Diploma Thesis:
Utility Support for Checking OCL Business Rules in Java Programs

Ralf Wiebicke

December 2000

Copyright

Copyright © 2000 Ralf Wiebicke.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at http://www.gnu.org/copyleft/fdl.html.

The source code developed together with this paper is Copyright © 2000 Ralf Wiebicke and published under the GNU Lesser General Public License (LGPL). Additional source code was developed by Steffen Zschaler under LGPL.

Availability

This document is available at http://rw7.de/ralf/diplom00/intro.html in several electronic forms including LaTeX, Postscript, PDF, HTML and the original kLyX version.

The source code developed together with this paper is available at http://dresden-ocl.sourceforge.net/.

Modifications

This version includes several post-submission updates as listed below. For the submission version see the CVS repository.

- The paper moved to http://rw7.de/ralf/diplom00/intro.html.
- Versant provided a fix for the bug mentioned in chapter 5 and appendix C.3.
- Removed `--xmi-model` in B.2, was superfluous and caused some trouble.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 Task . . . 5
- 1.3 Organisation of This Work . . . 5

2 Related Work **7**
- 2.1 Runtime Constraint Checking . . . 7
- 2.2 Reverse Engineering . . . 8
- 2.3 Other Related Work . . . 8

3 Code Instrumentation **10**
- 3.1 Results of Code Generation . . . 10
  - 3.1.1 Preparation and Transfer Fragments . . . 11
- 3.2 Requirements and Design Decisions . . . 11
  - 3.2.1 Reversable Modification . . . 11
  - 3.2.2 Embedding Constraints in Java Source Code. . . . 12
  - 3.2.3 Checking the Element Type. . . . 13
- 3.3 Code Insertion . . . 13
  - 3.3.1 A Simple Approach . . . 13
  - 3.3.2 Wrapping Methods . . . 14
  - 3.3.3 Wrapping Constructors . . . 15
  - 3.3.4 Avoiding the Wrapper Loop . . . 16
  - 3.3.5 Cleaning the Code . . . 17
  - 3.3.6 Design of the Java Parser . . . 18
  - 3.3.7 Comparison to Jass . . . 18
  - 3.3.8 Comparison to iContract . . . 19
  - 3.3.9 Comparison to Byte Code Instrumentation . . . 20
  - 3.3.10 Summary . . . 21
- 3.4 Scope of Invariants . . . 21
- 3.5 Caching Results of Invariants . . . 22
  - 3.5.1 Design . . . 22
  - 3.5.2 Implementation . . . 23
  - 3.5.3 Detecting Collection Modification . . . 24

4 Model Information **27**
- 4.1 Representing Element Types . . . 27
- 4.2 Reverse Engineering . . . 29
  - 4.2.1 Source Code Analysis . . . 29

4.2.2 Runtime Analysis . . . 30
4.2.3 Byte Code Analysis . . . 30
4.2.4 Comparison . . . 31
4.2.5 Summary . . . 32

5 Industrial Example **33**

6 Summary **35**

7 Outlook **36**
7.1 Inheriting Constraints . . . 36
7.2 Integration with CASE Tools . . . 37
7.2.1 Code Generation . . . 37
7.2.2 Reverse Engineering . . . 38
7.3 Others . . . 38

A Maintance of the OCL Compiler **40**
A.1 Reflection Facade and OCL Library . . . 40
A.1.1 Polymorphism of Operation Parameters . . . 40
A.1.2 Mandatory Name Adapters . . . 40
A.1.3 Qualified Associations . . . 41
A.1.4 Type Mapping from OCL to Java. . . . 41
A.2 OCL Library . . . 41
A.2.1 Undefined Values . . . 41
A.2.2 Java Null Values . . . 42
A.3 Type Checker . . . 42
A.4 Java Code Generator . . . 42
A.4.1 Code Fragments for @pre . . . 42
A.4.2 Explicit Package Qualifiers. . . . 44

B Usage of the OCL Tool **45**
B.1 Example . . . 45
B.2 Reference . . . 46

C Code Examples **49**
C.1 Unreachable Post Condition Code in iContract . . . 49
C.2 Return Opcodes in Java Byte Code . . . 50
C.3 The Problem with Versant Database . . . 51

Chapter 1

Introduction

This work describes the design and implementation of runtime verification of OCL (Object Constraint Language) constraints in java programs. Therefore, the OCL compiler developed by Frank Finger [FF00] was extended into a java source code instrumentation tool.

1.1 Motivation

Many different approaches for increasing software quality have been developed in the past. Few of them have experienced wide spread usage in industry. One of them is Structural Programming, another one is the Object Oriented Paradigm. And there is just another one, which convinces through its simpleness: Check Statements (or `assertions` in C++).

Suppose a method for removing all objects from a collection:

```
void clear()
{
  // some complex handling of
  // hash tables or trees ...
}
```

Now, how does one verify, that the (supposedly complex) implementation is correct? Starting up the debugger and check for a few cases? Is Russian Roulette. Building some automatic test cases? Getting better. But why not make sure, that the critical code works correctly for the rest of its life:

```
void clear()
{
  // some complex handling of
  // hash tables or trees ...
  assert(size()==0)
}
```

The `assert` statement terminates the program, if the given expression evaluates to false. Otherwise it does nothing. Many bugs would be detected when showing

up for the first time[1]. Additionally it needs only a compiler switch to disable all assertions, thus removing any runtime penalty for release versions. As described in [SM93g], software quality can be improved dramatically by using assertions whenever possible. This meets with practical experiences of the author.

Unfortunately, there is no `assert` statement in Java. A similar functionality could be achieved using exceptions, but there would be no possibility to globally disable assertions.

Furthermore, assertions are just a special case of a much more powerful concept: Design by Contract (DbC). A good introduction is given in [TP98]. In DbC the example assertion above is transformed into a postcondition of the method:

```
/**
   @postcondition: size()==0
*/
void clear()
{
  // some complex handling of
  // hash tables or trees ...
}
```

The tool developed with this paper aims to support verification of design constraints as shown above. It instruments java source code, so that the instrumented code checks its own constraints on runtime.

1.2 Task

This work aims to extend the OCL compiler developed by Frank Finger with a java source code instrumentation tool. [FF00] section 3.6 already provides a list of requirements for such a tool. Additional attention is paid to java programs, where no UML class diagrams are available. The tool should be implemented to a sufficient extent. This includes maintance of the existing OCL compiler. For the java code provided by the industrial partner, net-linx AG, a small set of typical OCL constraints will be developed and experimented with.

1.3 Organisation of This Work

Chapter 2 lists work related to this paper, particularly software aiming for similar functionality. Chapters 3 and 4 present the design and implementation of the two major extensions of the OCL compiler developed in this paper: the java source code instrumentation and the completed model information for OCL. Chapter 5 reports experiences made using the tool on an industrial strength project. Chapter 6 summarizes the results of this work, while chapter 7 points out directions for future work.

Appendix A lists all modifications of the OCL compiler made during this work. Appendix B contains a short manual of the software together with an illustrative example. Finally, some very detailed descriptions have been shifted into appendix C to keep the main text clear.

[1] The rare case, where a bug in `size()` happens to hide a bug in `clear()` is neglected here.

This work comes with a CD, containing a current snapshot of the developed software, an electronic version of this paper and software and literature referred to in this work, where available.

Chapter 2

Related Work

This chapter lists both papers and software related to this diploma thesis. For some of the work there is a detailed comparison in subsequent chapters.

2.1 Runtime Constraint Checking

This section lists tools, which perform runtime constraint checking on java programs, more or less similar to the tool developed in chapter 3.

JMSAssert ([MMS]) provides OCL support for java. Constraints are embedded into javadoc comments. The tool links into the JVM to make the constraints checked. This approach does not involve source code modification. This makes it easier to use, but also platform dependent (currently Windows only). It also requires just-in-time compilers to be switched off. Binaries are available at no cost.

Several approaches instrument java byte code to make constraints checked, such as Handshake ([DH98]) and jContractor ([KHB98]). Section 3.3.9 discusses, whether the approach of source code instrumentation presented in this paper is adaptable to byte code instrumentation. Handshake uses separate text files with a non-standard syntax to express class invariants and pre/postconditions. jContractor implements constraints with dedicated java methods. Neither Handshake nor jContractor binaries are publicly available.

iContract ([RK98]) is a preprocessor for java. It instruments java source code to check constraints. It supports a subset of OCL. Constraints are embedded into javadoc comments. iContracts source code instrumentation is analyzed in detail in section 3.3.8. Binaries are available at no cost.

For some applications OCL is just too powerful. A simpler approach is demonstrated in [KSR00]. It implements a number of predefined constraint types, such as *numeric range* or *ordering of arrays*. For instance to have an attribute `age` constrained to positive values, one just adds a method `getAgeMinValue() {return 0;}`. Most OCL constraints used in the development of the OCL toolkit also could have been expressed with such simple means. Constrained classes must be valid JavaBeans. Also, the class must announce the modification of attributes manually to have the constraints reevaluated. KBeans is released under GPL. Additionally there is a GUI for simulating an object population and checking constraints against it.

Jass ([JASS]) is a preprocessor for java assertions developed at the University of Oldenburg ([DB99]). Apart from class invariants and method pre- and postconditions it provides check statements (like assert() in C++) and loop invariants and variants. Assertions are expressed in java, extended by universal and existential quantifiers. Section 3.3.7 analyses the code instrumentation of Jass in detail. Jass is available under GPL.

The following table summarizes the tools introduced above. The last line shows the tool developed with this paper.

	Constraint Source	Verification Method	Availability
JMSAssert	OCL in javadoc	links JVM (platform dependent)	binary
Handshake	proprietary language	byte code instr. / proxy system library	not available
jContractor	java methods	byte code instr. / class loader	not available
iContract	OCL in javadoc	source code instr.	binary
KBeans	predefined constraint types / java methods	in special environment	GPL
Jass	java fragments in javadoc	source code instr.	GPL
Dresden Toolkit	OCL in javadoc	source code instr. (reversible)	LGPL

2.2 Reverse Engineering

This section lists work related to reverse engineering needed in chapter 4.

A powerful approach to reverse engineering has been developed at the MIT [JW99]. The tool Superwomble extracts an object model from java byte code. Object models are roughly a subset of UML class diagrams, featuring inheritance and object associations. An important challenge for the analysis is the detection of element types of container attributes. The tool performs this very efficiently, without requiring any additional help from the user. Thus, it complements the two approaches presented in this paper. A detailed comparison is provided in section 4.2.

