Bachelor's Thesis

Implementing a Debugger for Dresden OCL

submitted by

Lars Schütze

born 08.02.1989 in Dresden

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dipl.-Medieninf. Claas Wilke
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted March 26, 2013

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE BACHELOR-ARBEIT

Name des Studenten:	**Lars Schütze**
Immatrikulations-Nr.:	**3569963**
Studiengang:	**Bachelor Informatik**

Thema: **Entwicklung eines OCL-Debuggers für Dresden OCL**

Zielstellung:

Die Object Constraint Language (OCL) wurde als Spracherweiterung für die Unified Modeling Language (UML) entwickelt, um UML-Modelle und Metamodelle durch Constraints und Navigationssemantik detaillierter zu spezifizieren und einzuschränken. Heute wird OCL als Constraint- und Query-Sprache auf zahlreichen Modellierungs- und Metamodellierungssprachen eingesetzt. Ein Tool zur Parser- und Auswertungsunterstützung von OCL-Constraints auf zahlreichen Metamodellen und Modellen wird im Rahmen des Forschungsprojektes Dresden OCL entwickelt.

Zur Auswertung von OCL-Ausdrücken auf Modellen existieren dabei im Wesentlichen zwei Gruppen von Ansätzen: interpretative und generative Ansätze. Während generative Ansätze OCL- Ausdrücken vor der Auswertung in eine spezifische Implementierungs- oder Query-Sprache übersetzen, werten interpretative Ansätze OCL- Ausdrücken während einer Traversierung auf deren Syntax-Bäumen aus. Ergebnisse von Ausdrücken-Auswertungen sind in beiden Ansätzen meist Boolesche oder Numerische Werte oder Mengen dieser Werte. Die Nachvollziehbarkeit der Ergebnis-Ableitung ist dabei oft nur schwer möglich, da OCL-Ausdrücke komplexe Integritätsbedingungen beschreiben können, an denen zahlreiche Modell-Elemente beteiligt sind.

Fortsetzung Rückseite

Betreuer: Dipl.-Medieninf. Claas Wilke

Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Beginn am:	02.01.2013
Einzureichen am:	26.03.2013

[illegible]

Dresden, 12.12.2012

Prof.Dr.rer.nat.habil. U. Aßmann
verantwortl. Hochschullehrer

Fortsetzung Zielstellung:

Im Rahmen dieser Arbeit soll deshalb ein Debugging-Werkzeug für Dresden OCL entwickelt werden, dass den Nutzer bei der Interpretation von OCL-Constraints unterstützt. Der Debugger soll dabei mit dem existierenden, EMFText-basierten OCL-Editor von Dresden OCL eng verbunden sein und auf das existierende Debugging Framework von Eclipse aufsetzen.

Im Detail sind folgende Teilaufgaben zu bearbeiten:

- Einarbeitung in die Architektur von Dresden OCL, EMFText und das Eclipse Debugging Framework
- Erarbeitung eines Mappings (von Zeilennummern und Sprungmarken) zwischen dem Abstrakten Syntaxbaums (AST) des EMFText-basierten OCL-Editors und dem Abstrakten Syntaxmodells (ASM) von Dresden OCL
- Entwicklung einer Integration/Erweiterung des existierende OCL-Interpreters für das Eclipse Debugging Framework
- Test und Dokumentation des entwickelten OCL-Debuggers

Der Schwerpunkt dieser Bachelor-Arbeit liegt auf der Entwicklung und Implementation des OCL-Debugging-Werkzeugs. Die schriftliche Arbeit kann dementsprechend kürzer gehalten werden.

Contents

1 Introduction

"Anything that can go wrong will go wrong"

(Murphy's Law)

The *Object Constraint Language (OCL)* is a declarative language used to describe constraints on models or model extensions without side effects [OMG12, p. 21], e.g. without altering the state of the model. Developed in 1995 at *IBM* as a business modeling language [Hußoo], it became adopted by the *Object Management Group (OMG)* as an extension of the *Unified Modeling Language (UML)*. Today, it is part of UML itself [OA99].

"A debugger is a tool to (stepwise) execute and observe a program and its data"[1] [FH11, p. 223]; and debugging is the process of finding errors in a program. When the complexity of the program grows, quality assurance activities (e.g. pair programming, audits) come up against their limits. They cannot prevent errors but improve the overall code quality.

