is desirable since it would be an elegant design. However, one of the issues with this approach is that the core code

generator in CrazyBeans is incomplete. It is quite buggy and is expected to be difficult to fix, especially due to the lack of documentation. Also, we would prefer to have the Java code generated from Rose rather than CrazyBeans, since Rose is an established and tested product.

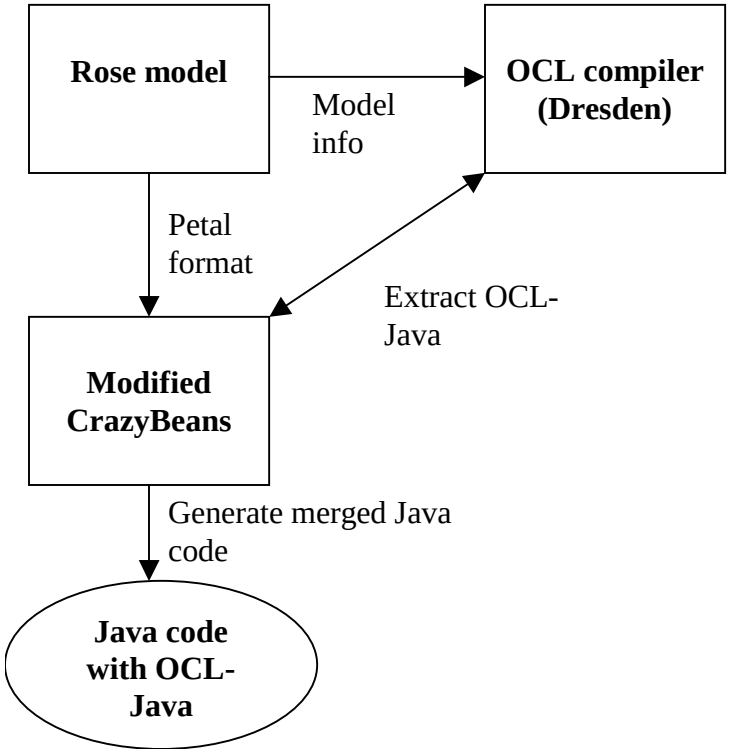


Figure 2 Approach two

Approach three

In order to eliminate the drawbacks of the above two approaches, we have to use the Rational Rose's code generator and make sure that the model information is present in some way within the Java code shells generated. The idea that we came up was that

we could insert the OCL constraint statements into the “Documentation” field of the class or the method. With this, the OCL constraint applying to the class or method would appear just above the respective classes and methods in the Rose-generated Java code shells. This means that we can get the model information just by processing the Java code shells, unlike the other two approaches mentioned before. Thus, for invariants, we would enter the constraints in the documentation field for the class and for pre and post-condition, we would enter the constraints in the documentation field for the method. Unlike the first approach, we must use the “Documentation” field and not any other field in the Rose tool to enter OCL constraints. Unlike the second approach, we use the Rose-generated Java code shells and hence can be assured that the code shells generated are stable.

Figure 3 depicts the overall architecture. The system designer develops the class diagram using Rose, inserting OCL constraint statements in the “Documentation” field, as mentioned before. We chose this field since Rose does not have an exclusive field to input OCL. The Java code shells are then generated using the Rose’s code generator. The code shells generated would contain the OCL constraints as comments before the class or the method declaration. The Java code shells would then be parsed by the Java parser to extract these OCL constraints. Once we get the OCL constraint, we would pass this to the Dresden OCL compiler. The OCL compiler takes the OCL constraint and the Rose model in the XMI format and gives back to the Java parser, the OCL-Java. Since the OCL constraints appear (as comments) just before the class or the method for which this OCL constraint would hold, the Java parser would know where exactly to insert the

OCL-Java into the Java code shells. Of course, the Java parser should be modified for the above functionality. The new merged Java code shell is written back. This is repeated for all the packages and files.

In the above method, we use standard stable tools already available for as much of the work as possible. Specifically, we use:

- The Unisys XMI plug-in to generate the model's XMI file. The Unisys plug-in is stable and is commercially supported.
- The Rational Rose's code generator that outputs the correct Java code for a given class diagram.
- The parser generated by JavaCC. This has been tested and found to be suitable for parsing Java files using the Java language grammar [2].
- The Dresden Compiler for OCL is also quite stable for simple class diagrams. As will be mentioned in the *Implementation* section, we have altered the Dresden compiler to be stable for most of class diagram features like association classes, nested classes and packages.

Thus, all the tools used in approach three are stable.

Among all the above designs, the approach three seems most efficient and practical. We thus chose this design for our implementation.

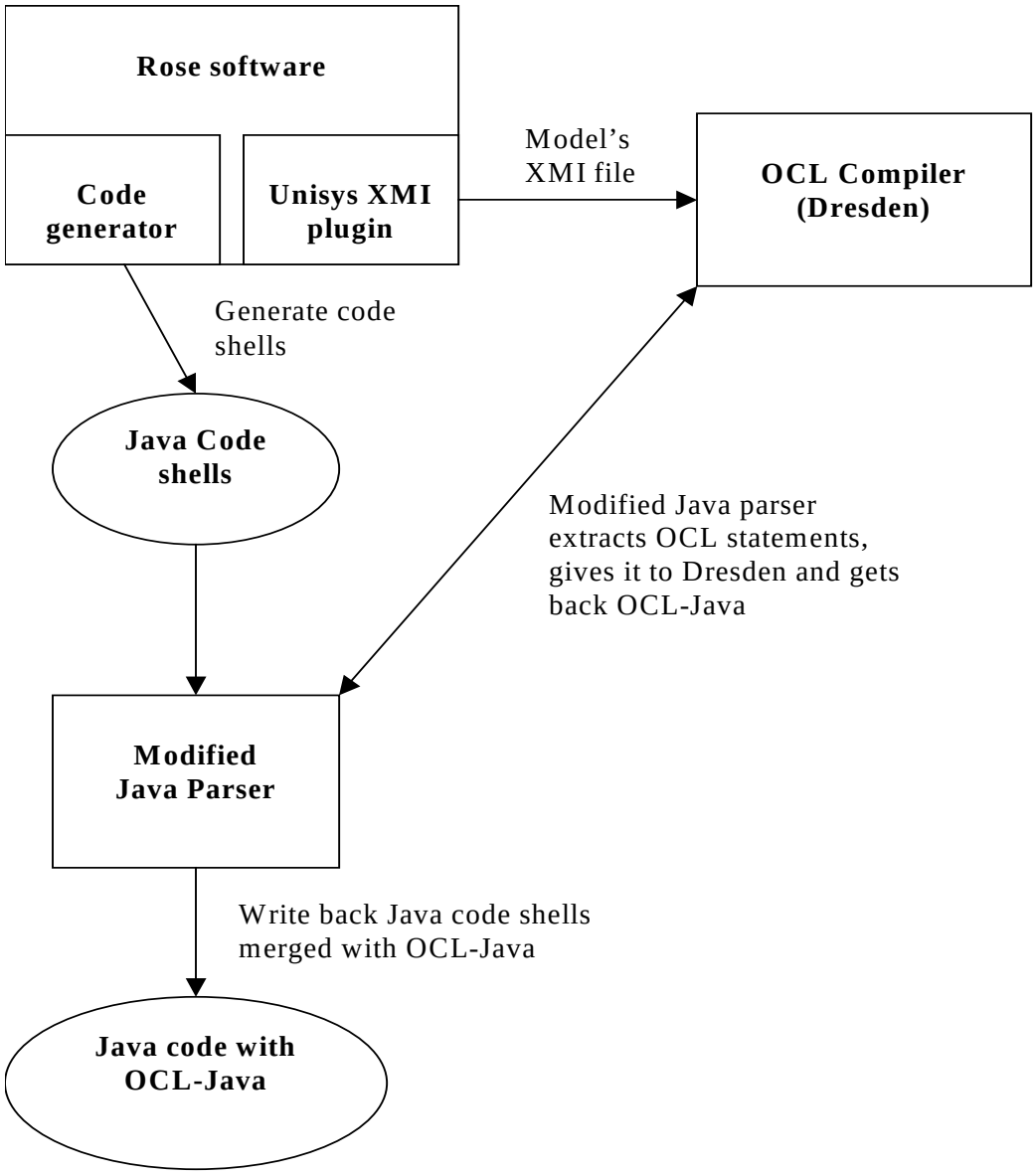


Figure 3 Approach three

Code patterns

One of the most important aspects of the merging of the Java code shells and the OCL-Java is the code patterns to be used during this procedure. The code pattern is the way in which the OCL-Java has to be inserted into the Java code shells. Code patterns

are needed for the following OCL constraints: the invariants, pre- and post-conditions. The requirement of the code pattern is that it should be able to effectively detect the OCL constraints violations and report them to other appropriate entities in the program. Also, the code patterns should be simple (so that it does not obfuscate the developer) and maintainable (so that it allows for future changes). It should also make sure that the model information, as represented in the code shells, is preserved to the maximum extent possible. This means that, if the code pattern necessitates the introduction of new methods, the number of methods introduced for the purpose of detecting constraint violations should be minimized.

The detection of the constraint violation could be achieved in a number of ways. However, in Java, the best method to do this is to throw an exception [10]. This is because, we can encapsulate all the error-related data (such as the stack trace, error message, constraint violated and the reason for the error) into the exception object. The calling method can then examine this object and take appropriate action (which could involve re-throwing the exception). The pre, post and the invariant constraints throw the exceptions – *PreconditionException*, *PostconditionException* and the *InvariantException*, respectively. We defined these classes and they inherit all properties of the Java `Exception` class [10], along with the information about the constraint violated and the type of constraint. Note that the above exceptions are derived from the class `Exception` and not `RuntimeException` to ensure that the calling application is aware that an exception could be generated due to a constraint violation and that it

should implement routines to properly handle them. The code pattern for each of the OCL constraint conditions is discussed below.