JVision [OI] produces class diagrams from java source or byte code. It's easy to use and has nice auto-layout. But it does not handle associations in any way. Collection attributes are simply shown as attributes. Instead it analyses, which classes instantiate/use each other. This is not nearly as useful as associations.

2.3 Other Related Work

Cybernetic Intelligence develops an OCL compiler ([CI]). The current prototype claims to support syntax checking only. Type checking is under development. Frontends are available for Select Enterprise and Rational Rose.

Elixir ([ET]) claims OCL support in it's CASE tool and its java IDE. The CASE tool provides an OCL text field only, without any syntax checking. For the IDE a plugin is provided to integrate iContract.

Several approaches implement OCL upon object repositories, such as USE [RG00] and ModelRun [BS]. The object repositories can be populated and animated visually, with OCL constraints continuesly being checked.

There is a universal code instrumentation toolkit ([CMA]) available for java. It parses java files into parse trees, preserving white space and comments. The parse tree can be modified and written back into the file. There are various applications for this, including tracing/profiling of program execution. The code instrumentation developed in this paper could probably be realized using this toolkit. However, the parser analyses the complete java file, thus is much more heavy-weight than the parser developed with this paper. There is a test version available at no cost, limited in the size of source programs it can handle.

Chapter 3

Code Instrumentation

Insertion of generated code into java programs is the main subject of this paper. Such an automatic source code transformation is commonly referred to as code instrumentation. In this paper it covers anything beyond code generation, to get a java program checking its own constraints. For an idea, where code generation ends and instrumentation starts, see section 3.1.

This is followed by an analysis of requirements for the code instrumentation and resulting design decisions in section 3.2. Section 3.3 describes the solution in detail.

Finally sections 3.4 and 3.5 discuss the more fundamental issue, when and how often invariants have to be checked.

3.1 Results of Code Generation

The java code generator developed in [FF00] produces a set of code fragments[1]. These code fragments have the following properties:

Property	
Constrained type	The class this constraint applies to.
Kind	Specifies, whether this fragment is an invariant, a pre- or a postcondition or a transfer or preparation fragment for a postcondition.
Constrained operation	The operation, this constraint applies to (not valid for invariants).
Code	Contains the actual java code to be executed.
Result variable	Specifies the name of the boolean variable, which contains the result of the OCL expression after code execution.

For each postcondition containing a @pre expression there are two additional code fragments called preparation and transfer. See below.

[1] see documentation of class `tudresden.ocl.codegen.CodeFragment`.

3.1.1 Preparation and Transfer Fragments

The meaning of preparation and transfer fragments is explained on a dramatically simplified example.

Suppose a post condition for operation `employ()`, that leaves the attribute `age` unchanged:

```
context Person::employ()
post: age=age@pre
```

The following code fragments will be produced:

Kind	Code
Transfer	`int node1;`
Preparation	`node1=this.age;`
Post Condition	`int node2=this.age;` `boolean result=(node1==node2);`

Typically these fragments would be inserted as follows:

```
class Person
{
  void employ()
  {
    int node1;       // transfer fragment
    node1=this.age;  // preparation fragment
    // original code of employ()
    // post condition fragment
    node2=this.age;
    boolean result=(node1==node2);
  }
}
```

Note, that precise semantics of code fragments involving the @pre expression has been changed, so that the original meaning described in [FF00] section 7.1.2 is no longer fully correct. For a detailed comparison see section A.4.

3.2 Requirements and Design Decisions

This section analyzes the requirements for the code instrumentation and derives some fundamental design decisions.

3.2.1 Reversable Modification

The most important feature is the reversability of code instrumentation. It must be possible to

- clean the code tracelessly from all inserted fragments.

- redo the instrumentation on source code that has already been modified, for instance when constraints have been changed.
- edit the modified source code without losing all changes at the next instrumentation.

These requirements makes things quite a bit more difficult, but there are serious reasons for this. Otherwise there would be two versions of source code: the original and the modified version. This raises some unpleasant problems:

1. Configuration management must handle two source code trees.
2. Developers must be careful to edit the original version only.
3. Running the instrumentation is required after every change of the java source code, not only when the constraints have been changed.
4. Stack traces of runtime exceptions point to the modified source code. Developers must look for the corresponding place in the original version.

The implementation of reversable modification requires a strategy of minimally invasive modification. This is realized by two design decisions:

1. Method wrappers, explained detailed in section 3.3.
2. Explicit package qualifiers for the OCL library in the generated code. Otherwise, an `import` statement for the OCL library would be necessary. This would be just another spot, were the original source code had to be touched. Additionally, this may introduce name conflicts between OCL library and user code.

3.2.2 Embedding Constraints in Java Source Code.

It should be possible to embed constraints in the javadoc comments. The placement of embedded constraints implicates (and replaces) the context of the constraint. See the example below.

```
/**
   @invariant ageGreaterZero: age>0
*/
class Person
{
  int age;

  /**
     @postcondition: age=age@pre
  */
  void employ();
}
```

Constraints should be immediately visible to the editor of a java file. Also, this generally promotes the single source approach. The author is strongly

convinced, that constraints stored in an extra text file are too far away from attention.

Invariants may also be placed on an attribute or method of their context class. This is for convenience, since most invariants are clearly related to one specific attribute or method.

3.2.3 Checking the Element Type.

The instrumented code must check, that container attributes comply to the `@element-type` and `@key-type` javadoc tags. For collections the `@element-type` tag specifies the type of the objects allowed in the collection. For maps `@element-type` specifies the type of values, while `@key-type` defines the type of key objects. A detailed description is provided in section 4.1.

Note, that this feature may be used standalone, without OCL expressions at all. Then it provides a runtime check for typed collections.

3.3 Code Insertion

The main task of code instrumentation is to have some code executed immediately before and after all methods (and after all constructors too). This section describes, how this is done by the tool developed with this paper.

At first, section 3.3.1 intoduces a simple approach, and why this would not work. Sections 3.3.2 and 3.3.3 explain the approach followed in this work. A somewhat tricky caveat and how it is solved is worked out in 3.3.4. Section 3.3.5 shows, how it is managed to undo all modifications of the instrumentation. The java parser used to do all this is outlined in 3.3.6.

Sections 3.3.7, 3.3.8 and 3.3.9 compare this solution to three other tools approaching a similar task quite differently. This is summarized in 3.3.10.

3.3.1 A Simple Approach

A straight-forward solution would insert the code directly into the method. Consider the following method.

```
int someMethod(double x)
{
  // here comes the code.
  return result;
}
```

The generated code could be inserted like this:

```
int someMethod(double x)
{
  // some code checking invariants/preconditions.
  // here comes the code.
  // some code checking invariants/postconditions.
  return result;
}
```

But this raises some severe problems:

1. The code to be executed after the method (postcondition and invariants) has to be inserted before any return statement.

2. The post condition code must have the return value available. Instead of the `result` variable in the example above, there could be a complex expression. Such a return expression has to be computed in advance, if the post condition code refers to the return value or the return expression produces side effects.

3. There may be name conflicts between the original and the generated code, since the generated code defines local variables.

4. For methods with return type void it must be decided, whether the post condition code has to be inserted at the end of the method. This depends on whether the end of the method is a reachable point of code. For the decision it needs a complete control flow analysis of the method. Note, that if the post condition code is wrongly inserted at the end of the method, the java compiler will fail due to unreachable statements.

An implementation would need a complete java parser. The following code instrumentation would have to modify the original code at many different places and in a complicated way. This runs contrary to the strategy of minimally invasive modification as decided in section 3.2.1.

Additionally, item 4 requires much of the semantic analysis performed by a java compiler. This makes the simple approach very hard to implement.

Jass ([JASS]) and iContract ([RK98]) use this simple approach and encounter all the problems mentioned above. For a detailed comparison see sections 3.3.7 and 3.3.8.

Method wrappers solve all these problems in a nifty but simple way. This is introduced in the following section.

3.3.2 Wrapping Methods

Some code tells more than thousand words, so an example is used to explain. Consider the following method.

```
int someMethod(double x)
{
  // here comes the code.
}
```

This is transformed into two methods.

```
int someMethod_wrappedbyocl[2](double x)
{
  // here comes the code.
}

int someMethod(double x)
{
```

[2]This is not yet the full truth, see section 3.3.4.

```
  // some code checking invariants/preconditions.
  int result=someMethod_wrappedbyocl(x);
  // some code checking invariants/postconditions.
  return result;
}
```

Now let's have a look back at the problems encountered for the simple approach. None of them exist anymore.

- The code to be executed after the method has to be inserted once only.
- When the post condition code is executed, the return expression is already evaluated and ready to use.
- No name conflicts are possible, since user code and generated code are strictly separated into different methods.
- No control flow analysis is needed.

The user code is modified in a simple way: a suffix is appended to the method name. For an implementation a very fragmentary java parser is sufficient, which understands "java signature level" only. This signature level covers anything outside of method bodies and attribute initializers. This is a very small part of the java language and easily to be analyzed by a hand-crafted parser. See section 3.3.6 how easy it is.

This kind of method wrapping still has a problem, which has been called the "wrapper loop" in this paper. Section 3.3.4 shows how this is solved.

3.3.3 Wrapping Constructors

Another transformation is used for constructors, since they cannot be renamed. Suppose an example constructor.

```
SomeClass(String x)
{
  // here comes the code
}
```

Instead of renaming, the original constructor gets an additional dummy argument.

```
SomeClass(String x, Dummy[3] oclwrapperdummy)
{
  // here comes the code
}

SomeClass(String x)
{
  this(x, (Dummy)null);
  // some code checking invariants.
}
```

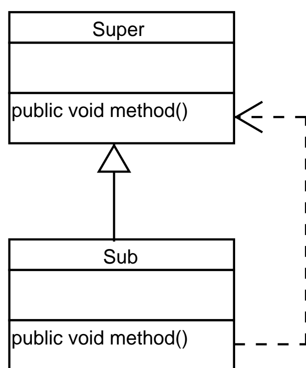
[3]Actually this is class `tudresden.ocl.injection.lib.WrapperDummy`.

A special case occurs if a class doesn't provide any constructors. Then the java compiler generates a default constructor as specified in [GJS96] section 8.6.7. This default constructor cannot be wrapped. Instead it is replaced by an explicit constructor with the same access modifier as the generated default constructor would get.