There are many ways of debugging. By definition, to log and print the flow of the program (also called tracing) or to print values in between are called debugging. But debugging does not only help to find errors. It also helps to understand complex scenarios and code. However, *Debugging* is used with different meanings in the context of model-based development. The Oxford Dictionary defines debugging as "[to] identify and remove errors from (computer hardware or software)" [Pre10]. In [SSJ$^+$03] counter-example generation is understood as debugging. Gogolla et al. understands debugging as the generation of trace outputs visualizing interim results [BGHK12]. "Debugging [...] support[s] [...] understanding the nature of bugs and typically offers functions such as running an expression step by step, conditional breakpoints [...], tracking and changing the values of variables" [COD10].

In this thesis debugging is understood as a graphical, disruptable (e.g. to suspend and resume) interpretation of (parts of) OCL expressions on model instances, similar to the graphical, stepwise debugging of Java programs in state-of-the-art IDEs such as Eclipse. Hence, the impact of each step during interpretation can be visualized.

Finding errors in a complex environment is not an easy task. To track the error down debugging is the best method in the field of OCL, since it is no general purpose language. As a *Domain Specific Language (DSL)*, OCL does not provide the ability to output to the console or any other standard output (e.g. `stdout` or `stderr` in C). Furthermore, debugging has been one of the most-wanted features regarding an `IDE4OCL` among other 20 identified features [COD10].

There are many tools around OCL [URL13c, URL13a, URL13b]. However, to the best of my knowledge there is no graphical debugging support for OCL, yet. This work addresses this issue and realizes a graphical debugger for *Dresden OCL* using the *Eclipse Debug Platform*, i.e. integrating a debugger for OCL into the Eclipse Debug Platform.

Thus, the major task of this work is to develop a graphical debugger integrated into *Dresden OCL* and the *Eclipse Integrated Development Environment (IDE)*. First, *Dresden OCL* is presented, then the *Eclipse Debug Platform* is introduced. Based on these tools, the requirements

[1]The original text is in German. Translation was done by the author.

for an OCL debugger are analyzed. Afterwards, the architecture is discussed and evaluated. At last, the graphical user interface is presented and the work is summarized.

The remainder of this thesis is organized as follows:

- Chapter 2 introduces *Dresden OCL*, its history and detailed information about important key features.
- In Chapter 3 the *Eclipse Debug Platform* is presented and explained.
- Then, the requirements for the debugger are derived and compared to other existing work in Chapter 4.
- Afterwards, Chapter 5 presents the architecture and design decisions. It is shown how the implementation is tested. Chapter 6 shows and explains the graphical user interface.
- In Chapter 7 the work is summarized and future work is discussed. Finally, in Chapter 8 a conclusion is drawn.

Some typographical rules which are used all over this work to remember:

- *Italics* are used for keywords or scientific terms,
- `Typewriter fonts` are used for programming constructs or programming languages, such as Java, Scala or OCL.

2 The Dresden OCL Toolkit

Dresden OCL is a set of tools to write and parse OCL constraints and to evaluate OCL constraints on various models. Furthermore, it provides code generation support. Dresden OCL has been developed at the Technische Universität Dresden since 1999. To this day, various versions have been released. The different releases and the corresponding UML versions are illustrated in Figure 2.1.

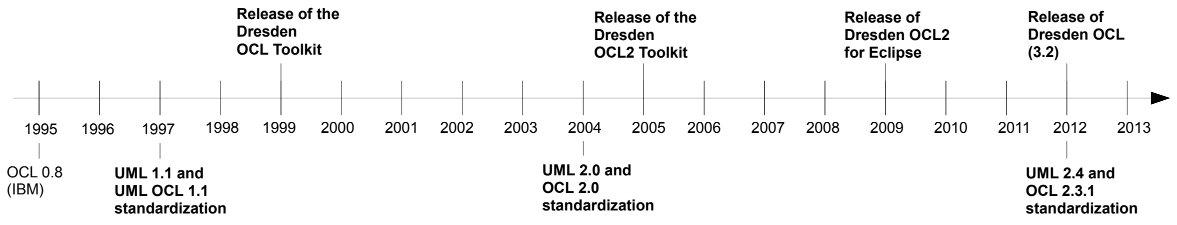

Release of the Dresden OCL Toolkit
Release of the Dresden OCL2 Toolkit
Release of Dresden OCL2 for Eclipse
Release of Dresden OCL (3.2)
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
OCL 0.8 (IBM)
UML 1.1 and UML OCL 1.1 standardization
UML 2.0 and OCL 2.0 standardization
UML 2.4 and OCL 2.3.1 standardization