Precondition

A precondition in OCL can be specified in the following format:

```
context <className>::<methodName> (param1: Type, ...) : <returnType>
      pre: <OCL statement evaluating to boolean type>
```

In our scheme of entering OCL statements in the documentation fields, a precondition occurs in the Java code shell as a comment just above the method to which it applies. Since a precondition should be enforced before the method body is executed, the OCL-Java should be inserted before the first statement of the method. Thus, the modified class would appear as in figure 4. All code inserted as part of the code pattern is in bold font. The code in a regular font is the code emitted by the Rose code generator.

```
class <className>
{
      private ...
            ...

      public <returnType> <methodName> (param1: Type, ...) throws
      PreconditionException

      {
            <insert the OCL-Java for the precondition>

            boolean violated = <evaluate the OCL-Java code fragment as
            boolean>

            if (violated)
                  throw new PreconditionException(<OCL constraint
                  violated>);


            <method body>

      }
}
```

Figure 4 Precondition code pattern

Thus, when the precondition constraint is violated, a precondition exception is thrown to the calling method. Note that, in the case of constructor, precondition is meaningless, since the attributes typically get initialized in the body of the constructor, and thus no code is inserted for constructors.

Post-condition

The post-condition can be specified in the following format in OCL:

```
context <className>::<methodName> (param1: Type, ...) : <returnType>
      post: <OCL statement evaluating to boolean type or checking the
      value of the keyword
      'return'>
```

The pre and the post-condition, however, could be combined in a single context, too. This can be done by combining both the `pre`: and the `post`: constraints under the

same `context` specification. Like the precondition, the post condition also occurs as a comment just above the method's definition. With a post-condition, the OCL-Java code that checks for the constraint violation must be executed just before the method returns and no sooner. However, a method, before the insertion of OCL-Java, might return in two ways:

- Clean-return: The method might return normally with no exceptions. Since the method might have several return points, the code pattern must make sure that the post condition is evaluated, regardless of the return path.
- Return with Exception: The unmodified Java code shell might have methods that throw user-defined exceptions. The method thus might return to the caller via such exceptions. In such cases, it is desirable the OCL-Java code be not executed and the method returns via the user-defined exceptions. This necessitates that we devise a method to recognize if an exception has been thrown before we start the execution of the OCL-Java code.

In order to take care of the above two cases, the code pattern for the post condition constraint violation can be written as shown in figure 5. Code in bold indicates the code inserted by the modified Java parser. The code in a regular font is the code emitted by the Rose code generator.

```
class <className>
{
      private ...
            ...
      public <returnType> <methodName> (param1: Type, ...) throws
      PostconditionException, Exception

      {
            boolean exceptionThrown = false;
            try
            {
                  <method body>

            } catch(Exception e)
            {
                  exceptionThrown = true;
                  throw e;
            }
            finally
            {
                  if (!exceptionThrown)
                  {
                        <insert the OCL-Java for postcondition>

                        boolean violated = <evaluate the OCL-Java as
                        boolean>

                        if (violated)
                              throw new PostconditionException(<OCL
                              constraint violated>);
                  }
            }
      }
}
```

Figure 5 Post-condition code pattern

The code pattern above distinguishes between a clean-return and return with exceptions. The `exceptionThrown` boolean flag indicates if an exception is thrown by the method body or not. In order to catch the exceptions thrown by the method body, we encapsulate the entire method body within the `try` and `catch` block. The `finally` clause is *always* executed, irrespective of the manner in which the method returns. If an exception was thrown by the method body, then we do not process the

post-condition OCL-Java; instead we just re-throw the exception. Otherwise, we process the post condition OCL-Java. Also, note that we throw an exception of type `Exception` and not of the actual user-defined type. The following are the reasons for throwing a type of `Exception`:

- We thought of typecasting it to the runtime class name, but this kind of dynamic typecasting is not supported in Java.
- Although we throw an exception of type `Exception`, the calling method's `catch` [10] will do the nearest match to the runtime class name. Thus, in the calling method's `catch` clause, an attempt to the exact match to the runtime class name is made by the Java Virtual Machine [13], which is what we wanted. This effectively means that the run-time type of an object is maintained even if it is upcasted and the `catch` clause can be used to match with the exact exception type.

Thus, when no user-defined exceptions occur, the post-condition OCL-Java is evaluated and a post-condition exception is thrown if the constraint is violated. However, if a user-defined exception does occur, the post-condition OCL-Java is not executed and the user-defined exception is thrown to the calling method.

Invariant

The invariant can be stated in OCL in the following way:

```
context <className> inv: <OCL statement evaluating to boolean type>
```

Since the invariants are bound to a particular class, they occur in the Java code shells within the comments just above the class declaration. The invariants typically apply to the attributes of a class, though through the context, attributes of other classes can be included in the constraint. As per the OCL specification, it is necessary to make sure that the invariant constraint holds good throughout the class. This means that we should make sure that it applies to all the methods in the class. There are two different cases:

- Consider the Java code shells after they have had the programmer-written code inserted into them. This means that the invariant conditions should be checked in the programmer-written code whenever the value of the attribute on which the constraint is defined is changed. This requires that we implement a *checkFor* to the constraint violation. Such a checkFor would check for the attribute's value each time it is accessed. After such a check, if the invariant constraint is violated, the `checkFor` would throw an exception indicating the same. A code pattern using such a checkFor is shown in figure 6. In our example, an example `set` method for the attribute changes the attribute's value. We call the `checkFor` function just after this change. We thus note here that the `checkFor` method should be called whenever the attribute on which the constraint is enforced is changed. This requires that we scan the Java code shell (after the programmer has written his/her code) to intelligently determine such statements and insert a call to the `checkFor` method just after those statements. This requires a post-processing tool that can parse such Java files to identify statements that change the attribute's value.

- Consider the Java code shells that do not have programmer-written code. These are the kind of code shells that are typically generated from Rose's code generator. For such code shells, it is only possible to check for invariant constraint violation just before the method body starts and just before the method returns. This implies that, for this case, we combine the code patterns for the pre and the post condition. Code pattern for this is shown in figure 7. Code in bold indicates the code inserted by the modified Java parser.

```
public void checkFor_<attribute name> () throws InvariantException
{
      <insert the OCL-Java for the invariant attribute>
      boolean violated = <evaluate the OCL-Java as boolean>

      if (violated)
            throw new InvariantException(<OCL constraint violated>)
}

// an example method calling the checkFor_<attribute name>

public void set<attribute name>(<data type> newValue) throws Exception
{
      <attribute name> = newValue; // <attribute name> is changed
      checkFor_<attribute name>(); /* call the checkFor method just
      after <attribute name> is changed */

}
```

Figure 6 Invariant code pattern with checkFor

```
class <className>
{
      private     ...
            ...
      public <returnType> <methodName> (param1: Type, ...) throws
      InvariantException, Exception

      {

            <insert the OCL-Java for the invariant constraints>

            boolean violated = <evaluate the OCL-Java boolean
            expression>

            if (violated)
                  throw new InvariantException(<OCL constraint
                  violated>);


            boolean exceptionThrown = false;
            try
            {
                  <method body>

            } catch(Exception e)
            {
                  exceptionThrown = true;
                  throw e;
            }
            finally
            {
                  if (!exceptionThrown)
                  {
                        <insert the OCL-Java for the invariant
                        constraints>

                        boolean violated = <evaluate the OCL-Java
                        boolean expression>

                        if (violated)
                              throw new InvariantException(<OCL
                              constraint violated>);
                  }
            }
      }
}
```

Figure 7 Invariant code pattern

Typically, the Java code shells do not have any programmer-written code and hence the method's body will be minimal. In such cases, the code pattern in figure 7 is much easier to implement than the one in figure 6. Since we are trying out just the concepts, in our prototype, we implement the code pattern in figure 7.

IMPLEMENTATION NOTES

In order to test our concept and the design approach, we developed a prototype tool in Java. The tool is command-line based and can process a single file or an entire package. The Java code shells generated by Rose are modified to include the OCL-Java produced by a modified Dresden OCL compiler, and these modified files are written out with a special file tag, such as file names ending with `.ocl`. The resultant file can be readily compiled by a standard Java compiler.

To create the model and generate the code shells, we used the Rational Rose software. In order to generate the XMI file of the model, we used the Unisys XMI plug-in [27]. The OCL-Java was generated using the Dresden compiler [22] after it was modified to fix a variety of bugs. The Java parser was modified so that the OCL-Java was inserted into the Java code shells according to the code patterns. Some of the implementation notes and issues are presented in this section.

Java parser modifications

The Java parser forms an important component, since the merging of the code from the Java code shells and the OCL-Java is handled by this module. There are multiple aspects to accomplishing this. First, the Java parser must be modified so that it can recognize the OCL statements as output by the Rose code generator. Secondly, it must invoke the Dresden OCL compiler for these OCL statements and save the resulting Java code. Finally, it must insert the resulting code in the proper place(s) in the Java

source code output by the Java parser. Note that this is complicated somewhat by issues such as:

- Invariant constraints must be output in multiple places.
- The constraint output code is usually placed in a different location in the source code stream than the point at which Rose places the non-executable OCL statements.
- One must deal with nested class definitions and the propagation of invariants to nested classes.
- One must deal with packages with multiple source modules in them.

In this section, we describe how these issues were addressed.

Our modified Java parser works on the Java code shells generated from Rose. The Java parser first parses the Java code shells as per the Java grammar and produces the corresponding tokens. We will refer to this stage as the initial-parsing phase. Once this is done, we can process the different tokens in the order they appear in the Java code shell. It is during this token-processing stage that we accomplish most of the OCL-Java code insertion.