3.3.4 Avoiding the Wrapper Loop

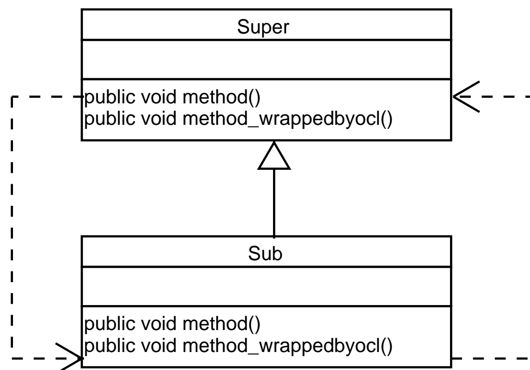
Wrapping methods as described in section 3.3.2 causes a problem for a special situation. This sections describes this situation and provides a solution.

The critical situation is shown in the figure below:



The dotted arrow represents a method call: `Sub.method()` contains a statement `super.method();` somewhere.

The code instrumentation changes the structure as shown below:



The arrows show the problem: there's an infinite loop of method calls. In detail the following happens:

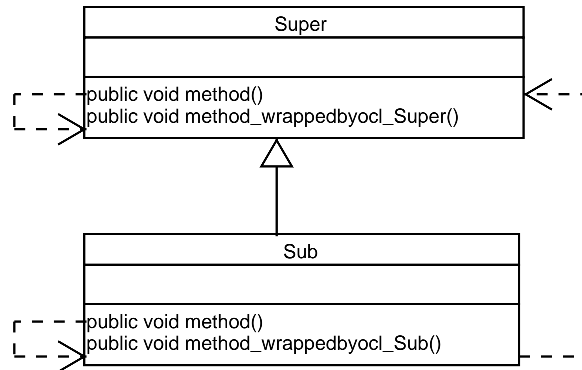
1. Method `Sub.method()` is called somewhere in the user program. This is a wrapper method replacing the original method, which is named `method_wrappedbyocl()` now.

2. `Sub.method()` does some OCL specific things, before it executes the statement `method_wrappedbyocl()`. This calls the original method `Sub.method_wrappedbyocl()` as it is supposed to be.

3. `Sub.method_wrappedbyocl()` contains the `super.method();` statement, therefore calls `Super.method()`.

4. `Super.method()` is a wrapper method replacing the original method, which is called `Super.method_wrappedbyocl()` now. It does some OCL specific thing, before it executes the statement `method_wrappedbyocl()`. But this statement does not call `Super.method_wrappedbyocl()` as it is supposed to be, but `Sub.method_wrappedbyocl()`, which finally causes the infinite loop.

The principal solution approach is simple: method `Super.method()` should force `Super.method_wrappedbyocl()` to be executed, although this method was overridden in class `Sub`. Java language does not provide a way, to call a method, which was overridden. Therefore we do a small trick:



Wrapped methods get the class name appended. Thus, a wrapper method can call the wrapped method of its own class.

3.3.5 Cleaning the Code

Reversable modification means, that the instrumented code can be cleaned from any modifications without leaving any traces. This section explains, how this requirement is met.

The user code is modified in two different ways only:

1. Renaming the wrapped methods/constructors.

2. Adding new object features, e.g. wrapper methods, methods for checking invariants and observing attributes.

For each method to be wrapped the suffix `_wrappedbyocl` is appended to the name. This transformation is done on the unparsed method header, so all typographical extras (line breaks, comments etc.) are preserved. This transformation is easily reversed, when the code has to be cleaned. For constructors, this works similarly with appending the dummy parameter to the parameter list.

Removing generated class features relies on the fact, that all generated features get a special tag as shown below.

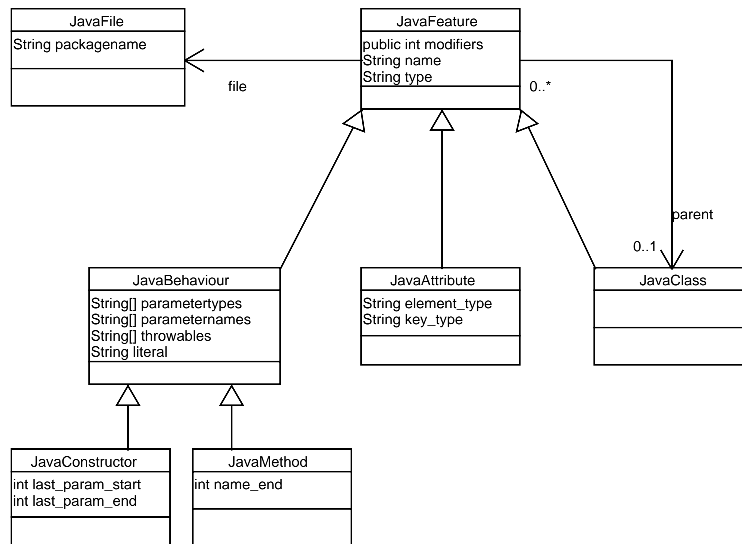


Figure 3.1: Design of the Java Parser used by the OCL Instrumentation

```
/**
   @author ocl_injector
*/
void checkOclInvariants();
```

When cleaning the code, all object features carrying such an `@author` tag are removed. This is quite simple and functional.

3.3.6 Design of the Java Parser

Previous sections stated, that a very simple parser is sufficient for implementing wrapper methods. This is proven in this section by giving an overview of the parser's design. In fact, it is as simple as a parser used for syntax highlighting and class browsers in a java IDE.

First, the parser is actually a manipulator. The java file is simultaneously read, parsed, modified on-the-fly and written to an output file. For a class diagram of the parse tree produced see figure 3.1.

The parser analyses things which are relevant for the parse tree only. Particularly method bodies and attribute initializers are ignored. These skipped parts may not even compile. As long as the parenthesis balance is held, the java parser will process them correctly.

3.3.7 Comparison to Jass

Jass ([JASS]) is a precompiler for checking assertions in java programs. It translates jass files into java. Jass files are valid java source code with assertions specified in comments. The generated java file contains additional code checking these assertions. Thus, Jass performs something similar to the code instrumentation presented in the sections above.

However, Jass directly inserts generated code into user code as described in section 3.3.1. Thus the problems found there should occur in Jass too:

1. *The code to be executed after the method has to be inserted before any return statement.*
   This is done by Jass. Thus, it requires a full java parser (JavaCC here).

2. *The return expression has to be computed in advance.*
   This is also done by Jass.

3. *There may be name conflicts between the original and the generated code.*
   Jass just defines a number of names (e.g. `jassResult`), which cannot be used in the user code.

4. *For methods with return type void it must be decided, whether the end of the method is a reachable point of code.*
   This is a known problem of Jass. It will simply fail in such cases. There is a work around: enclose the method body into a `if(true){...}` statement.

Jass performs a complex modification of the java source code. This modification cannot be reversed as described in section 3.2.1. Thus, Jass must be run before compilation whenever the source code has changed. Additionally, the source code repository must hold jass files instead of java, which causes administration effort for existing projects.

Jass cannot use wrapper methods, since this would not allow loop invariants and check statements to be implemented. But without these features as in OCL, method wrappers are much better than direct insertion of code.

3.3.8 Comparison to iContract

iContract ([RK98]) is a precompiler for checking OCL constraints in java programs. It extracts constraints from javadoc comments and produces modified java source files checking these constraints. This is exactly the functionality, which the tool presented in this paper tries to provide.

Just like Jass it uses direct insertion of generated code into user code. Once again the problems found in section 3.3.1 are reviewed.

1. *The code to be executed after the method has to be inserted before any return statement.*
   This has to be done by iContract. However it fails here for most cases. According to the list of known problems *"iContract generates wrong code or crashes, if there is more than one return statement in a method."*. This matches with the experience of the author.

2. *The return expression has to be computed in advance.*
   This is also done by iContract.

3. *There may be name conflicts between the original and the generated code.*
   Also iContract forbids a number of names (e.g. `__return_value_holder_`), to be used in the user code, although this isn't documented.

4. *For methods with return type void it must be decided, whether the end of the method is a reachable point of code.*
   iContract doesn't do the control flow analysis needed to decide this question. This has been proven on example in appendix C.1.

Experience with iContract shows, that correctly modifying java source code isn't trivial at all. The current list of known problems suggests, that iContract isn't usable for a real-world task. Additionally, the problem of reachable ends of method bodies remains.

3.3.9 Comparison to Byte Code Instrumentation

This paper is focused on source code instrumentation. Another approach is to modify java byte code. This section discusses, whether it's possible and useful, to apply the concept of method wrappers to byte code instrumentation.

For the third time the problems found in section 3.3.1 are reviewed.

1. *The code to be executed after the method has to be inserted before any return statement.*
   This is also needed for byte code instrumentation, but it's much easier. The code just has to be scanned for return opcodes.

2. *The return expression has to be computed in advance.*
   This is not needed. Whenever a return opcode is executed, the return value is ready-to-use on the top of the execution stack.

3. *There may be name conflicts between the original and the generated code.*
   This cannot happen. Name conflicts with local variables are not possible, since variable names do not exist anymore in byte code. Name conflicts with non-local variables are not possible too, since they are handled by different opcodes.

4. *For methods with return type void it must be decided, whether the end of the method is a reachable point of code.*
   This is also no problem. The java compiler takes care for it - just scan for return opcodes. If the end of the method body isn't a reachable point of code, then there is also no return opcode. On the other hand, if the end of a method *is* reachable, there will also be a return opcode, no matter whether there was a return statement in the source. This is shown on example in appendix C.2, since the JVM Specification [LY97] wasn't that clear about it.

Thus, it makes no sense, to apply the approach of method wrappers to byte code instrumentation.

In contrary to the approach presented in this paper, byte code instrumentation has to be redone after every compilation of the source code. This may be done on the fly, when loading the classes into the JVM. For instance jContractor [KHB98] uses a class loader to instrument java code. This may cause problems, if the user code registers a class loader of it's own. This is solved by Handshake [DH98], which uses a proxy system library to intercept the JVM when opening class files. Thus, Handshake buys total transparency to the JVM with platform dependency.

3.3.10 Summary

Wrapping methods seems to be the best choice when instrumenting source code for checking constraints. Several problems encountered with the simple approach of direct insertion are solved.