Figure 2.1: Releases of Dresden OCL, UML and OCL

2.1 The Dresden OCL Toolkit

The first work on OCL at the Technische Universität Dresden was done in 1998, as Alexander Schmidt researched a mapping from OCL to the *Structured Query Language (SQL)* [Sch98]. Frank Finger realized an implementation of the OCL Standard Library and the possibility to load UML models and OCL constraints as well as a Java code generator [Fin99, Fin00]. Afterwards, Ralf Wiebicke developed the instrumentation of the generated Java code [Wie00]. The work of Alexander Schmidt, Frank Finger and Ralf Wiebicke resulted in a release called the *Dresden OCL Toolkit (DOT)*.

2.2 The Dresden OCL2 Toolkit

In 2003 Stefan Ocke adopted the DOT to the *Netbeans Metadata Repository* [Ock03]. This release was called the *Dresden OCL2 Toolkit (DOT2)*. DOT2 provided the first OCL to SQL code generation [Hei05, Hei06]. Ronny Brandt ported the Java code generation and instrumentation to this new release [Bra06].

2.3 Dresden OCL2 for Eclipse

In 2007 Matthias Bräuer realized the *pivot model*, which makes the toolkit independent from specific models or meta-model repositories [Brä07b]. Based on this work Nils Thieme developed an OCL parser [Thi08] for loading UML models and parsing OCL constraints and Ronny Brandt implemented a new OCL interpreter [Bra07a] for interpreting OCL constraints. This new release was called *Dresden OCL2 for Eclipse (DOT4Eclipse)*. Claas Wilke developed a new Java code generator [Wil09] using AspectJ and finally Björn Freitag reengineered the SQL code generation [Fre11].

2.4 Dresden OCL

Today the toolkit is known as *Dresden OCL*. Based on Dot4Eclipse, Dresden OCL currently supports the parsing and interpretation of OCL 2.3.1, Java and SQL code generation as well as further features such as OCL metrics and the tracing of OCL interpretation [WTFS12, p. 46].

The Architecture of Dresden OCL

Dresden OCL is build as a set of Eclipse plug-ins. Figure 2.2 shows an excerpt of the relevant parts of the toplevel architecture. Dresden OCL is separated into four parts: the *API*, the *tools*, the *OCL* and the *variability layer*.