In the Java code shells, the OCL constraint statements appear as comments. For invariants, the constraints appear just before the class declaration. For the pre- and post-conditions, the constraints appear just before the method declaration on which they are enforced. Since the comments field typically also contains other information, we distinguish the constraints enclosed within a special tag, following the XML structure. An example is shown in figure 8.

```
<ocl>
context BankAccounts inv:
self.balance > 0
</ocl>
```

Figure 8 Example tagged OCL statement

In order to extract the OCL constraint statements and to insert OCL-Java, we have to recognize the beginning of classes and methods. The Java language grammar [2] specifies the rules and the productions of the Java language. We used the Java 1.4 grammar, with some parts of the grammar modified (without changing the language) as per the JavaCCtoHTML [2] grammar in order to enable marking of the tokens. While specifying the grammar in the JavaCC format (called the *jj* file), we specified the *look-ahead* [1] option, so that the parser can inspect the tokens beyond its current position which can be of great help in recognizing the method declaration construct. With the help of look-ahead, we marked the opening and the closing braces of the methods and the classes. This helped us to recognize the beginning and the end of methods and classes while we were processing the tokens generated by the parser. Knowing the beginning of the methods, we can insert the OCL-Java for the invariants and the preconditions, as per the code patterns. Similarly, knowing the end of the methods, we can insert the OCL-Java for the post-conditions and invariants.

Some of the post-condition OCL statements can be specified using the `result` keyword. The `result` keyword signifies the return value of the function. This gives rise to a couple of additional issues in handling these kinds of constraints.

First, the OCL-Java, generated by the Dresden OCL compiler, assumes that the result to be returned is assigned to a variable names `result`. This would require both that the programmer declare this variable to be of the return type and that the programmer assign the return value to `result` and then issue a `return result` statement. It is highly desired that the system not have to rely on the programmer remembering to do these operations. Thus, to overcome this problem, we had to insert a `result` declaration as part of the Java parser processing. That, in turn, required us to determine the various return types of all the methods. The return types of the methods were determined and cached in a data structure during the initial-parsing stage. During the token-processing stage, we retrieved these whenever we wanted to output the `result` declaration statement.

The second issue associated with the `result` variable that cannot be handled by modifying the Java parser. Since the programmer would typically add the method body after OCL-Java insertion, he/she would have to make sure that he first assigns the return value to the `result` variable and returns this `result` variable. This is quite an imposition on the programmer. Thus, we developed a tool, called the resultTool, that would perform this automatically. The resultTool looks for methods that have a post-condition constraint enforced on them that involves the `result` keyword. For the code written within all such methods, it searches for the `return` statement. For each of the `return` statement, it extracts the return expression, assigns it to the `result` variable and returns this `result` variable. An example of such a code snippet is shown in figure 9.

```
// Before processed by result tool

Integer result = null;

try {
      /* METHOD BODY GOES HERE */

      Integer addAmount = new Integer(500);
      return new Integer( 5 * addAmount.intValue());

} catch (Exception e) {
      exceptionThrown = true;
      throw e;
}



// After processed by the result tool

Integer result = null;

try {
      /* METHOD BODY GOES HERE */

      Integer addAmount = new Integer(500);
      result = new Integer(5 * addAmount.intValue());
      return result;

} catch (Exception e) {
      exceptionThrown = true;
      throw e;
}
```

Figure 9 Result tool processing example

There were several other modifications that went into the Java parser

- By default, the Java parser omits the comments in the Java code shell. We had to modify the grammar so that the comments are recognized as a single token, as opposed to every word in the comment appearing as a separate token. We extract the OCL statements from these tokens.

- The methods in the modified OCL-Java might or might not throw an exception based on the constraints that were defined on it. We had to modify the signature of the function to enable it to throw an exception. To achieve this, we had to analyze the signature of the method before the OCL-Java insertion and then decide if we needed to alter the signature. This was accomplished by searching for the `throws` clause to determine the existing exception list. If in this list, the required exception type was already present (such as `Exception` type), we simply output the original exception list unchanged. Otherwise, we add the required exception type and then output the exception list.
- Another issue arose with nested classes. Since nested classes occur in the same Java code shell file, we should clearly determine which invariant applies for which nested class. In order to accomplish this, we adopt the following logic: we initially create a stack, whose purpose is to store the invariants of all the enclosing classes, in the last-in-first-out manner. When we encounter a nested class, we push the invariants of the current class and consider only the invariants of the new nested class. We could not consider the invariant code for all its enclosing classes, since the OCL-Java generated from the Dresden compiler for the enclosing classes holds only for methods in those respective classes. When we pass the end of a nested class, we pop off the top of stack and consider this popped-off invariant for OCL-Java code insertion.
- The modified Java parser integrates the functionality provided by the Dresden OCL compiler by making use of the JAR files provided as part of the compiler.

The Java parser can thus be thought of having two distinct modules – one which generates the OCL-Java by calling the Dresden compiler functionalities and the other that inserts the OCL-Java in accordance with the code patterns.

- Another property of the modified Java parser is that it can process either a single file or an entire package. In the case of a package, there might be multiple files and directories. The directories are processed by traversing through the directory structure recursively and extracting the files in each such directory. The modified Java code shells written out are nicely formatted, enhancing readability.

The implementation introduces certain keywords that the designer/programmer must be aware of. They are:

- To enter OCL constraints in Rose: We use the <ocl> and </ocl> tags to mark the OCL constraints (specified within the "Documentation field" of the respective class or method) in Rose. The designer should not use this tag for any other purpose.
- For Post-conditions and Invariants: We use the boolean variable `exceptionThrown` as part of the code pattern. The programmer/designer is advised not to use a variable by that name in the actual argument list to the method in which it occurs.
- For post-conditions that involve the **result** keyword, the programmer/designer is advised not to use a variable by that name in the actual argument list to the method in which it occurs.

Dresden compiler modifications

The Dresden complier takes in the model file and the OCL constraint and gives out the OCL-Java. We found few issues that needed to be fixed for our purpose.

- The Dresden compiler does not support any of the Rose model file formats. However, it does accept the XMI format, and we can generate the XMI file of the model from Rose by using the Unisys XMI plug-in. However, there were inconsistencies with the XMI format that was produced by the XMI plug-in and those accepted by the Dresden compiler. This was mainly because of the non-conformance of the Dresden compiler with the XMI version that the Unisys plug-in generated. We had to modify the XMI parser in the Dresden compiler so that the XMI tags are consistent with the XMI specification. Some of the tags that we modified were the associations (association beginning and end tags), generalization tag and the package tag. These tags were modified as per the XMI version 1.0 which supports UML version 1.3.
- Another issue was that the Dresden compiler did not support nested classes and the package context in the XMI model file. We had to modify the XMI parser code in the compiler so that it could recognize the nested classes and the package context.
- As mentioned before, the OCL specification supports some pre-defined basic types. However, we found that the compiler did not support the basic types. So, we had to modify the compiler to recognize tokens representing the basic types.

- More of an inconsistency than an issue is that the Dresden compiler accepts OCL constraints for methods in which the formal parameter list is separated by semicolon, as against comma, which the OCL specification requires. This is due to the code within the Dresden compiler that parses the OCL constraint statements. We were unable to alter this behavior since we could not find the original OCL grammar specification that was used to produce the code that parsed the OCL constraint statements.

Thus, most of the modifications were to ensure that the Rose model in the XMI format is completely supported by the Dresden compiler.

EXAMPLES AND TESTING

In order to test our concept and the developed prototype, we have developed test models to test each of the features that were included in the prototype system. The intention of this testing is to check the various types of OCL statements, the Java code shells generated and the merged code with the OCL-Java. In this section, we will describe the different models, give few OCL constraint examples and demonstrate the working of the prototype tool.

Example Rose model

Figure 10 shows the Rose model that we used to test the various OCL constructs. It is similar to the example model in [17]. A person's relationship with his company and bank accounts he owns is shown. Every person can have multiple bank accounts. All employees are persons who are employed with a company. The company has more than one employee and several job types. Each employee is associated with a job type. A job type defines the employee's designation in the company. The `Employment` class describes the employee's employment details. The `GlobalConstants` is used to define some global attributes that apply throughout the model. Note that the `JobType` is an association class [18].

The object-oriented features like inheritance, associations and the association class that are used frequently are present in the model. We wrote some OCL constraint statements on the classes' methods and attributes. Some of the constraint statements are discussed in the next sub-section. All the OCL invariant constraint statements are placed

in the “Documentation” field of the concerned class while OCL pre and post-condition constraint statements are placed in the “Documentation” field of the concerned method.