Method wrappers cannot be used, if implementation constraints (assertions, loop invariants) are to be checked. Since OCL does not support implementation constraints, this does not affect the scope of this paper.

Method wrappers are not useful for instrumentation on byte code level. The biggest advantage of source instrumentation (if it is reversible) is, that it doesn't need to be redone on each recompilation.

3.4 Scope of Invariants

This section discusses the issue, when an constraint is required to be fulfilled. This is trivially for pre/post conditions, but for invariants it's not so easy.

[WK99] section 5.4.2 suggests to check invariants immediately after an object has changed[4]. This is not workable, even if runtime efficiency is ignored. Modifications on the model often produce intermediate states, which are not consistent according to the constraints.

When using databases the answer is simple: invariants must be valid outside of transactions. Since the java system does not provide transactions there are several strategies offered for various user requirements.

Invariants may be required to be fulfilled on:

- All methods. This may be too strict, since private methods may intentionally leave an object in an inconsistent state.
- Public methods (or any other access modifier). This may be not strict enough.
- Tagged methods. A special tag in the javadoc comment declares, that a method promises to leave the system in a consistent state. This tag is then part of the interface contract. This is the best solution, but requires additional effort spent by the developer.
- Explicit request. This is the way of choice, if the model is held in a database backend. Then the checking of invariants is simply done immediately before committing.

These strategies may be used in conjunction. Except of *All Methods* together with *Public Methods* and/or *Tagged Methods* all other combinations make sense for special user requirements.

There won't be a single solution for this problem. Many applications will require their own individual scope of invariants. The scope may even differ between several classes of invariants.

The current implementation supports scopes needed during the ongoing diploma thesis only. Up to now, all invariants share the same scope. Also, tagged methods are not yet supported.

[4] This is partially corrected in the errata [WK99e].

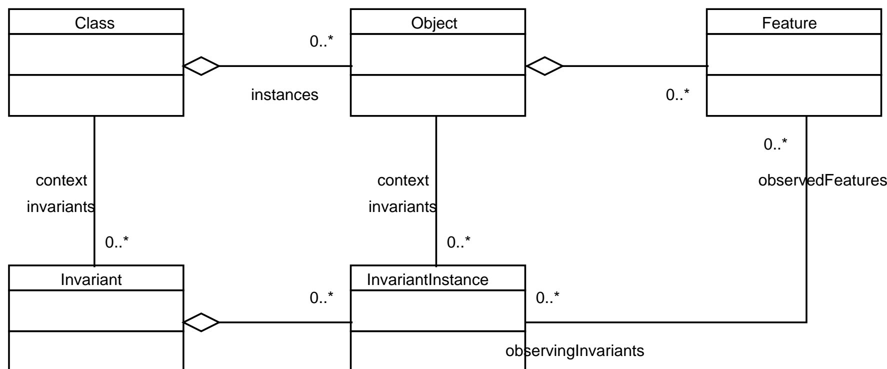


Figure 3.2: Design for Observing Invariants

3.5 Caching Results of Invariants

The previous section discussed, when we have to make sure, that all invariants are fulfilled. But even then it's not absolutely necessary to evaluate all invariants. The implementation developed along with this paper checks only those invariants, whose result may possibly have changed by recent changes of the model.

3.5.1 Design

Caching is realized with an observer design. Each invariant determines all object attributes it depends on and registers to these attributes as observer. Figure 3.2 shows the meta model of the principal design.

The classes in the UML chart have the following meaning:

Class		Example
Class	An arbitrary class of the user model	`class Person`
Invariant	An invariant in the context of a class	`context Person inv: age>=0`
Object	An instance of a class	Person Joe
Invariant-Instance	An invariant in the context of an object.	Has Joe a positive age?
Feature	A feature (attribute or query method) of an object.	Joe's age

For now, lets think of features as attributes only. How to deal with query methods is explained below.

The cycle of checking invariants contains two stages.

1. Evaluating invariants. When evaluating an invariant instance, this invariant instance registers to all object attributes used during evaluation as observer. This means, the attribute promises to notify the invariant instance, when the attributes value changes.

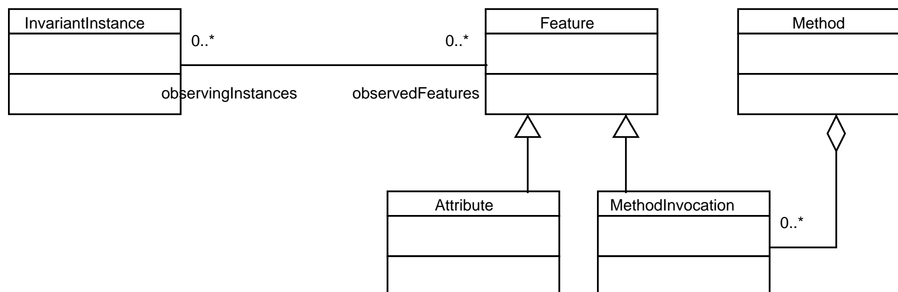


Figure 3.3: Design for Observing Methods

2. Running the model. When an attribute changed its value during execution of user code, it notifies all observing invariant instances. Then, the attribute unregisters all observers, so they must register again on the next evaluation stage. See section 3.5.2 how changed attributes are detected.

This design can be extended to query[5] methods. If the query does not have parameters, it's exactly like attributes. Things get a bit more complex, if the queries are parameterized. See figure 3.3.

The point is, that not methods but method invocations are features observed by invariants. A method invocation is a method together with a parameter sequence suitable to invoke this method.

Up to now, the implementation observes attributes only.

3.5.2 Implementation

Not all of these classes exist explicitly in the implementation. *Class* and *Object* are provided by the user model already. *Invariant* exists only as an additional method `checkOclInvariant_<name>` of its context class. *InvariantInstance* is an explicit class[6] of the instrumentation runtime library. Finally *Feature* is provided by the user model, but cannot be referred to as a single java object. (`java.lang.reflect.Field` is a field of a class, not of an object.) Whenever a feature has to be referred to, it is represented by it's observer collection object, which is sufficient for the needs of this implementation.

Changes of object features are detected with polling. For each feature a backup attribute is added to the class.

```
class Person
{
  int age;
  int age_oclbackup=age;
}
```

Additionally there is a utility method added comparing each attribute to it's backup. If there is a difference, the observers of the attribute are notified.

[5]Operations used in ocl expressions must not have side effects.
[6]called a bit confusingly `tudresden.ocl.injection.lib.Invariant`.

```
private void checkForChangedFeatures()
{
  if(age!=age_oclbackup)
  {
    age_oclbackup=age;
    // notify observers of age
  }
  // ... further attributes
}
```

This method is called immediately before and after each method of the class. If the attribute contains an object reference, the comparison tests object identity, not object equality. This means, the `!=` operator is used as for basic types and not the `equals()` method.

For collections the backup stores a hashcode of the collection to avoid the overhead of maintaining a complete backup collection. Since comparison to backups is done very often, hashcode computation is required to be lightweight. The following section discusses this in detail.

3.5.3 Detecting Collection Modification

Detecting changes of attributes is essential for caching of invariants as described above. For atomic attributes this is trivial: `age!=age_backup` does it all. For collection attributes it's not that easy. The backup of a collection shouldn't be a collection as well, since this would consume large amounts of memory. Also, comparison between two collection isn't that fast.

Thus, the backup for collection attributes stores a single integer value only. This value is computed from the collection. When the collection changes, also the value is expected to change. Such a value is commonly referred to as hash value.

However, these hash values are not required to be uniformly distributed. Also, the hash value is required to be constant for unmodified collections only, not generally for equal collections. This means: if an element is added to the collection and removed immediately afterwards, the collection is *not* required to have the same hash code as before. This relaxion of requirements will be used below. It has to be admitted, that the term "hash code" is used here mainly for historical reasons.

A first try to implement the hash functions follows the implementation of `hashCode` methods in the Java Collection API. These methods cannot be used directly, since they call method `hashCode` for each of their elements. This is not desired, since change detection covers object identity, not object value. To achieve the intended behavior, collection hash functions had to be rewritten with calls to `System.identityHashCode`. This has been implemented in class `HashExact`[7].

These hash functions are good at detecting changes. However, they are too slow. Each invocation involves an iteration over the whole collection. For object populations of a few hundert instances with many relations between them, this virtually causes the system to stop.

[7] All the source code is in `tudresden.injection.lib.Hash*.java`.

A quick but incomplete fix is achieved with the hash functions in `HashSize`. They simply return the size of the collection. This is very quick of course. But it does not detect changes which don't affect the collections size.

Finally, `HashModCount` performs a hash function which is a) nearly as fast as `HashSize`, but b) performs change detection even better than `HashExact`. It uses the fact, that standard collections already provide a change detection mechanism for implementing fail-fast iterators. Each collection maintains a modification counter, which is incremented whenever the collection is modified. Iterators create a copy of this counter on creation, and check this copy on every access. Unfortunately, the counter isn't publicly available. Thus, `HashModCount` has to access the private counter attribute via reflection. This is highly dependent on the internal details of collection classes. The implementation has been tested successfully on JDK 1.2.2. With other versions better do not expect it to work.

There is a work-around for this. The collection backup could be an iterator instead of an integer. To detect a modification, just access the iterator and wait for a `ConcurrentModificationException`. This has not been implemented yet, since it imposes several problems. Method `hasNext()` does not check for modifications. Thus, method `next()` has to be invoked. But this may also fail due to `NoSuchElementException`if there are no elements left. Thus, the iterator used for backup would have to be recreated, whenever there is no element left. Empty collections would require a special treatment.

Using the fail-fast mechanism obviously does not work for arrays. However, a fall back to `HashExact` or `HashSize` is easily provided. This could even be decided dynamicely depending on the size of the arrays.

The table below summarizes the different detection mechanisms:

	Exact	Size	ModCount	Iterator
Modification Detection	nearly perfect	insertion/ deletion only	perfect	perfect
Runtime Complexity	linear	constant/ very low	constant/ low	constant/ intermediate
Implemention	yes	yes	yes	-
Works with				
Arrays	yes	yes	fall back to other	fall back to other
Collections without Fail-Fast Iterators	yes	yes	runtime error	silent failure
Non-JDK Collections with Fail-Fast Iterators	yes	yes	runtime error	yes
JDK Standard Collections	yes	yes	yes	yes

Finally there is the question, which to choose. Probably one should start with *Exact* (the default). If this works, everything is fine. If the application slows down more, than one is willing to accept, try *ModCount* (invoked with option `--modcount-hash`). If this works, it's fine. Otherwise it will fail-fast throwing a `RuntimeException`. Then one may try *Size* (`--simple-hash`). Hopefully it doesn't miss too many modifications.