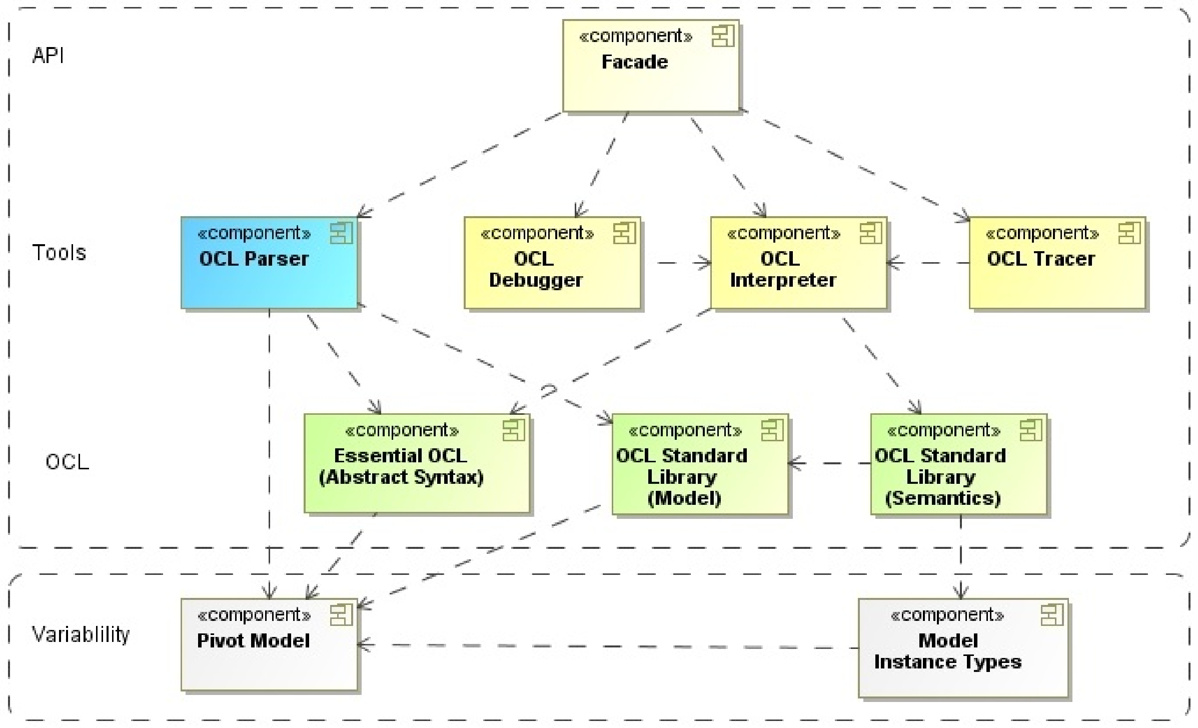

API
«component» Facade
Tools
«component» OCL Parser
«component» OCL Debugger
«component» OCL Interpreter
«component» OCL Tracer
OCL
«component» Essential OCL (Abstract Syntax)
«component» OCL Standard Library (Model)
«component» OCL Standard Library (Semantics)
Variablility
«component» Pivot Model
«component» Model Instance Types

Figure 2.2: Toplevel architecture of Dresden OCL (redrawn from [WTFS12])

The API layer declares the facade that hides all implementation details and provides an API to access all functionality Dresden OCL provides. Since Dresden OCL is provided as a single library this is what the end-user will use.

The tools layer provides each tool separated in its own Eclipse plug-in. Dresden OCL takes huge advantage of the Eclipse/OSGi implementation to separate each part from another.

The OCL layer consists of an implementation of OCL (i.e. its syntax and semantics, used as packages by the tools). It also contains the OCL Standard Library which provides a set of standard types (e.g. primitive types, enumerations and collections) as well as their semantics for OCL interpretation.

The variability layer consists of the pivot model and the model instance types. They are used to map arbitrary meta-models or model instances, respectively, on Dresden OCL. Those meta-models adopted are currently an *Eclipse Modeling Framework (EMF) Ecore Meta-Model*, a *Java Meta-Model*, a *UML2 Meta-Model* and a *XSD Schema Meta-Model* and their model instances, respectively [WTW10, p. 3f].

The various tree representations of OCL in Dresden OCL

When interpreting an OCL file, a model and a model instance have to be loaded. The OCL file is checked against the model, e.g. whether all contexts reference to existing classifiers in the model. Then it is parsed and its static semantics is checked, e.g. whether all invariants result in a boolean type. At last the constraints get interpreted on elements being part of the model instance selected beforehand, e.g. constraints defined on a class Person are evaluated for all persons being part of a model instance. Figure 2.3 illustrates the tool chain.