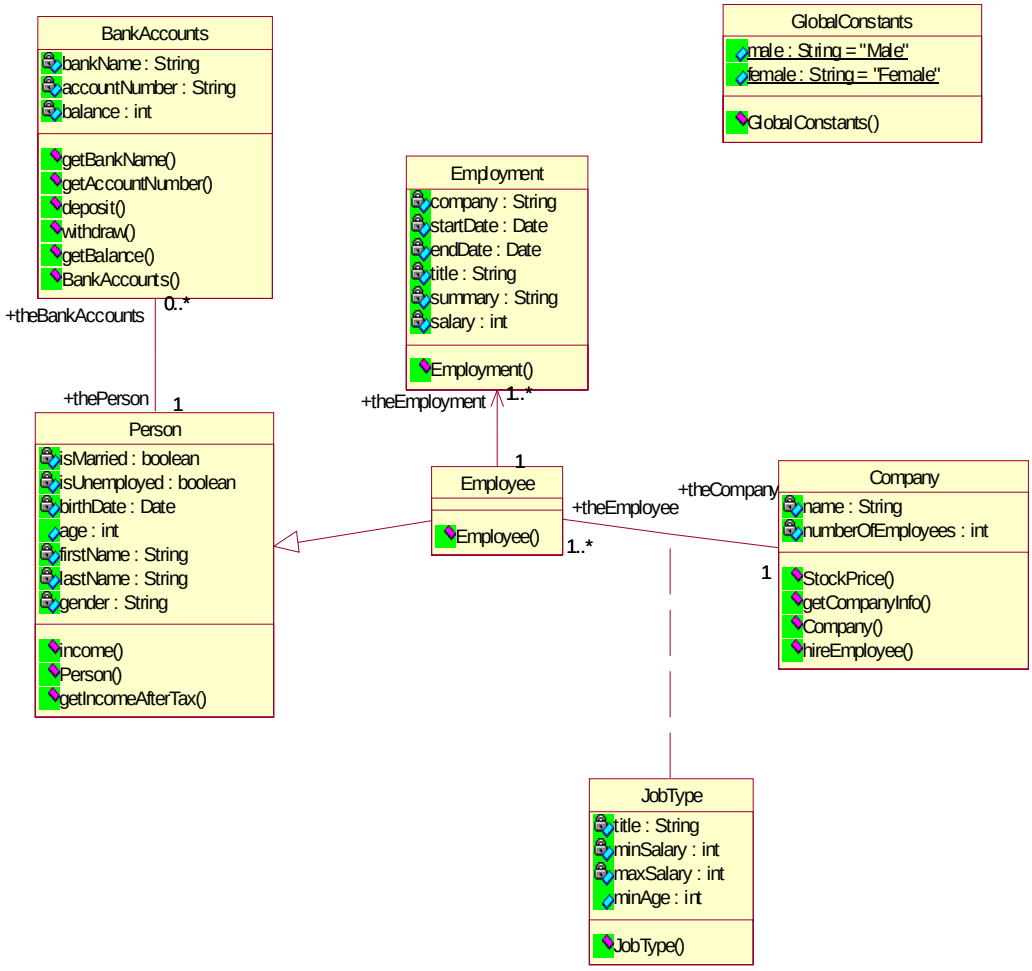


Figure 10 An example Rose model

Example OCL statements

Consider the class `BankAccounts`. The attribute `balance`, which signifies the remaining balance, should never be below zero (or some positive quantity like 25). Thus, we can specify an invariant on balance like the one shown in figure 11. Note that this invariant is placed in the "Documentation" filed of class `BankAccounts`.

```
<ocl>
context BankAccounts inv:
self.balance > 0
</ocl>
```

Figure 11 Simple Invariant example

The method `withdraw` performs the withdrawal of amount from a bank account. We can define a precondition on this operation to confirm whether the amount that is being withdrawn is greater than zero. This is shown in figure 12.

```
context BankAccounts::withdraw(amount:Real):Integer
pre: amount > 0
```

Figure 12 Simple precondition example

In OCL, we can access members from other classes too. Figure 13 shows a precondition that makes sure that we hire only persons who are above a minimum age for a particular job type. This precondition is imposed on the `hireEmployee` method of the class `Company`. Note that `JobType` is an association class.

```
context Company::hireEmployee(name: Person; job: JobType) :
Boolean
pre: name.age > job.minAge
```

Figure 13 OCL accessing other class' members

The post-condition can be specified on the methods to ensure that the return value of the method conforms to the action it performs. In the case of `withdraw` method in `BankAccounts`, we can ensure that the return value is the new balance. The OCL constraint for such a post condition is shown in figure 14.

```
context BankAccounts::withdraw(amount:Real):Integer
post: result = balance
```

Figure 14 Post condition example

A collection represents a group of objects in OCL. As mentioned before in *Background* section, we can use the pre-defined methods on the collection. For example, the `Company` has an array of employees, denoted by `theEmployee` (as shown in figure 10). We can check if the attribute `numberOfEmployees` is indeed equal to the `theEmployee`'s size with the constraint in figure 15.

```
context Company inv:
self.numberOfEmployees=theEmployee->size
```

Figure 15 OCL collection example

We can define new variables in OCL using the `let` keyword. Using this, we can, as a trivial example, check if a person's last name's first character is a capital letter or not. This is shown in figure 16. As before, this constraint is entered into the documentation field of the class `Person`.

```
context Person inv:
let firstAlpha:String=lastName.substring(1,1) in
firstAlpha = firstAlpha.toUpper
```

Figure 16 Using `let` in OCL

Code Examples – before and after OCL-Java

In order to show how the code looks before and after insertion of OCL-Java, consider the example of `BankAccounts.java`. The `BankAccounts.java` is generated from the Rational Rose's code generator and looks as shown in figure 17.

```
//Source file: C:\\data\\Pramod\\OCL\\ex3-roseCode\\BankAccounts.java

/**
<ocl>
context BankAccounts inv:
self.balance > 0
</ocl>
 */
public class BankAccounts
{
   private String bankName;
   private String accountNumber;
   private int balance;
   public Person thePerson;

   /**
   @roseuid 3F8B134002F0
```

Figure 17 BankAccounts.java before insertion

```
 */

public BankAccounts()
{

}

/**
@return String
@roseuid 3F8B075B016C
 */
public String getBankName()
{
 return null;
}

/**
@return String
@roseuid 3F8B080D0223
 */
public String getAccountNumber()
{
 return null;
}

/**
<ocl>
context BankAccounts::deposit(amount: Integer): Boolean
pre: amount > 0
</ocl>
@param amount
@return boolean
@roseuid 3F8B084F03C9
 */
public boolean deposit(int amount)
{
 return true;
}

/**
<ocl>
context BankAccounts::withdraw(amount:Real): Integer
pre: amount > 0
</ocl>
<ocl>
context BankAccounts::withdraw(amount:Real): Integer
post: result = balance
</ocl>

@param amount
@return int
```

Figure 17 Continued

```
   @roseuid 3F8B088A0138
    */
   public int withdraw(Float amount)
   {
    return 0;
   }

   /**
   <ocl>
   context BankAccounts::getBalance():Integer
   post: result = self.balance
   </ocl>
   @return Integer
   @roseuid 3F8B09860201
    */
   public Integer getBalance()
   {
    return null;
   }
}
```

Figure 17 Continued

Note that the OCL statements appear *just* before the corresponding class or methods. The method's body is typically defined after the code is generated from Rose. From figure 17, we notice that the withdraw method has all the three constraints – invariant, pre and post conditions. We now show the snippets of how the withdraw method looks like after the OCL-Java.

For the invariant condition, the code as shown in figure 18 is generated.

```
{
      final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
            tudresden.ocl.lib.Ocl.toOclAnyImpl(
                  tudresden.ocl.lib.Ocl.getFor(this));
      final tudresden.ocl.lib.OclInteger tudOclNode1 =
            tudresden.ocl.lib.Ocl.toOclInteger(
                  tudOclNode0.getFeature("balance"));
      final tudresden.ocl.lib.OclInteger tudOclNode2 =
            new tudresden.ocl.lib.OclInteger(0);
      final tudresden.ocl.lib.OclBoolean tudOclNode3 =
            tudOclNode1.isGreaterThan(tudOclNode2);
      if (!tudOclNode3.isTrue())
            throw new InvariantException(
                  "  context BankAccounts inv:  self.balance > 0
                  ",
                  "Invariant constraint violation");
}
```

Figure 18 Invariant code

We notice that the entire code for invariant is encapsulated in a block. This is to avoid re-definition of the variables used in evaluating the OCL constraint. The code in figure 18 is generated from the Dresden compiler and is inserted into the Java code shell in figure 17 using the code patterns mentioned before. We observe that the result variable is tudOclNode3. This is evaluated to check if the constraint has been violated or not. If violated, we throw the `InvariantException`, mentioning the reason and the constraint violated. The complete code for `BankAccounts.java` after OCL-Java insertion is attached as Appendix A.

The precondition code is inserted as per the code pattern. However, we notice from Appendix A that, since both precondition and invariant are present, we need to first insert the invariant OCL-Java block followed by the precondition OCL-Java block. The separate block used for each of them avoids the same-variable name conflicts.

For the post-condition, we have defined the result variable as per the return value of the method, which is integer, `int`. This can be seen as shown in figure 19. Since invariant OCL-Java must also be executed before method's exit, there are two code blocks within the finally block – one for the invariant and the other for the post-condition.

```
int result = 0;

try {
      /* METHOD BODY GOES HERE */

      return 0;

} catch (Exception e) {
      exceptionThrown = true;
      throw e;
}
```

Figure 19 Post condition result handling

From the above code samples, we can see that the invariants, pre and post condition constraints have been correctly transformed into the corresponding OCL-Java and inserted according to the code patterns. In order to show that the constraint violation is actually detected, we will show a small demonstration, again using the modified (Java code shells with OCL-Java) `BankAccounts.java`.

Demonstrating constraint violation

In order illustrate the result of violating the invariant constraint, we initialize the balance attribute to a negative value, say, -100. Figure 20 shows the Invariant exception that was thrown when a new object of class `BankAccounts` was created with this

negative balance attribute. Note that the invariant exception is thrown in the constructor of the `BankAccounts` class. This exception was then caught and the details of the exception were printed out by the calling method.

To demonstrate the pre and the post condition violation, we will use the `withdraw` method as an example. We call the withdraw method with a negative amount that is being withdrawn. The calling code is shown in figure 21. The violation that is caused is shown in figure 22.

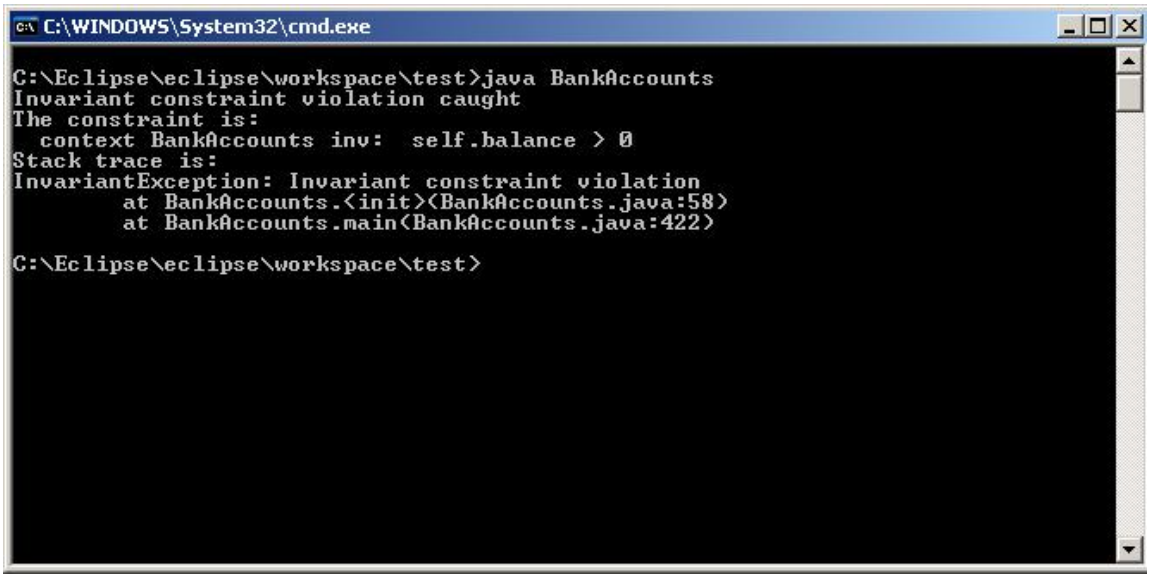


Figure 20 Invariant violation caught

```
try
{
      BankAccounts ba = new BankAccounts();
      ba.withdraw(new Float(-200.0));

} catch (PreconditionException e)
{
      System.err.println("Precondition constraint violation
      caught");
      System.err.println(
            "The constraint is:\n" + e.oclPre + "\nStack
            trace is:");
      e.printStackTrace();
}
```