If this is not acceptable, one may implement the *Iterator* method. It is important, that all collections used in the application provide fail-fast iterators (JDK standard collections do). Otherwise modifications may be missed silently.

Chapter 4

Model Information

The OCL compiler needs model information for type checking. How this works is explained in [FF00] section 5.3.3.

One possible source of model information may be a UML model exported from a CASE tool. This is probably the most elegant way. But since most real-world projects don't have a (up to date) UML representation of their business model, this isn't feasible in practice.

Another source is the java code itself, accessed through the reflection API[1]. This is very convenient, since no additional model is needed. However, java reflection lacks some model properties which are important for type checking.

1. Element types of collections, particularly collections representing associations. From a C++ perspective, java lacks templates implementing parameterized container classes.
2. Qualifier types of maps, representing qualified associations.
3. The isQuery tag of operations. Note, that OCL expressions may use operations without side effects (queries) only.

This chapter presents a solution to the first two items above. The information needed is put into the source code. Section 4.1 explains, how this information is stored, while section 4.2 presents several approaches, how this information is generated.

The third item could be solved in a similar way, by putting an isQuery tag into the source code. However, this is not an urgent problem. Without an explicit solution the developer has to be careful to call java methods without side effects only in OCL expressions.

4.1 Representing Element Types

Element types and qualifier types are specified using special tags in javadoc comments. See the example below.

[1] see class `tudresden.ocl.check.types.ReflectionFacade`.

```
class Company
{
  /**
     All persons employed by this company.
     @element-type Person
  */
  Collection employees;
}
```

The `@element-type` tag takes an parameter specifying a java class or interface. Thus, it's similar to `@see` as defined in [GJS96] section 18.4.1. The `@element-type` tag is valid for attributes only, and there must be at most one such tag per javadoc comment. The tag is not restricted to attributes of type `java.util.Collection`, since future implementations could use other collection API's as well.

Analogously, the `@key-type` tag is introduced for association qualifiers.

```
class Bank
{
  /**
     Customers qualified by their account number.
     @element-type Person
     @key-type Integer
  */
  Map customers;
}
```

Note, that the reflection model is restricted to qualified associations with one qualifier only. [UML] allows multiple qualifiers, but there is no convenient representation for this in java.

Furthermore, UML specifies a qualified association which has not been qualified in the OCL expression to be a set, i.e. there must be no duplicates. For the java example above, this means, that the following invariant must hold:

```
context Bank inv:
  customers->size()=customers->asSet()->size()
```

Since this is not enforced by `java.util.Map` (only keys are guaranteed to be unique), the OCL library provides an appropriate runtime check.

Implementation. A really comfortable implementation would let the java compiler do the parsing, and provide the information through an extended reflection API. This would be similar to the `@deprecated` tag. However, this approach would require the java compiler, the JVM and the standard runtime library to be modified. Apart from the effort of making these modifications, most java developers probably have a profound aversion against using a dedicated java environment just for checking OCL constraints.

The implementation developed with this paper extends[2] the reflection facade by scanning the source code for these comments on demand. This implies, that

[2]Encapsulated in `tudresden.ocl.check.types.ReflectionExtender`.

the java source code is necessary for type checking OCL constraints in addition to the class files.

There is a crucial question left: Where do the tags come? Possible sources are:

- A UML model. The code generator of a CASE tool could generate these tags.
- Maintained by hand. It is good-practice of programming, to specify which kind of objects are supposed to be in a collection attribute. The tags just make this information available formally.
- Reverse Engineering. This is discussed in detail in section 4.2 below.

Collection attributes with type tags are verified on runtime by the instrumented code. Note, that this kind of type information is useful for reverse engineering a UML model from given java code.

4.2 Reverse Engineering

Section 4.1 explained, how to store additional type information of a java model in javadoc tags. This section discusses, how to create this information.

Actually, these type tags have to be created manually. None of the automated procedures is perfect, so these procedures are suitable for decision support only. This chapter tries to support the developer with an interactive tool for inserting `@element-type` and `@key-type` tags into the code. There are two main features of this tool:

1. Graphical User Interface: Clear presentation of missing type tags and comfortable editing facilities.
2. Decision Support: Giving hints to the developer. These hints are either derived statically (section 4.2.1 and 4.2.3) or gathered dynamically on runtime (section 4.2.2). There should be a special indication, if several hints suggest different types.

A prototype[3] of this tool according to the ideas presented in this section has been developed by Steffen Zschaler. The prototype currently features the graphical user interface and the runtime analysis of section 4.2.2.

4.2.1 Source Code Analysis

Information about element types may be derived from static properties of the class, such as parameter types of methods and other tags in javadoc comments.

The following example suggests some of these properties. The element type of `employees` is obviously `Person`, but this information is not yet available to the OCL compiler. The tool could derive an appropriate hint for the developer from each of the underlined features.

[3]run class `tudresden.ocl.injection.reverseeng.RevengGUI`.

```
/**
   All employed {@link Person persons} of this company.
   @see Person
*/
Collection employees;

boolean isEmployee(Person);
void addEmployee(Person);
void removeEmployee(Person);
```

Note, that the example above requires linguistic knowledge about plural and singular form of nouns (employee here). This gets far more difficult, if identifiers are not English.

4.2.2 Runtime Analysis

This section describes, how to trace element types of collections on runtime. This is useful, if there is no static type information available, as described in the previous section.

For each collection attribute, the object types encountered during a run of the program are collected and fed into the interactive tool. This requires the program to be executable. Additionally there must be extensive test cases available, otherwise only a subset of all possible element types will be encountered.

The interactive tool presents the set of object types for every collection attribute. Additionally, the tool highlights all types, for which there is no super type in this set. Formally, these are the local minima of the set respective to the generalization partial order. These local minima are good candidates for an element type, especially if there is only one minimum. Presenting minima simplifies the decision if there are many types encountered in the collection attribute.

Implementation. The instrumented code makes a static method `traceTypes`[4] to be executed whenever a collection attribute changes its contents. Class `TypeTracer` maintains a static data structure containing all element types and key types for all attributes, as well as the minima of these type sets. This information is continuesly written to a log file. The interactive tool can read this log file and display the information.

4.2.3 Byte Code Analysis

This section describes how type information can be extracted from java byte code. This technology and its implementation (called Superwomble) was developed by Daniel Jackson and Allison Waingold at the MIT. This section outlines the parts of their paper ([JW99]) related to type information together with experiences from practical experiments with the tool.

Superwomble is a powerful reverse engineering solution. It generates object graphs from nothing but java byte code. Object graphs are roughly speaking a subset of UML class diagrams. They feature classes with generalizationships

[4]Actually `tudresden.ocl.injection.lib.TypeTracer.traceTypes`.

and associations between them. The graph is finally fed into a tool named dot, which does a nice layout for the graph.

One of the tricky parts of this tool is the detection of element types for object containers, which is exactly what this whole chapter is about. How this works, is explained on the Company-Person example from section 4.1 (page 27).

Suppose `company` is a variable of type `Company` and `person` of type `Person`. A typical program around this example would probably contain a statement like this:

```
company.employees.add(person);
```

The operation `add` takes an argument of type `Object`, but is called with a variable of type `Person`. This is a good hint, that the element type of `employees` is `Person`.

The same works for objects returned from the container. The expression

```
person=(Person)(company.employees.iterator().next());
```

strongly suggests the element type `Person`.

Another highlight of Superwomble is, that container classes are detected even if they don't implement `java.util.Collection`. In fact the type is not cared at all. Instead there are some heuristics applied to decide, whether a class is an object container or not.

The tool was used on several parts of both the OCL toolkit and the net-linx code, and it produced good and reliable results.

Integration of Superwomble results into the interactive tool should be possible. The object graph is exported to a human readable text file. However, this task is outside the scope of this paper.

4.2.4 Comparison

This section provides a comparison between the three approaches presented in the sections above.

	Source Code	Byte Code (Superwomble)	Runtime
Source Code required	yes	no	yes
Required Code Quality	fairly parseable	fully compileable	up and running
Availability of Results	intermediate	good	good
Reliability of Results	good	very good	intermediate
Application Effort	low	low	high
Implementation Effort (starkly subjective)	low	high	very low
Availability	LGPL	Binary at no cost	LGPL

Runtime analysis requires the source code to be instrumented before. Only byte code analysis requires no source code. This argument is weakened by the fact, that the type information is to be inserted into source code anyway. However, byte code analysis may cover libraries, which aren't available in source code, but provide useful type information about other parts of the program.

Anyway, byte code analysis requires, that there is a fully compilable source code somewhere, even if it's not available to the user. Source code analysis even makes do with incorrect source code, as long as signature data is parsable (method headers etc.) and method bodies hold the bracket balance. Most demanding on source code quality is runtime analysis, which requires a running system, with complete test cases.

Source code analysis is most demanding on the "beauty" of implementation. To deliver results, it requires some kind of getter/setter methods for the container attributes. In contrary, byte code and runtime analysis even work for public container attributes manipulated from outside of the class.

For runtime analysis *reliability of results* depends heavily on completeness of test cases. If the test cases are insufficient, results may be wrong. Byte code analysis provides best reliability, it's more difficult for a poor quality code to fool the analysis.

Runtime analysis also requires most *application effort* for the user. The system must be actually run. Particularilly all requirements for runtime (libraries, database, configuration etc.) must be available.

The *implementation effort* is very subjective for this paper. Both source code and runtime analysis require parsing and instrumenting of java source code, which was already built for runtime verification of OCL constraints. Thus the implementation effort in this paper was low. Byte code analysis is something completely different.

Finally, *availability* is about whether it is allowed to use, review and adapt the implementation. According to [FSF00], the difference between *LGPL* and *Binary at no cost* is same as between free speech and free beer.

4.2.5 Summary

There have been three approaches presented for acquiring type information. Runtime analysis is implemented and fully integrated into the project. Source code analysis is not yet implemented, but this should be easy to add. Experiences with byte code analysis where drawn from the tool Superwomble ([JW99]). This is fully implemented but not integrated into the OCL toolkit, thus not ready to use.