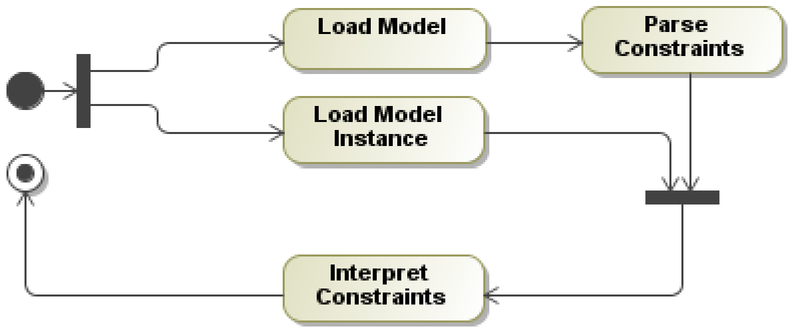


Figure 2.3: The tool chain of Dresden OCL

Each activity takes an input and produces an output. For Dresden OCL this process is realized as follows (cf. Figure 2.4): an EMFText-based parser parses the OCL file according to a defined *Concrete Syntax (CS)* specification of OCL. The output is a tree-based representation of the CS; called the *Abstract Syntax Tree (AST)* [ABB$^+$12]. This is the input to the OCL static semantics checker and type resolution component which maps the type, operation and property references in the AST to the pivot model elements of the constrained model and transforms the AST into another tree-based representation, called the *Abtract Syntax Model (ASM)* [BD08]. In the ASM the OCL constraints are represented as hierarchical expressions, which in contrast to the AST's representation can be traversed more easily and is thus, easier to interpret. Finally, the ASM is passed to the OCL Intepreter which interprets it on the model instance elements.

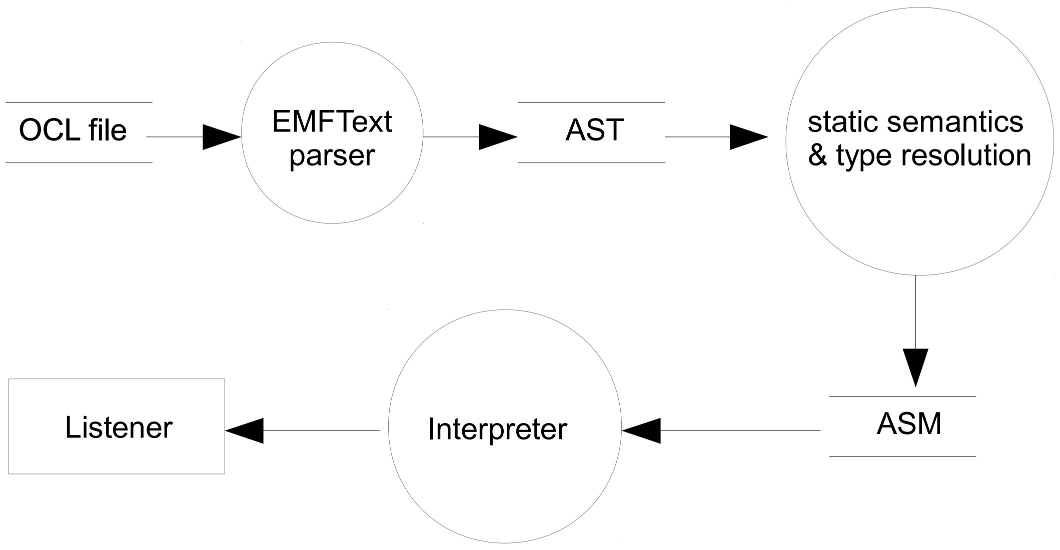


Figure 2.4: Data Flow Diagram (according to DeMarco [DeM79, p. 411]) while interpreting an OCL file

Concrete Syntax, Abstract Syntax Tree and Abstract Syntax Model by Example

To ease the understanding of the parsing and interpretation process in Dresden OCL, consider the following example: the example model is a single UML class `Person` with an attribute called `age`. Listing 2.1 shows the constraint defined for this model.

```
context Person
inv: (age + 1) > age
```

Listing 2.1: OCL invariant: age incremented by one is bigger than age

The concrete syntax can be written in the *Extended Backus-Naur Form (EBNF)*. Listing 2.2 shows an excerpt from the concrete syntax of OCL sufficient for the given example. It describes how the language is assembled. The complete concrete syntax of OCL is defined in the OCL specification [OMG12].

```
ExpressionInOclCS ::= CallExpCS
CallExpCS ::= FeatureCallExpCS
FeatureCallExpCS ::= OperationCallExpCS
OperationCallExpCS ::= OclExpressionCS[1] simpleNameCS OclExpressionCS
   [2]
PropertyCallExpCS ::= simpleNameCS
IntegerLiteralExpCS ::= <Integer Lexical Representation>
BracketExpCS ::= "(" OclExpressionCS ")"
```

Listing 2.2: Concrete Syntax for the simple invariant

Figure 2.5 shows the Abstract Syntax Tree. The Abstract Syntax Tree contains all neccessary information (e.g. it does not contain the parantheses anymore, that are defined with the *BracketExpCS* in the concrete syntax) to parse the tree. To ease the interpretation the AST is mapped on the ASM which is shown in Figure 2.6.