Figure 21 Sample calling code for precondition violation

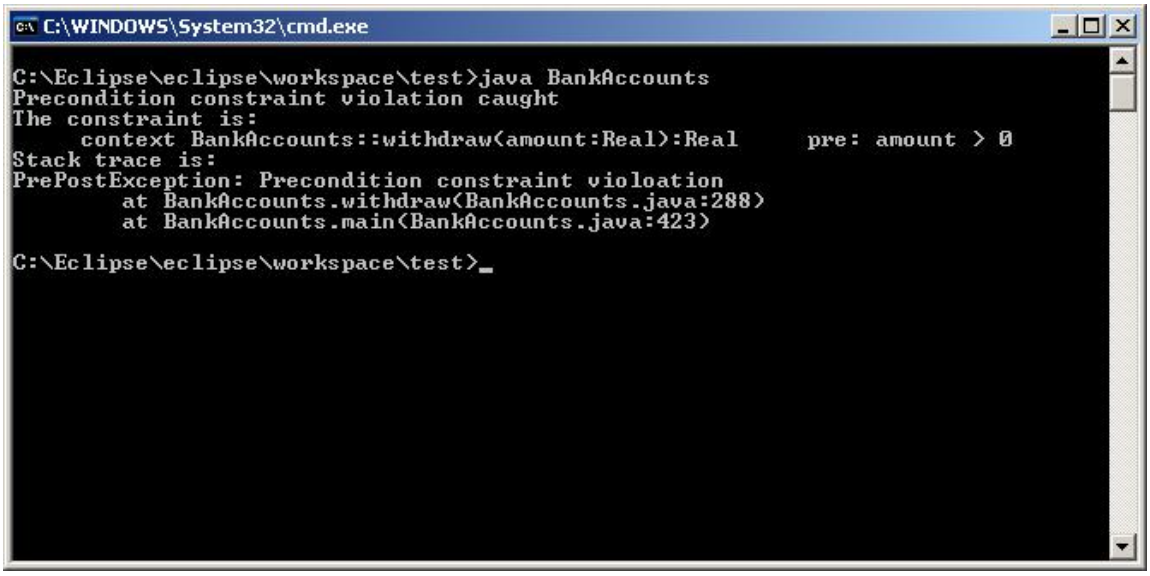


Figure 22 Precondition violation caught

Figure 23 shows an erroneous code that a programmer might write for the `withdraw` method. Figure 24 shows the post-condition violation when the `withdraw` is called with a positive withdraw amount, but the return from the method is different from the balance value, which is due to the erroneous code.

```
try {
      /* METHOD BODY GOES HERE */
      balance = balance – amount.intValue();

      return 0;// programmer returns 0 instead of ‘balance’
}
```

Figure 23 Erroneous code written for the `withdraw` method

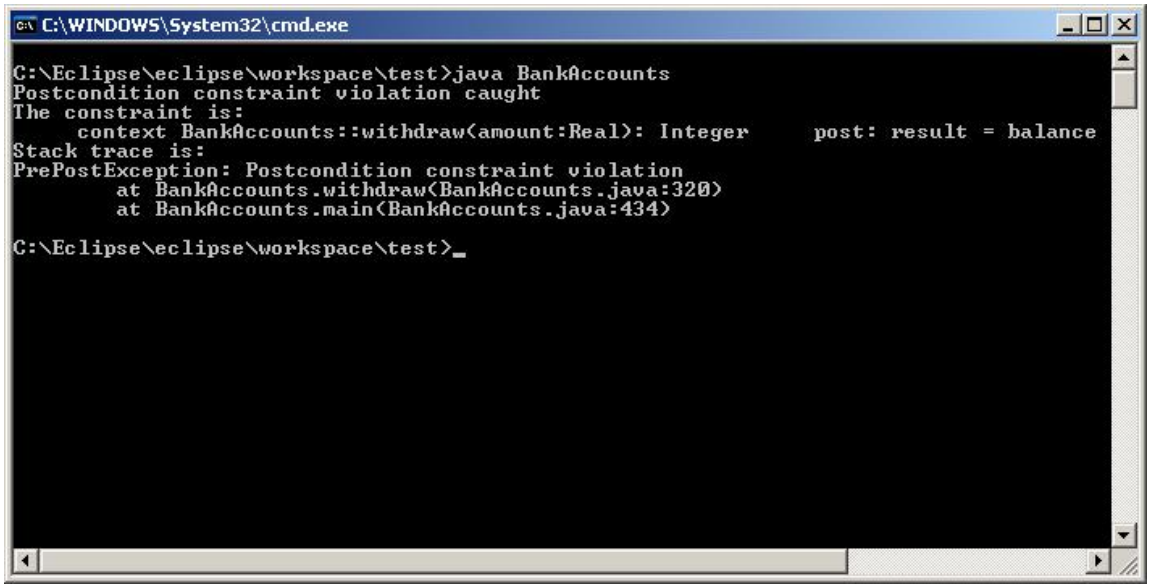


Figure 24 Post condition violation caught

From the above examples and demonstration, it is clear that constraint violations can be effectively caught using the modified Java code shell into which the OCL-Java has been inserted. It is worth noting that the above detection of constraint violations works irrespective of the code that the programmer might write after generating the modified Java code shells.

More examples

We now illustrate example OCL statements on the nested classes and the association class.

Consider the OCL statement as in figure 13. It shows how we can access the association class, `JobType`, from another class, `Company`. The OCL-Java that checks for the constraint is shown in figure 25. Note how the code can access another class' member; in this case, `JobType`'s `minAge` attribute.

```
            {
                  final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                              tudresden.ocl.lib.Ocl.getFor(this));
                  final tudresden.ocl.lib.OclAnyImpl tudOclOpPar0 =
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                              tudresden.ocl.lib.Ocl.getFor(name));
                  final tudresden.ocl.lib.OclAnyImpl tudOclOpPar1 =
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                              tudresden.ocl.lib.Ocl.getFor(job));
                  final tudresden.ocl.lib.OclInteger tudOclNode1 =
                        tudresden.ocl.lib.Ocl.toOclInteger(
                              tudOclOpPar0.getFeature("age"));
                  final tudresden.ocl.lib.OclInteger tudOclNode2 =
                        tudresden.ocl.lib.Ocl.toOclInteger(
                              tudOclOpPar1.getFeature("minAge"));
                  final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                        tudOclNode1.isGreaterThan(tudOclNode2);
                  if (!tudOclNode3.isTrue())
                        throw new PreconditionException(
                              "     context Company::hireEmployee(name:
Person; job: JobType) : Boolean     pre: name.age > job.minAge     ",
                              7,
                              "Precondition constraint violation");
            }
```

Figure 25 OCL-Java for the example constraint in figure 13

Now consider the case of Nested classes. Appendix C shows an example nested class. In processing the OCL constraints on nested classes, we do not consider the constraints of the enclosing classes. We could not consider the invariant code for all its enclosing classes, since the OCL-Java generated from the Dresden compiler for the enclosing classes holds only for methods in those respective classes. Thus, only those constraints imposed on the class that is currently being parsed by the Java parser are considered. In the Nested class model example shown, although the invariant constraint is specified on `testAttr1` in `Mainclass` and `testAttr2` in `NestedClass`, in the method `op2` of `NestedClass`, the invariant code for only `testAttr2` is inserted and not for both `testAttr1` and `testAttr2`. The code snippet for `op2` is shown in figure 26.

```
public int op2(int arg2) throws Exception
{
  {
      final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                  tudresden.ocl.lib.Ocl.toOclAnyImpl(
                  tudresden.ocl.lib.Ocl.getFor(this));
      final tudresden.ocl.lib.OclInteger tudOclNode1 =
                  tudresden.ocl.lib.Ocl.toOclInteger(
                  tudOclNode0.getFeature("testAttr2"));
      final tudresden.ocl.lib.OclInteger tudOclNode2 =
                  new tudresden.ocl.lib.OclInteger(0);
      final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                  tudOclNode1.isGreaterThan(tudOclNode2);
      if (!tudOclNode3.isTrue())
                  throw new InvariantException(
                  "     context NestedClass inv:
                  testAttr2 > 0     ",
                  "Invariant constraint violation");
  }

  boolean exceptionThrown = false;
  // rest of the code for the method.
```

Figure 26 Invaraint handling for nested class's method

CONCLUSION AND FUTURE WORK

The automated insertion of code that can detect constraint violations during run-time can prove to be extremely useful in any software system, including real-time and critical systems such as the Space Shuttle software. The Object Constraint Language provides a method by which the designer can specify constraints early in the design of the system. With the work performed in this thesis, it is possible to transform such non-executable OCL statements specified in the Rational Rose software into executable code, inserted according to code patterns, that can effectively detect violations of constraints. We demonstrated the use of various types of constraints by using a test model that implemented different object-oriented features such as inheritance, associations, association classes, nested classes and packages.