Adding up the scores, byte code analysis is probably the best. However, most of the criteria listed above are some kind of orthogonal, so adding up scores might not be sufficient for a decision. Each application may emphasize different criteria, so a universal solution is not available.

All approaches have one thing in common: they are not perfect. Thus, they cannot be used directly in the type checker of the OCL compiler. The intermediate step of the `@element-type` tags is necessary to allow correcting intervention of a human user.

Chapter 5

Industrial Example

For practical experiments with an industrial strength project, the partner netlinx AG provided the source code of its emerging product nxCom.

The product is a three tier solution providing a business directory service. There is a core business logic module, a maintance interface implemented with Java Swing and a customer web interface using Java Server Pages.

The experiments where limited to the business logic module. It consists of 245 classes and 17131 lines of code. Persistence is realized using the Versant ODBMS and the Java Versant Interface (JVI) [VC]. The OCL toolkit does not work together with JVI due to a bug in the latter. (This bug has been fixed after submission of this paper.) A detailed analysis can be found in appendix C.3. For the test a special developer configuration was used, which employs a XML file for realizing persistence.

For experimentation 10 invariants have been developed and inserted into this module. It wouldn't make much sense to give an introduction into the business logic module, therefore the actual constraints have been transformed into analogous counterparts for the Person - Company model used in [WK99]:

- 5 invariants checking the consistency of bidirectional associations such as:

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

- 3 invariants where several attributes where tested for consistency to the object's current state in life cycle, such as:

  ```
  context Person inv:
    isMarried implies (wife->isEmpty xor husband->isEmpty)
  ```

- one invariant ensuring a minimum cardinality of an association:

  ```
  context Company inv has_employees:
    employees->size>0
  ```

- and finally one invariant checking the consistency of a redundant attribute, in this case the uppercase version of a string:

  ```
  context Company inv:
    uppername=name.toUpper
  ```

The impact of code instrumentation on the size of the module is shown in the table below. Lines of Code where determined using CCCC [TF]). Execution times where measured on a AMD Athlon 600 system.

	Original Code	Instrumented Code
Size of Source Code	1.1 MB	3.2 MB
Lines of Code	17131	43587
(Re-)Instrumentation	36 s	91 s
(Re-)Cleaning	23 s	73 s

Running the instrumented code revealed a number of constraint violations. Some of them have been reconstructed by hand, and turned out to be inconsistencies in the test data. Due to the amount it was not possible to correct these inconsistencies within this work.

Chapter 6

Summary

In this work, a java source code instrumentation tool has been developed. This tool allows to insert code generated by the OCL compiler into arbitrary user code. The instrumentation features an insertion scheme called method wrappers, which seems to be new in respect to the related work researched by the author. The insertion scheme provides a reversible instrumentation. This means, that the instrumented code can be modified without losing all changes on the next instrumentation.

A concept for caching results of invariants has been developed and partially implemented.

The OCL type checker was extended to gather additional information from javadoc tags. This makes the OCL compiler fully independent of a UML representation of the java code. A concept for generating such tags and inserting them into the java code has been developed. This concept has been partially implemented by Steffen Zschaler.

The existing OCL compiler has been maintained and slightly extended. These extentions provide additional flexibility needed for the components developed in this work. Coupling beetween existing and new components is kept low by restricting dependencies to java interfaces.

The tool has been experimented with using the java source code of an industrial application. For this application, a small set of typical constraints has been developed.

The OCL tool is still far from being complete. Some ideas for future work are discussed in the next chapter. However, the tool seems to be in a condition, where an early adopter could actually start using it for some real world task.

Chapter 7

Outlook

The results of this paper leave various directions for future work on the OCL toolkit. This chapter outlines the (in the author's opinion) most interesting ideas.

7.1 Inheriting Constraints

Inheriting constraints is completely neglected in the current version. This means, that a subclass does not inherit the constraints of its ancestors. At a glance this is an implementation problem only. But it is not that easy.

Simply promoting constraints of a class to all its descendants is not sufficient. This downward promotion implies a conjunction (logical AND) of inherited and own constraints. This is suitable for invariants and postconditions, but not for preconditions. Design by Contract (DbC) requires inheritance of *contracts*, not just constraints, thus preconditions cannot be strengthened. To make this sure, DbC suggests a disjunction (logical OR) of inherited and own preconditions.

This causes some uncomfortable effects. Consider a class representing a bank account. The account has an attribute `balance` and a method for drawing money. There is no debit allowed, and for some technical reason the amount to be drawn cannot exceed 10000 per transaction. This could be expressed like this: (Postconditions have been omitted for brevity).

```
context Account::drawMoney(amount:integer)
  pre: amount<=10000
  pre: balance-amount>=0
```

A subclass weakens the second precondition by allowing a debit of 1000.

```
context DebitableAccount::drawMoney(amount:integer)
  pre: balance-amount>=(-1000)
```

But what happened now? The effective precondition of DebitableAccount (*OR-ing* preconditions of both classes) is now just balance-amount>=(-1000). The upper limit for amount disappeared. This is perfectly compatible with DbC, but certainly not the intention of the developer.

A work-around is to repeat the upper limit for amount in the subclass. But this is not convenient, particularly when considering 10 such technical preconditions instead of just one.

A sensible solution should allow to inherit a precondition as such without repeating it in the subclass. A suitable solution could feature named preconditions and a "per name" disjunction of these:

```
context Account::drawMoney(amount:integer)
  pre max_amount:  amount<=10000
  pre min_balance: balance-amount>=0

context DebitableAccount::drawMoney(amount:integer)
  pre min_balance: balance-amount>=(-1000)
```

These considerations suggest, that there is still some investigation and development needed to make the OCL toolkit usable for supporting Design by Contract.

7.2 Integration with CASE Tools

7.2.1 Code Generation

The OCL toolkit works with arbitrary java source code. Thus, it is definitely compatible with java code generators of all CASE tools. Still, there are some issues to be mentioned.

Export of OCL. If the CASE tool is able to store OCL constraints, there should be some export function for them as well. A sophisticated export function could even embed the constraints into javadoc comments (section 3.2.2), providing a single source solution. This should be easy to implement for CASE tools with the source code available, otherwise the feasibility depends on API's of the CASE tool.

Export of Element Types. Even more important is the export of `@element-type` javadoc tags (section 4.1). These are needed for type checking of constraints. The feasibility of such an extension depends again on the availability of either the source code or suitable API's.

Superseding Name Adapters. Code generators of Argo/UML [AU], Rational Rose [RSC] and Together [TSC] require the employment of name adapters. This is, since an association end called `employers` does not result in a collection attribute of the same name, as one would expect. Instead, the attribute is called `myEmployers`, `theEmployers` or `lnkEmployers`, depending on the CASE tool. The necessary translation is encapsulated in name adapters. It would be nice to make code generators use the original name. This would remove the need for name adapters, making the OCL toolkit and its application a bit less complex.

7.2.2 Reverse Engineering

The OCL toolkit provides support for inserting and validating `@element-type` javadoc tags into arbitrary source code. This information could be used for reverse engineering from Java to UML.

This significantly changes the way of reverse engineering: Up to now, a CASE tool analyses the source code, and wherever there is some information missing (particularly the element type of container attributes) it makes some simple assumptions. In a second step, the user is expected to review the results and adjust them where necessary.

The OCL toolkit can be used to swap the order of steps. First, the missing information is inserted into the source code in form of `@element-type` tags. This information is then used by the CASE tool to straightly produce a correct UML model. The advantage is, that the additional information can be verified using the runtime checks provided by the OCL toolkit.

Of course, the `@element-type` tag does not provide all the needed information. For instance bidirectional associations cannot be determined, since they cannot be distinguished from two independent one-way associations. This could be approached by introducing some kind of `@reverse-direction` tag. Given the infrastructure already provided, such extensions should be easy to implement.

These considerations suggest, that the OCL toolkit together with a CASE tool could be extended into a complete round trip engineering solution - with special support for starting the round trip at the "java side" of the circle.

7.3 Others

Reduce Observed Attributes. Caching of invariants (section 3.5) requires observing object attributes for modifications. Up to now, all attributes found in the java source code are observed, regardless whether they are used in some OCL expression or not. This could be reduced to those attributes, which are actually referred to in any of the constraints. This would significantly improve runtime efficiency of the instrumented code. For an implementation one would probably intercept the OCL type checker to get the necessary information.

Observing Queries. Caching the results of invariants requires observing of both attributes and query methods. However, observing queries has not been implemented yet. The total approach to caching requires a significant memory overhead. Observing queries will make this even worse by introducing another step of indirection. This should not be attempted until this approach has shown to be functional for big projects with many and complex invariants.

Java Constraints. Constraints could be expressed in Java instead of OCL. This may be useful for developers, who are unfamiliar with UML and OCL. An advantage is, that such a tool would be dramatically smaller and faster. A serious drawback is the loss of some quite comfortable functionality of OCL, such as universal and existential quantifiers and `@pre` modifiers. Furthermore, caching of invariants for java constraints requires much more implementation effort.

Enhancing Flexibility. Simply put, this means more options and probably some kind of configuration file. A good example is iContract [RK98], which allows to enable constraints independently for packages, classes and methods using a configuration file. Additionally there could be a more flexible scope of invariants, as suggested in section 3.5.

Appendix A

Maintance of the OCL Compiler

This chapter describes all major changes to Frank Fingers OCL compiler. This includes bugfixes too, if they caused changes of internal or external interfaces. It it some kind of update for [FF00], listing everything changed since.

A.1 Reflection Facade and OCL Library

Many changes occurred both in the reflection model facade and in the OCL library. This section groups these changes.

A.1.1 Polymorphism of Operation Parameters

Both `ReflectionFacade.navigateParameterized` and `OclAnyImpl.getFeature` lacked polymorphism of operation parameters. This means, that a method is found only if actual parameter types match formal parameter types exactly. The correct behavior is, that actual parameter types may also be subtypes of formal parameter types. For a detailed description see [RW00] section 3.1.5.

The new implementation made `ReflectionAdapter.getClassForType` superfluous, so it was removed from the interface.

A.1.2 Mandatory Name Adapters

Previous versions of the OCL library provided a default functionality, if no name adapter had been set explicitly.

Now it is mandatory to set a name adapter. Otherwise a NullPointerException is thrown. The default functionality has been moved into a separate name adapter (`SimpleNameAdapter`) which is used in the reflection facade as well. The name adapter may also be set by the java property `tudresden.ocl.lib.nameadapter`.