The prototype developed was to test our concepts and design. It does not, however, consider all types of OCL expressions. For example, the OCL specification defines a package context for the OCL statements themselves, which is not supported in the prototype. This kind of a package context is necessary if the OCL statements are specified in a separate file, rather than with explicit context (such as specifying in the "Documentation" field, as was done in our prototype). We could extend our prototype to include such package contexts, but prefer to have the OCL integrated with the model rather than specified separately.

Another candidate for improvement is that we could also devise an alternate way to handle the insertion of OCL-Java for invariants of nested classes. Instead of the current method of considering only the invariants of the current nested class, we could

consider to output the code for invariants of all its enclosing classes. This would require us to alter the Dresden compiler so that the OCL-Java generated can be used in nested classes as well as its enclosing classes.

Yet another improvement would be to implement the code pattern as shown in figure 6 for invariants. This would help in detecting invariant violations anywhere in the programmer-written code.

The OCL considered in our work as part of the UML 1.4 specifications. However, the OCL specification is constantly being improved. The latest draft of UML 2.0 [18] contains quite a few changes to the OCL specification. One of the most notable changes is that it explains what OCL messages are and how they can be used. OCL messages are similar to notification messages and can help specify if any attribute value changes. This could impact our invariant code pattern shown in figure 6. Developing code patterns and the OCL-Java for such features can be quite challenging and could be a valuable addition to our prototype.

Automatic insertion of code that can detect constraint violations can greatly help to ensure that a software system's run-time behavior is as close to the behavior specified (during the design phase) as possible. It frees the programmer from writing routine code to detect constraint violations and hence increases his/her productivity. When an OCL constraint violation occurs, it gives the programmer, the freedom to choose how he/she can deal with it; thus making the software system to graciously deal with constraint violations, rather than crashing all together. We have shown that it is feasible to create a

tool that can insert code to detect constraint violations and be used by designers and developers alike to develop highly reliable software systems.

REFERENCES

[1] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[2] CoBase research group, "Java Grammars for JavaCC," 2002, http://www.cobase.cs.ucla.edu/pub/javacc/#Jsection

[3] M. Dahm, "The CrazyBeans project," 2001, http://crazybeans.sourceforge.net/

[4] A. Duncan and U. Hölzle, "Adding Contracts to Java with Handshake," Technical Report *TRCS98-32*, Computer Science Department, University of California, Santa Barbara, December 1998.

[5] Eclipse, "The Eclipse platform," 2003, http://eclipse.org

[6] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Second Edition, Addison-Wesley, Reading, MA, 1999.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*. First Edition, Addison-Wesley, Reading, MA, 1995.

[8] H. Hußmann, B. Demuth and F. Finger, "Modular architecture for a toolset supporting OCL," *Science of Computer Programming*, vol. 44, no.1, pp. 51-69, 2002.

[9] IBM Corporation, "IBM OCL Parser," 1997, http://www-3.ibm.com/software/awdtools/library/standards/ocl-download.html

[10] B. Joy, G. Steele, J. Gosling and G. Bracha, *Java Language Specification*. Second edition, Addison-Wesley, Reading, MA, 2002.

[11] M. Karaorman, U. Hölzle and J. Bruno, “jContractor: A Reflective Java library to support Design by Contract,” Technical Report *TRCS98-31*, Computer Science Department, University of California, Santa Barbara, December 1998.

[12] R. Kramer, “The Java design by contract tool,” *Proc. of the Technology of Object-Oriented Languages and Systems*, pp. 295-307, IEEE, Washington DC, 1998.

[13] T. Lindholm and F. Yelling, *The Java Virtual Machine Specification*. Second Edition, Addison-Wesley, Reading, MA, 1999.

[14] Man Machine Systems, “The JMSAssert Tool,” 2002, http://www.mmsindia.com/JMSAssert.html

[15] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[16] Novosoft, “The NSUML library,” 2001, http://nsuml.sourceforge.net/

[17] Object Management Group, “OCL Specification,” 2003, http://www.omg.org/docs/formal/03-03-13.pdf

[18] Object Management Group, “Unified Modeling Language,” 2004, http://www.omg.org/uml/

[19] Object Management Group, “XMI specification,” 2002, http://www.omg.org/technology/documents/formal/xmi.htm

[20] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. Fifth edition, McGraw Hill, New York, 2000.

[21] Rational Rose, “Rational Rose software,” 2002, http://www.rational.com

[22] Technische Universität Dresden, “Dresden OCL toolkit and injector,” 2000, http://dresden-ocl.sourceforge.net

[23] The Cybernetic Intelligence, “OCL compiler,” 2001, http://www.cybernetic.org/prodocl15.htm

[24] The LCI team, “The OCL Environment,” 2003, http://lci.cs.ubbcluj.ro/ocle/

[25] Tigris Community, “The ArgoUML project and its limitations,” 2001, http://argouml.tigris.org/ and http://argouml.tigris.org/documentation/umlsupport/

[26] A. Toval, V. Requena, and J. L. Fernandez, “Emerging OCL tools,” *Software and Systems Modeling*, vol. 2, no.4, pp. 248-261, 2003.

[27] Unisys Corporation, “The Unisys XMI plug in,” 2003, http://www-106.ibm.com/developerworks/rational/library/content/03July/2500/2834/Rose/rational_rose.html

[28] University of Oldenburg, “The Jass Tool,” 2001, http://csd.informatik.uni-oldenburg.de/~jass/

[29] B. Verheecke and R. Straeten, “Specifying and implementing the operational use of constraints in object-oriented applications”, *Proc. of the Fortieth International Conference on Tools Pacific*, pp. 23-32, Australian Computer Society, Sydney, Australia, 2002.

[30] S. Viswanadha, “The JavaCC Parser,” 2003, http://javacc.dev.java.net/

[31] W. Wang, “The Java Tree Builder,” 2000, http://www.cs.purdue.edu/jtb/

[32] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, Reading, MA, 1999.

[33] R. Wiebicke, “Utility Support for Checking OCL Business Rules in Java Programs,” Diploma Thesis, Dept. of Computer Science, Dresden University, Germany, 2000.