`ArgoNameAdapter` has been generalized into `PrefixNameAdapter`, which takes an arbitrary name prefix (“`my`” for Argo/UML) as a constructor parameter. Using this adapter with “`lnk`” and “`the`” should work for Together/J and Rational Rose.

A.1.3 Qualified Associations

The OCL support was enhanced by adding a simplified form of qualified associations to the reflection facade and the OCL library. “Simplified” means, that there may be only one qualifier attribute.

Qualified associations are represented in java with `java.util.Map` by default, but this may be changed by implementing `ReflectionAdapter.isMap(Class)`.

A.1.4 Type Mapping from OCL to Java.

The mapping between OCL types and java types now supports the collections API introduced in JDK version 1.2. The changes concern `DefaultOclFactory.getOclRepresentationFor(Object)` and `DefaultReflectionAdapter.getClassForType`.

A special handling of `java.util.Vector` supports code generated by Argo/UML. The static configuration variable `Ocl.TAKE_VECTORS_AS_SET` causes vectors to be mapped to sets, instead of sequences.

The new mapping is listed below.

Java (java.util)	take vectors as set	OCL
List	-	Sequence
Vector	false true	Sequence Set
Set	-	Set
Map	-	Set (qualified)

Furthermore, arrays are now supported. They are mapped into sequences of the appropriate element type.

A.2 OCL Library

Some modifications occurred in the OCL library only.

A.2.1 Undefined Values

Previous versions of the OCL library implemented undefined values as singletons for each type.

This was given up. Now undefined values carry the reason for their creation with them. When the undefined value is tried to be evaluated, this reason is added to the exception message.

This made `Ocl.STRICT_CHECKING` superfluous, so it was removed. Wherever this property was used, the library now produces an undefined value, parameterized with the message of the exception formerly thrown.

Undefined values were introduced at some other operations of OCL objects:

1. Method `OclCollection.setToRange` now creates an undefined collection (instead of throwing an exception), if lower bound is greater than upper bound.

2. The methods `Ocl.to<OclType>(OclRoot)` now return an undefined value of the appropriate type, if argument is undefined.

A.2.2 Java Null Values

Previous versions handled a special null value for OCL strings. This null value behaved like an empty string for most cases (e.g. concatenation). However a null string and an empty string were not equal. This was removed. Now a null string does exactly behave like an empty string.

Previous versions returned an undefined OCL collection, if the java collection field was null. Now, null collections are treated exactly like empty collections.

This behavior is encapsulated into a new method `getOclRepresentationForNull(Class)` in `OclFactory`. This method is called in `OclAnyImpl.getFeature`.

A.3 Type Checker

Previous versions of the type checker missed support for classifier `OclAny` in class `DefaultTypeFactory`.

Method `getClassifier` in class `TypeFactory` has been removed and replaced by `TypeFactory.get`.

A.4 Java Code Generator

A.4.1 Code Fragments for @pre

Meaning of code fragments created for @pre expressions (TRANSFER, PREPARATION, POST) has been changed. The new behavior is easier and more flexible for different code instrumentation tools. For a description of the new behavior see section 3.1.

For the old behavior compare to [FF00] section 7.1.2. Below, there is a detailed comparison on the example used there.

The OCL expression is

```
context Person::getIncomeAfterTax(tax:Real):Real post:
  age = age@pre
```

The TRANSFER fragment generated is still the same:

```
OclInteger tudOclNode2;
```

However, the PREPARATION fragment has changed. Below, there is the old version. The new version just lacks the parts set in italics.

```
final OclAnyImpl tudOclNode0=Ocl.toOclAnyImpl(Ocl.getFor(this));
final OclReal tudOclOpPar0=Ocl.toOclReal(Ocl.getFor(tax));
final OclReal tudOclResult0=OclReal.UNDEFINED;
final OclInteger tudOclNode1=Ocl.toOclInteger(
    tudOclNode0.getFeature("age"));
final OclInteger tudOclNode2=Ocl.toOclInteger(
    tudOclNode0.getFeature("age"));
final OclBoolean tudOclNode3=tudOclNode1.isEqualTo(tudOclNode2);
this.tudOclNode2=tudOclNode2;
```

The POST fragment changed as well. Again, this is the old version, with the new version lacking the first line put in italics.

```
final OclInteger tudOclNode2=this.tudOclNode2;
final OclAnyImpl tudOclNode0=Ocl.toOclAnyImpl(Ocl.getFor(this));
final OclReal tudOclOpPar0=Ocl.toOclReal(Ocl.getFor(tax));
final OclReal tudOclResult0=Ocl.toOclReal(Ocl.getFor(result));
final OclInteger tudOclNode1=Ocl.toOclInteger(
    tudOclNode0.getFeature("age"));
final OclBoolean tudOclNode3=tudOclNode1.isEqualTo(tudOclNode2);
```

The old code fragments were intended to be inserted into the user code as follows:

```
public class Person
{
  // TRANSFER FRAGMENT

  public double getIncomeAfterTax(double tax)
  {
    getIncomeAfterTaxPRE(tax);
    // USER CODE
    getIncomeAfterTaxPOST(tax, result);
    return result;
  }

  private void getIncomeAfterTaxPRE(double tax)
  {
    // PREPARATION FRAGMENT
  }

  private void getIncomeAfterTaxPOST(double tax, double result)
  {
    // POST FRAGMENT
  }
}
```

With the new fragments, this still works. However, it's also possible to insert the TRANSFER fragment into the method, thus making it a local variable:

```
public class Person
{

  public double getIncomeAfterTax(double tax)
  {
    // TRANSFER FRAGEMENT
    {
      // PREPARATION FRAGMENT
    }
    // USER CODE
    {
```

```
      // POST FRAGEMENT
    }
    return result;
  }
}
```

The code instrumentation developed with this paper uses the second insertion scheme only.

A.4.2 Explicit Package Qualifiers.

The java code generator now optionally prepends the explicit package qualifier `tudresden.ocl.lib.` to OCL library classes. This makes an import statement superfluous, and therefore fulfills the corresponding requirement in section 3.2.1.

Appendix B

Usage of the OCL Tool

This chapter describes, how to use the instrumentation tool. Section B.1 contains an example quickly demonstrating the main features. Section B.2 provides a complete reference of the command line options.

B.1 Example

Step 1: First, get the following files: dresden-ocl-injector.jar, xerces.jar and royloy.jar. These files are available on our website at dresden-ocl.sourceforge.net or may be produced from the sources using the `makeJar` script. Put these files into one directory and change into that directory.

The file royloy.jar contains the example code. To unzip it type:

```
jar -xf royloy.jar
```

You may now have a look at the example code, its in tudresden/ocl/test/royloy/. The java code already contains OCL expressions in javadoc comments, for an easy start see `Person.java` and `Company.java`. Additional OCL is in the file `oclexpressions`.

Step 2: Now it's time to start the OCL tool. Type the following command on a single line. Note, that the wildcards require a Unix shell to be expanded.

```
java -jar dresden-ocl-injector.jar
   -r tudresden.ocl.test.royloy
   --modify tudresden/ocl/test/royloy/*.java
```

The example java code has now been modified. The modified code will have the same behavior as the original code, but will additionally check the OCL constraints embedded in the javadoc comments.

Step 3: To check this, compile the modified java code (again type everything on a single line):

```
javac
   -classpath dresden-ocl-injector.jar
   tudresden/ocl/test/royloy/*.java
```

and

Step 4: Run the test main function provided:

```
java
   -cp .:dresden-ocl-injector.jar
   tudresden.ocl.test.TestInjectionRoyloy
```

Enjoy the messages about violated OCL constraints running down the screen. Messages contain information about the violated constraint and the object involved.

Step 5: Clean the code from the modifications the OCL tool made.

```
java -jar dresden-ocl-injector.jar
   --clean --modify tudresden/ocl/test/royloy/*.java
```

The example code is now exactly the same as you downloaded it.

Note, that the code actually was cleaned. No backups involved. To check this, try the following:

1. Make a backup of one or more java files just before running the OCL tool (step 2).
2. After running the tool make some small modifications to the files, you made a backup for. For example just add a single System.out.println to a method body.
3. Proceed until step 5.
4. Compare the cleaned code to your backup.

The difference will be just the small modifications you made.

B.2 Reference

Synopsis is below.

```
java tudresden.ocl.injection.Main [options] file.java ..
```

Provide all java files to be modified.

Options recognized are below. Most options have a short and a long version. Long versions try to be self explanatory and should be used in scripts.

`-m --modify`

Enables modifying java files. If not provided, the modified java code is written to `file.java.injected`. This switch serves as a safety check, whether you *really* want to replace your source code with the output of the OCL tool.

`-c --clean`

Performs cleaning of source files instead of instrumentation. After cleaning, the source code should show no differences to the version before running the OCL tool.

`-f --constraint-file constraints.txt`

Specifies the text file containing the constraints. Usually not needed, since constraints may and should be placed into javadoc comments. See section 3.2.2.

`-r --reflection-model modelpackage`

Specifies the java packages containing the model covered. If there is a constraint `context Person`, the type checker will look for class Person in all packages given by this option. This is some kind of "`import` for OCL". For multiple packages use multiple options.

`-n --name-adapter [none|argo]`

Specifies the name adapter. Default is `none`. If you don't use code generated by Argo/UML, the default is sufficient. Further information on name adapters is available at [FF00] section 3.5.1 and 3.5.6.

```
-is --invariant-scope
      [all|private|protected|package|public|explicit]
```

Specifies the the scope of invariants used. See section 3.4 for a detailed explanation. Access modifiers select methods having the same or a more public access modifier. Thus, `all` and `private` are equivalent. Default is `all`.

`-vm --violation-macro macro`

Specifies what to do, if a constraint fails. This string is inserted verbatim into the code. The OCL tool appends a pair of parents enclosing a suitable message string. Good candidates are `System.out.println` (the default) or `throw new RuntimeException`. Note, that the latter must be enclosed in `""` on Unix shells.

The exception thrown must not be a *checked exception* as specified by [GJS96] section 11.2. Otherwise the modified code will not compile!

`-tt --trace-types`

Performs type tracing of collection elements. See section 4.2.2 for details. The information gathered is written to a log file specified by property `tudresden.ocl.injection.lib.TypeTracer.log`.

Note, that the log file must be specified when running the user program, not the OCL tool! For example use `-Dtudresden.ocl.injection.lib.TypeTracer.log=example.ocltypetrace`. If not specified, the type information is written to standard out.