APPENDIX A

This appendix lists the code shell for BankAccounts.java. The code shell is complete with code fragments to detect run-time violations of the constraints.

```
//Source file: C:\\data\\Pramod\\OCL\\ex3-roseCode\\BankAccounts.java

/**
<ocl>
context BankAccounts inv:
self.balance > 0
</ocl>
 */
import tudresden.ocl.*;
import java.util.*;

public class BankAccounts {
        private String bankName;
        private String accountNumber;
        private int balance;
        public Person thePerson;

        /**
        @roseuid 3F8B134002F0
         */
        public BankAccounts() throws InvariantException, Exception {
                boolean exceptionThrown = false;
                try {
                        /* METHOD BODY GOES HERE */

                } catch (Exception e) {
                        exceptionThrown = true;
                        throw e;
                } finally {
                        if (!exceptionThrown) {
                                {
                                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                        tudresden.ocl.lib.Ocl.getFor(this));
                                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                                tudresden.ocl.lib.Ocl.toOclInteger(
                                                        tudOclNode0.getFeature("balance"));
                                        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                                new tudresden.ocl.lib.OclInteger(0);
                                        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                                tudOclNode1.isGreaterThan(tudOclNode2);
                                        if (!tudOclNode3.isTrue())
                                                throw new InvariantException(
                                                        "  context BankAccounts inv:
                                                        self.balance > 0  ",
                                                        "Invariant constraint violation");
                                }

                        }
                }
        }

        /**
        @return String
        @roseuid 3F8B075B016C
         */
        public String getBankName() throws InvariantException, Exception {
                {
```

```
                final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                tudresden.ocl.lib.Ocl.getFor(this));
                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                        tudresden.ocl.lib.Ocl.toOclInteger(
                                tudOclNode0.getFeature("balance"));
                final tudresden.ocl.lib.OclInteger tudOclNode2 =
                        new tudresden.ocl.lib.OclInteger(0);
                final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                        tudOclNode1.isGreaterThan(tudOclNode2);
                if (!tudOclNode3.isTrue())
                        throw new InvariantException(
                                "  context BankAccounts inv:  self.balance > 0  ",
                                "Invariant constraint violation");
        }

        boolean exceptionThrown = false;
        try {
                /* METHOD BODY GOES HERE */

                return null;

        } catch (Exception e) {
                exceptionThrown = true;
                throw e;
        } finally {
                if (!exceptionThrown) {
                        {
                                final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                tudresden.ocl.lib.Ocl.getFor(this));
                                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                        tudresden.ocl.lib.Ocl.toOclInteger(
                                                tudOclNode0.getFeature("balance"));
                                final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                        new tudresden.ocl.lib.OclInteger(0);
                                final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                        tudOclNode1.isGreaterThan(tudOclNode2);
                                if (!tudOclNode3.isTrue())
                                        throw new InvariantException(
                                                "  context BankAccounts inv:
                                                self.balance > 0  ",
                                                "Invariant constraint violation");
                        }

                }
        }
}

/**
@return String
@roseuid 3F8B080D0223
 */
public String getAccountNumber() throws InvariantException, Exception {
        {
                final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                tudresden.ocl.lib.Ocl.getFor(this));
                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                        tudresden.ocl.lib.Ocl.toOclInteger(
                                tudOclNode0.getFeature("balance"));
                final tudresden.ocl.lib.OclInteger tudOclNode2 =
                        new tudresden.ocl.lib.OclInteger(0);
                final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                        tudOclNode1.isGreaterThan(tudOclNode2);
                if (!tudOclNode3.isTrue())
                        throw new InvariantException(
```

```
                                "  context BankAccounts inv:  self.balance > 0  ",
                                "Invariant constraint violation");
                }

                boolean exceptionThrown = false;
                try {
                        /* METHOD BODY GOES HERE */

                        return null;

                } catch (Exception e) {
                        exceptionThrown = true;
                        throw e;
                } finally {
                        if (!exceptionThrown) {
                                {
                                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                        tudresden.ocl.lib.Ocl.getFor(this));
                                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                                tudresden.ocl.lib.Ocl.toOclInteger(
                                                        tudOclNode0.getFeature("balance"));
                                        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                                new tudresden.ocl.lib.OclInteger(0);
                                        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                                tudOclNode1.isGreaterThan(tudOclNode2);
                                        if (!tudOclNode3.isTrue())
                                                throw new InvariantException(
                                                        "  context BankAccounts inv:
                                                        self.balance > 0  ",
                                                        "Invariant constraint violation");
                                }

                        }
                }
        }

        /**
        <ocl>
        context BankAccounts::deposit(amount: Integer): Boolean
        pre: amount > 0
        </ocl>
        @param amount
        @return boolean
        @roseuid 3F8B084F03C9
         */
        public boolean deposit(int amount) throws PreCondException, InvariantException,
Exception {
                {
                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                        tudresden.ocl.lib.Ocl.getFor(this));
                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                tudresden.ocl.lib.Ocl.toOclInteger(
                                        tudOclNode0.getFeature("balance"));
                        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                new tudresden.ocl.lib.OclInteger(0);
                        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                tudOclNode1.isGreaterThan(tudOclNode2);
                        if (!tudOclNode3.isTrue())
                                throw new InvariantException(
                                        "  context BankAccounts inv:  self.balance > 0  ",
                                        "Invariant constraint violation");
                }

                {
                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
```

```
                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                tudresden.ocl.lib.Ocl.getFor(this));
                final tudresden.ocl.lib.OclInteger tudOclOpPar0 =
                        tudresden.ocl.lib.Ocl.toOclInteger(
                                tudresden.ocl.lib.Ocl.getFor(amount));
                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                        new tudresden.ocl.lib.OclInteger(0);
                final tudresden.ocl.lib.OclBoolean tudOclNode2 =
                        tudOclOpPar0.isGreaterThan(tudOclNode1);
                if (!tudOclNode2.isTrue())
                        throw new PreCondException(
                                "     context BankAccounts::deposit(amount:
                                Integer): Boolean     pre: amount > 0     ",
                                7,
                                "Precondition constraint violation");
        }

        boolean exceptionThrown = false;
        try {
                /* METHOD BODY GOES HERE */

                return true;

        } catch (Exception e) {
                exceptionThrown = true;
                throw e;
        } finally {
                if (!exceptionThrown) {
                        {
                                final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                tudresden.ocl.lib.Ocl.getFor(this));
                                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                        tudresden.ocl.lib.Ocl.toOclInteger(
                                                tudOclNode0.getFeature("balance"));
                                final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                        new tudresden.ocl.lib.OclInteger(0);
                                final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                        tudOclNode1.isGreaterThan(tudOclNode2);
                                if (!tudOclNode3.isTrue())
                                        throw new InvariantException(
                                                "  context BankAccounts inv:
                                                self.balance > 0  ",
                                                "Invariant constraint violation");
                        }

                }
        }
    }

    /**
    <ocl>
    context BankAccounts::withdraw(amount:Real): Integer
    pre: amount > 0
    </ocl>
    <ocl>
    context BankAccounts::withdraw(amount:Real): Integer
    post: result = balance
    </ocl>

    @param amount
    @return int
    @roseuid 3F8B088A0138
     */
    public int withdraw(Float amount) throws PreCondException, PostCondException,
InvariantException, Exception {
        {
```

```
        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                        tudresden.ocl.lib.Ocl.getFor(this));
        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                tudresden.ocl.lib.Ocl.toOclInteger(
                        tudOclNode0.getFeature("balance"));
        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                new tudresden.ocl.lib.OclInteger(0);
        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                tudOclNode1.isGreaterThan(tudOclNode2);
        if (!tudOclNode3.isTrue())
                throw new InvariantException(
                        "  context BankAccounts inv:  self.balance > 0  ",
                        "Invariant constraint violation");
}

{
        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                        tudresden.ocl.lib.Ocl.getFor(this));
        final tudresden.ocl.lib.OclReal tudOclOpPar0 =
                tudresden.ocl.lib.Ocl.toOclReal(
                        tudresden.ocl.lib.Ocl.getFor(amount));
        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                new tudresden.ocl.lib.OclInteger(0);
        final tudresden.ocl.lib.OclBoolean tudOclNode2 =
                tudOclOpPar0.isGreaterThan(tudOclNode1);
        if (!tudOclNode2.isTrue())
                throw new PreCondException(
                        "     context BankAccounts::withdraw(amount:Real):
                        Integer     pre: amount > 0     ",
                        7,
                        "Precondition constraint violation");
}

boolean exceptionThrown = false;

// The 'result' variable is used by the OCL code.
/* Since 'result' is a key word in OCL, do NOT override the below 'result'
variable.*/
/* It may require appropriate initialization. Further, if used in OCL post
condition, make sure that 'result' is assigned the return value of the
method before returning from the function. Instead, you can also use our
ResultTool*/
int result = 0;

try {
        /* METHOD BODY GOES HERE */

        return 0;

} catch (Exception e) {
        exceptionThrown = true;
        throw e;
} finally {
        if (!exceptionThrown) {
                {
                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                        tudresden.ocl.lib.Ocl.getFor(this));
                        final tudresden.ocl.lib.OclReal tudOclOpPar0 =
                                tudresden.ocl.lib.Ocl.toOclReal(
                                tudresden.ocl.lib.Ocl.getFor(amount));
                        final tudresden.ocl.lib.OclReal tudOclResult0 =
                                tudresden.ocl.lib.Ocl.toOclReal(
                                tudresden.ocl.lib.Ocl.getFor(result));
                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
```

```
                                        tudresden.ocl.lib.Ocl.toOclInteger(
                                                tudOclNode0.getFeature("balance"));
                                final tudresden.ocl.lib.OclBoolean tudOclNode2 =
                                        tudOclResult0.isEqualTo(tudOclNode1);
                                if (!tudOclNode2.isTrue())
                                        throw new PostCondException("context
                                        BankAccounts::withdraw(amount:Real): Integer
                                        post: result = balance     ",
                                                21,
                                        "Postcondition constraint violation");
                        }

                        {
                                final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                        tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                tudresden.ocl.lib.Ocl.getFor(this));
                                final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                        tudresden.ocl.lib.Ocl.toOclInteger(
                                                tudOclNode0.getFeature("balance"));
                                final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                        new tudresden.ocl.lib.OclInteger(0);
                                final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                        tudOclNode1.isGreaterThan(tudOclNode2);
                                if (!tudOclNode3.isTrue())
                                        throw new InvariantException(
                                                "  context BankAccounts inv:
                                                self.balance > 0  ",
                                                "Invariant constraint violation");
                        }

                }
        }
        }

        /**
        <ocl>
        context BankAccounts::getBalance():Integer
        post: result = self.balance
        </ocl>
        @return Integer
        @roseuid 3F8B09860201
         */
        public Integer getBalance() throws PostCondException, InvariantException,
Exception {
                {
                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                        tudresden.ocl.lib.Ocl.getFor(this));
                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                tudresden.ocl.lib.Ocl.toOclInteger(
                                        tudOclNode0.getFeature("balance"));
                        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                new tudresden.ocl.lib.OclInteger(0);
                        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                tudOclNode1.isGreaterThan(tudOclNode2);
                        if (!tudOclNode3.isTrue())
                                throw new InvariantException(
                                        "  context BankAccounts inv:  self.balance > 0  ",
                                        "Invariant constraint violation");
                }

                boolean exceptionThrown = false;

                // The 'result' variable is used by the OCL code.
                /* Since 'result' is a key word in OCL, do NOT override the below 'result'
                variable.*/
                /* It may require appropriate initialization. Further, if used in OCL post
```

```
                condition, make sure that 'result' is assigned the return value of the
                method before returning from the function. Instead, you can also use our
                ResultTool*/
                Integer result = null;

                try {
                        /* METHOD BODY GOES HERE */

                        return null;

                } catch (Exception e) {
                        exceptionThrown = true;
                        throw e;
                } finally {
                        if (!exceptionThrown) {
                                {
                                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                        tudresden.ocl.lib.Ocl.getFor(this));
                                        final tudresden.ocl.lib.OclInteger tudOclResult0 =
                                                tudresden.ocl.lib.Ocl.toOclInteger(
                                                tudresden.ocl.lib.Ocl.getFor(result));
                                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                                tudresden.ocl.lib.Ocl.toOclInteger(
                                                        tudOclNode0.getFeature("balance"));
                                        final tudresden.ocl.lib.OclBoolean tudOclNode2 =
                                                tudOclResult0.isEqualTo(tudOclNode1);
                                        if (!tudOclNode2.isTrue())
                                                throw new PostCondException("     context
                                                BankAccounts::getBalance():Integer     post:
                                                result = self.balance     ",
                                                21,
                                                “Postcondition constraint violation");
                                }

                                {
                                        final tudresden.ocl.lib.OclAnyImpl tudOclNode0 =
                                                tudresden.ocl.lib.Ocl.toOclAnyImpl(
                                                        tudresden.ocl.lib.Ocl.getFor(this));
                                        final tudresden.ocl.lib.OclInteger tudOclNode1 =
                                                tudresden.ocl.lib.Ocl.toOclInteger(
                                                        tudOclNode0.getFeature("balance"));
                                        final tudresden.ocl.lib.OclInteger tudOclNode2 =
                                                new tudresden.ocl.lib.OclInteger(0);
                                        final tudresden.ocl.lib.OclBoolean tudOclNode3 =
                                                tudOclNode1.isGreaterThan(tudOclNode2);
                                        if (!tudOclNode3.isTrue())
                                                throw new InvariantException(
                                                "  context BankAccounts inv:
                                                self.balance > 0  ",
                                                "Invariant constraint violation");
                                }

                        }
                }
        }

}
```