`--insert-immediately`

Triggers immediate insertion of wrapper methods. Otherwise wrapper methods are inserted at the end of each class. Immediate insertion allows easier tracing of the modifications. This option is mainly useful for debugging.

`--trace-checking`

Adds code logging each constraint checked on an object to standard out. Useful for debugging.

`--simple-hash`

Uses simple hash functions in `HashSize`, instead of the sophisticated hash functions in `HashExact`. Simple hash functions just return the size of the collection, so they will detect insertions/deletions only. Reduces CPU load for big models.

`--modcount-hash`

Uses hash functions in `HashModCount`, instead of the hash functions in `HashExact`. `HashModCount` functions are even better in modification detection than functions in `HashExact`. Also, they are almost as fast as `HashSize`. However, they heavily depend on internals of the java collections implementation. Use at your own risk. See section 3.5.3.

Appendix C

Code Examples

C.1 Unreachable Post Condition Code in iContract

This appendix shows, that iContract cannot handle methods with an unreachable end of method body as stated in section 3.3.8.

The example is derived from the example code that accompanies iContract. The following method provides such a situation: The `throw` statement in the last line prevents the execution path from ever reaching the end of the method body.

```
/**
 * @post age > 0
 */
public void setAge( int age )
{
  age_ = age;
  throw new RuntimeException();
}
```

Now the post condition has to be included into iContracts configuration, so that it actually gets checked.

```
iContract.doc.tutorial.Person.Employee.setAge(int) post
```

Now iContract performs the instrumentation of source code. When trying to compile the instrumented code, the compiler fails:

```
Employee.java:166: Statement not reached.
/*|*/ try {
```

To see why, let's have a look at the instrumented code.

```
/**
 * @post age > 0
 */
public void setAge(int age )
```

```
{
  /*|*/ //#*#-----------------------------------
  /*|*/ [ shorted by the author]
  /*|*/ //-----------------------------------#*#
  /*|*/ try {
  /*|*/ //---------------------------------------
  /*|*/
   age_ = age;
   throw new RuntimeException();
  /*|*/ //#*#-----------------------------------
  /*|*/ try { <-- line 166 is here
  /*|*/ if (!(age > 0))
  /*|*/   [ shortened by the author ]
  /*|*/     }
  /*|*/ catch ( RuntimeException ex ) {
  /*|*/ [ shortened by the author ] }
  /*|*/
  /*|*/ //---------------------------------#*#
  /*|*/
  /*|*/ //#*#---------------------------------
  /*|*/ } finally {
  /*|*/ [ shortened by the author ]
  /*|*/ }
  /*|*/ //---------------------------------#*#
  /*|*/
}
```

The original code fragment is followed by generated code checking the post condition. Since this code isn't reachable, the java compiler fails.

C.2 Return Opcodes in Java Byte Code

This appendix lists the example used to verify the statement of section 3.3.9, that byte code instrumentation does not need control flow analysis when inserting post method code.

The end of method body below isn't a reachable point of code.

```
void method1()
{
  throw new RuntimeException();
}
```

Thus, the end of the compiled method does not have a return opcode.

```
Method void method1()
0 new #10 <Class java.lang.RuntimeException>
3 dup
4 invokespecial #14 <Method java.lang.RuntimeException()>
7 athrow
```

The end of the second method *is* a reachable point of code.

```
void method2()
{
}
```

Accordingly, the method ends with a return opcode, even if the source code did not contain a return statement.

```
Method void method2()
0 return
```

Disassembling was done with `javap -c`.

C.3 The Problem with Versant Database

This appendix shows why the Versant ODBMS and the Dresden OCL Toolkit don't work together.

The problem is a bug in Java Versant Interface [VC] (JVI). JVI provides an enhancer, which instruments the user application on byte code level. The enhancer transforms any access to object fields into a call to a wrapper method provided by JVI.

The class used for invariant caching must be database persistent. Thus, it also has to be enhanced. This class contains the following piece of source code.

```
Field f=...;
if(f==null)
  throw new RuntimeException(...);
f.setAccessible(true);
HashSet observer=(HashSet)(f.get(o));
```

The last line is `Invariant.java:69`. This line throws the exception shown below.

```
java.lang.NoSuchMethodException:
_vj_getfield_com_net_linx_iyp_businessObject_-
InColumnAd_startOnline
at com.versant.trans.Wrappers.method_invoke
  (Wrappers.java:209)
at com.versant.trans.Wrappers.java_lang_reflect_Field_get_internal
  (Wrappers.java:344)
at com.versant.trans.Wrappers.java_lang_reflect_Field_get
  (Wrappers.java:369)
at tudresden.ocl.injection.lib.Invariant.addObserver
  (Invariant.java:69)
...
```

The code line in question contains a method call to `Field.get`. This method cannot throw a `NoSuchMethodException` under normal circumstances.

What happened? The enhancer transformed the call to `Field.get` into a call to a wrapper method provided by JVI. This method probably does some database stuff, and then tries to access the field requested. Therefore it

tries to invoke another wrapper method `_vj_getfield_com_net_linx_iyp_-businessObject_InColumnAd_startOnline` on the object. The class of the object is `InColumnAd`, the field requested is `startOnline`. However, this method does not exist, since the field is inherited from its super class.

This bug prevents any code from accessing object fields via reflection, if the objects class inherits this field from a super class. This affects the runtime libraries of both the OCL code generator and the code instrumentation.

A possible work-around is to wrap all field accesses with corresponding methods. These methods would access their field without reflection, thus would not be affected by the bug. These methods could be created automatically by the OCL code instrumentation. This would require a moderate implementation effort. However, the design of the OCL toolkit would suffer a much tighter dependency between the OCL library and the code instrumentation. The author does not plan to implement this work-around.

On October 13, 2000 Versant support announced a bugfix *"available within the next few weeks."* After submission of this paper, on February 27, 2001 this bugfix was published as patch bundle 3 for JVI 2.4.0 on Linux.

List of Figures

3.1 Design of the Java Parser used by the OCL Instrumentation . . . 18
3.2 Design for Observing Invariants 22
3.3 Design for Observing Methods 23

Bibliography

[AU] Argo/UML. http://argouml.tigris.org/.

[BS] BoldSoft. ModelRun. http://www.boldsoft.com/products/modelrun/index.html.

[CI] Cybernetic Intelligence GmbH. OCL Compiler. http://www. cybernetic.org/prodocl.htm.

[CMA] Glen McCluskey & Associates LLC. Java Test Coverage and Instrumentation Toolkits. http://www.glenmccl.com/instr/index.htm.

[DB99] Detlef Bertetzko. Parallelität und Vererbung beim "Programmieren mit Vertrag" - Weiterentwicklung von JaWA (in german). Master Thesis. Universität Oldenburg, 1999.

[DH98] Andrew Duncan, Urs Hölzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, Computer Science Department, University of California, Santa Barbara, December 1998. http://www.cs.ucsb.edu/oocsb/papers/handshake98.html.

[ET] Elixir Technology Pte Ltd. http://www.elixirtech.com/index.html.

[FF00] Frank Finger. Design and Implementation of a Modular OCL Compiler. Diplomarbeit. TU-Dresden, 2000. http://dresden-ocl.sourceforge.net/.

[FSF00] Free Software Foundation. What is Free Software? 1996-2000. http://www.gnu.org/philosophy/free-sw.html.

[GJS96] James Gosling, Bill Joy, Guy Steele. The Java Language Specification. Edition 1.0. Addison-Wesley, August 1996. http://java.sun.com/docs/books/jls/html/index.html.

[JASS] The Jass Page. http://semantik.informatik.uni-oldenburg.de/~jass/.

[JW99] Daniel Jackson, Allison Waingold. Lightweight Extraction of Object Models from Bytecode. Proc. International Conference on Software Engineering. May 1999. http://sdg.lcs.mit.edu/womble/.

[KHB98] Murat Karaorman, Urs Hölzle, John 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. http://www.cs.ucsb.edu/oocsb/papers/TRCS98-31.html.

[KSR00] Holger Knublauch, Martin Sedlmayr, Thomas Rose. Design Patterns for the Implementation of Constraints on JavaBeans. http://www.faw.uni-ulm.de/kbeans/.

[LY97] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification, Second Edition. Addison-Wesley, 1997. http://java.sun.com/docs/books/vmspec/.

[MMS] Man Machine Systems. JMSAssert. http://www.mmsindia.com/JMSAssert.html.

[OCL] Object Constraint Language Specification. Chapter 7 in [UML].

[OI] Object Insight, Inc. JVISION. http://www.object-insight.com/html/product_info.html.

[PSM98] Jeffery E. Payne, Michael A. Schatz, Matthew N. Schmid. Finding bugs early. Dr. Dobb's Journal, January 1998. http://www.ddj.com/articles/1998/9801/9801d/9801d.htm.

[RG00] Mark Richters, Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000). Springer, Berlin, LNCS, 2000. http://www.db.informatik.uni-bremen.de/projects/USE/.

[RK98] Reto Kramer. iContract - The Java- Design by Contract- Tool. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[RSC] Rational Software Corporation. Rational Rose. http://www.rational.com/products/rose/index.jsp.

[RW00] Ralf Wiebicke. XML Query Languages for Repositories Based on XML Documents. Großer Beleg. TU-Dresden, 2000. http://dresden-ocl.sourceforge.net/.

[SM93] Steve Maguire. Writing Solid Code. Microsoft Press, 1993.

[SM93g] Steve Maguire. Nie wieder Bugs! Die Kunst der fehlerfreien C-Programmierung. Microsoft Press Deutschland. German translation of [SM93].

[TF] Tim Littlefair. CCCC - C and C++ Code Counter. http://cccc.sourceforge.net/.

[TP98] Todd Plessel. Design By Contract: A Missing Link In The Quest For Quality Software. Lockheed Martin / US EPA, August 1998. http://www.elj.com/eiffel/dbc/.

[TSC] TogetherSoft Corporation. Together. http://www.togethersoft.com/.

[UML] OMG Unified Modeling Language Specification, Version 1.3, June 1999.

[VC] Versant Corporation. J/VERSANT Interface Release 2.4.0. http://www.versant.com/us/products/vds/index.html.

[WK99] Jos Warmer, Anneke Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1999.

[WK99e] Errata for [WK99]: http://www.klasse.nl/ocl-boek/errata.htm.