APPENDIX B

This appendix lists the constraints that we used for testing. These constraints were entered into the Rose model's "Documentation" field of the class or the method.

```
<ocl>
context BankAccounts inv:
self.balance > 0
</ocl>

<ocl>
context BankAccounts::deposit(amount: Integer): Boolean
pre: amount > 0
</ocl>

<ocl>
context BankAccounts::withdraw(amount:Real): Integer
pre: amount > 0
</ocl>
<ocl>
context BankAccounts::withdraw(amount:Real): Integer
post: result = balance
</ocl>

<ocl>
context BankAccounts::getBalance():Integer
post: result = self.balance
</ocl>

<ocl>
context BankAccounts inv:
balance.oclIsTypeOf(Integer)
</ocl>
<ocl>
context BankAccounts inv:
self.getBalance()= self.balance
</ocl>

<ocl>
context Company inv:
numberOfEmployees > 0
</ocl>
<ocl>
context Company inv:
self.numberOfEmployees=theEmployee->size
</ocl>

<ocl>
context Company::hireEmployee(name: Person; job: JobType) : Boolean
pre: name.age > job.minAge
</ocl>

FOR THE NESTED CLASS:

<ocl>
context MainClass inv:
```

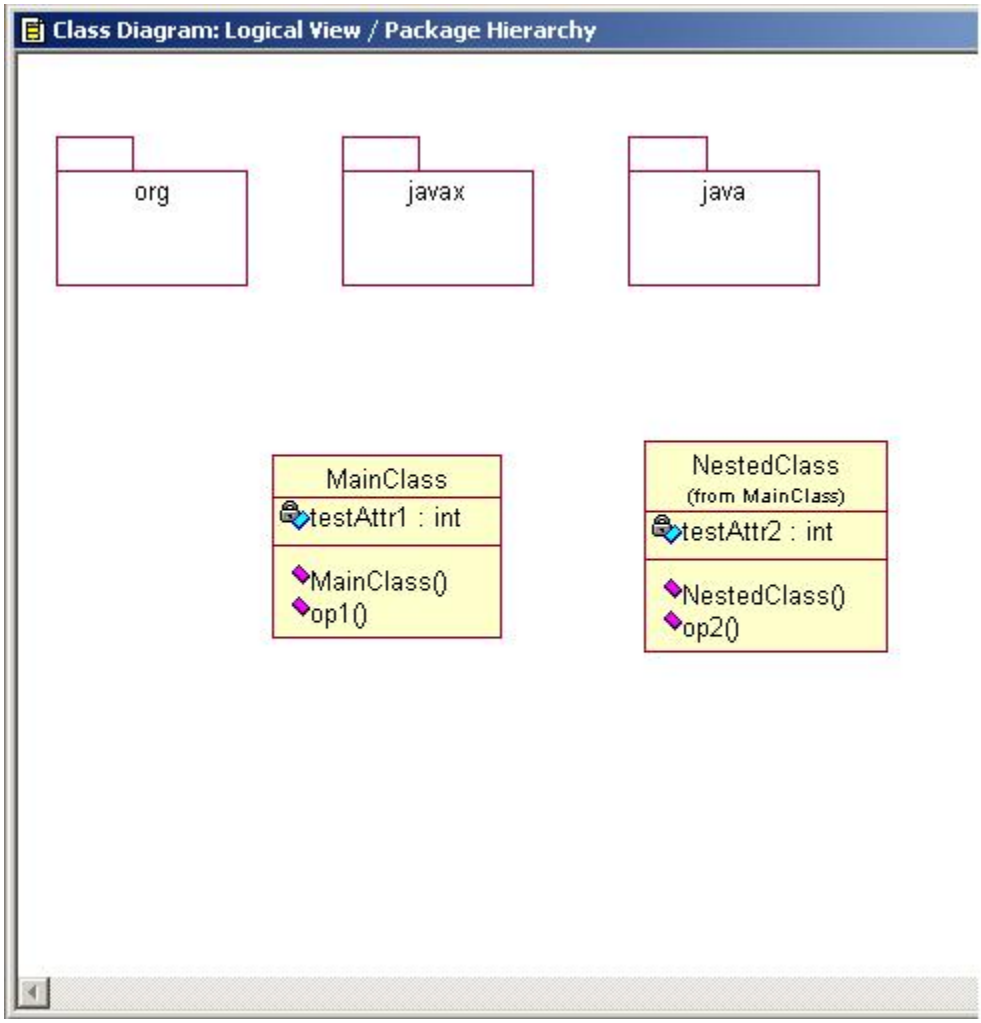
```
testAttr1 > 0
</ocl>

<ocl>
context MainClass::op1(arg1: Integer): Integer
pre: arg1 > 0
</ocl>

<ocl>
context NestedClass inv:
testAttr2 > 0
</ocl>
```

APPENDIX C

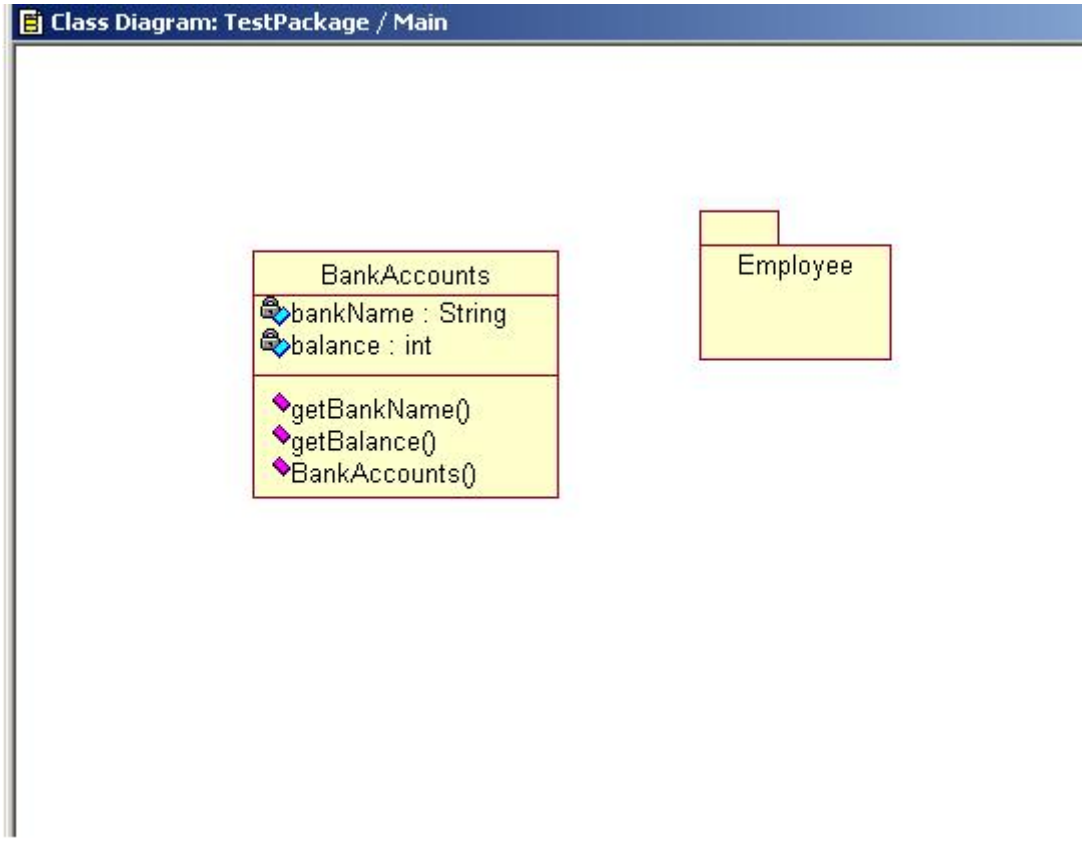
This appendix lists some of the screenshots.

The OCL constraint statements are inserted in the “Documentation” field in Rose as below:

The nested class example model is as shown below. It has a MainClass and a NestedClass.



The package example model is as shown below. We used this model to test if we could navigate through the package structure and insert the OCL-Java in the appropriate files correctly.



APPENDIX D

This appendix describes how to obtain the prototype tool, install and run it.

How to obtain the prototype tool

Please email Dr. Volz (volz@cs.tamu.edu) or Pramod Gurunath (pramodg@tamu.edu) for obtaining the software. You'll obtain a tar/zip file containing the jar file for the tool (ocl.jar) along with the associated jar files (xerces.jar, junit.jar, sablecc.jar, modifiedTudresden.jar, jdtcore.jar, runtime.jar, resources.jar). The Unisys XMI plug-in is also bundled.

How to install

Please perform the following steps:

1. Unzip/untar the zip/tar file into a directory.
2. Add the following jar files from the above directory to the classpath: modifiedTudresden.jar, xerces.jar, junit.jar, sablecc.jar, jdtcore.jar, runtime.jar, resources.jar and ocl.jar.
3. If you do not have the Unisys XMI plug-in for Rose, please download it from http://www-106.ibm.com/developerworks/rational/library/content/03July/2500/2834/Rose/rational_rose.html. Alternately, you can find the plug-in in the zip/tar file mailed to you. Install the XMI plug-in.

How to run

1. Develop a Rose model and insert your OCL constraints. For invariants, enter the constraints in the "Documentation" field of the class. For pre and post-conditions, enter the constraints in the "Documentation" field of the respective methods. Embed OCL within <ocl> and </ocl> tags.
2. Generate Java code shells from Rose.
3. Without loading the subunits (like `java`, `org` packages) for the Rose model, generate the XMI file by performing the following Menu actions in Rose: Tools->UML 1.3 XMI Addin->UML 1.3 XMI Export. Choose "Export Diagrams", "Treat <<type>> as <<datatype>>", "XMI 1.0" and output file basis as "Model". Hit "OK". Save the XMI file generated.
4. Run the tool in the JOCL folder. Type: java edu.tamu.ocl.OCLHandler <XMI file name generated in step 3> <Java code shell name OR a directory name>
5. The modified Java code will be found in the same directory used in step 4. The modified Java code file names will end in ".ocl".

VITA

Name:	Pramod Gurunath
Permanent Address:	c/o Dr. Volz, Department of Computer Science, Texas A&M University, College Station, TX – 77843-3112
Education:	B.S. in Computer Science, Bangalore University, India, 2000
Professional Experience:	Software Developer, Sun Microsystems, 2000-2002. Summer Intern, Sun Microsystems, Summer 2003.
Conference Papers:	"Generic Distributed Problem Solving Using JMX Technology," To appear in the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, 2004.