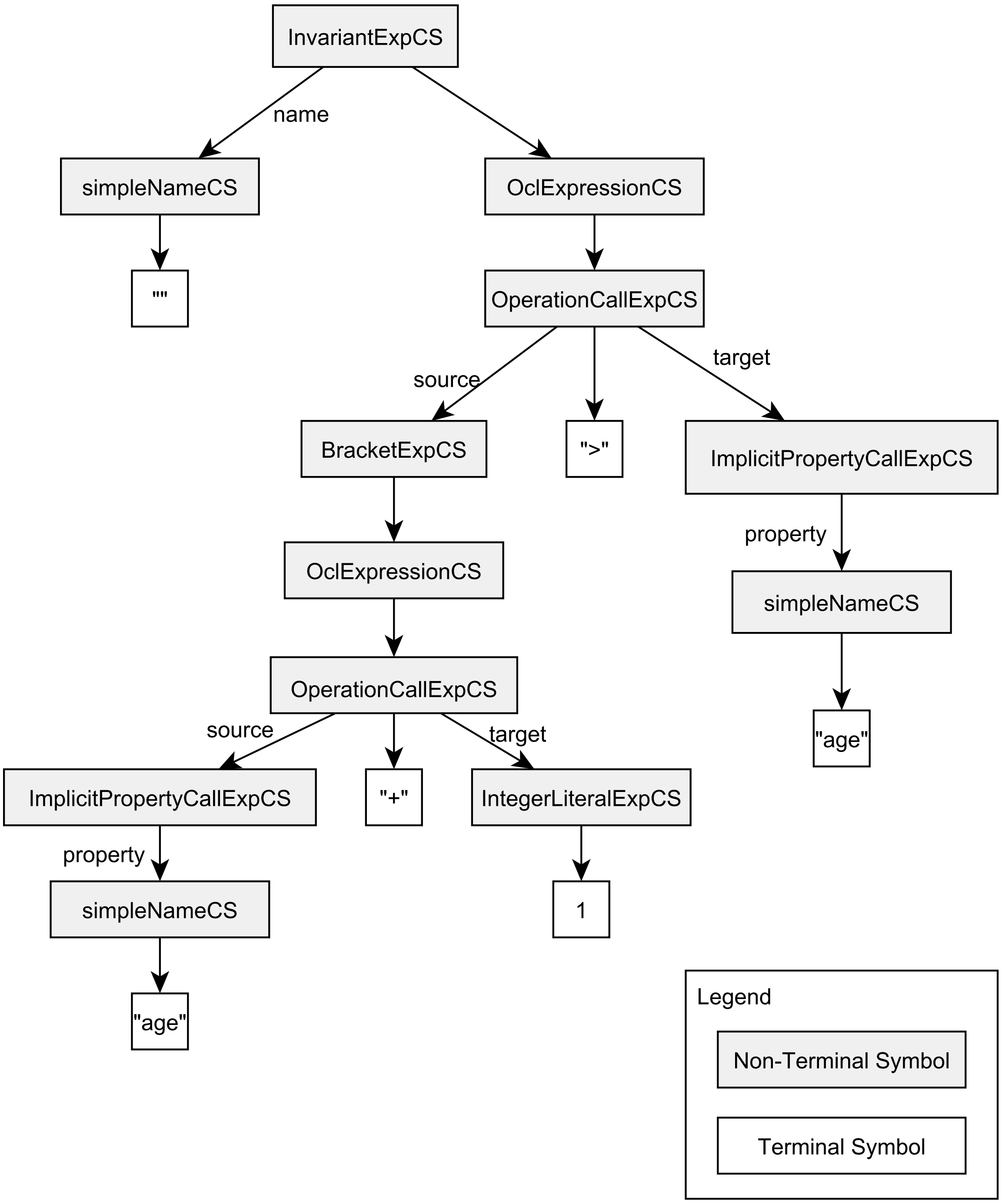


Figure 2.5: The Abstract Syntax Tree of a simple invariant

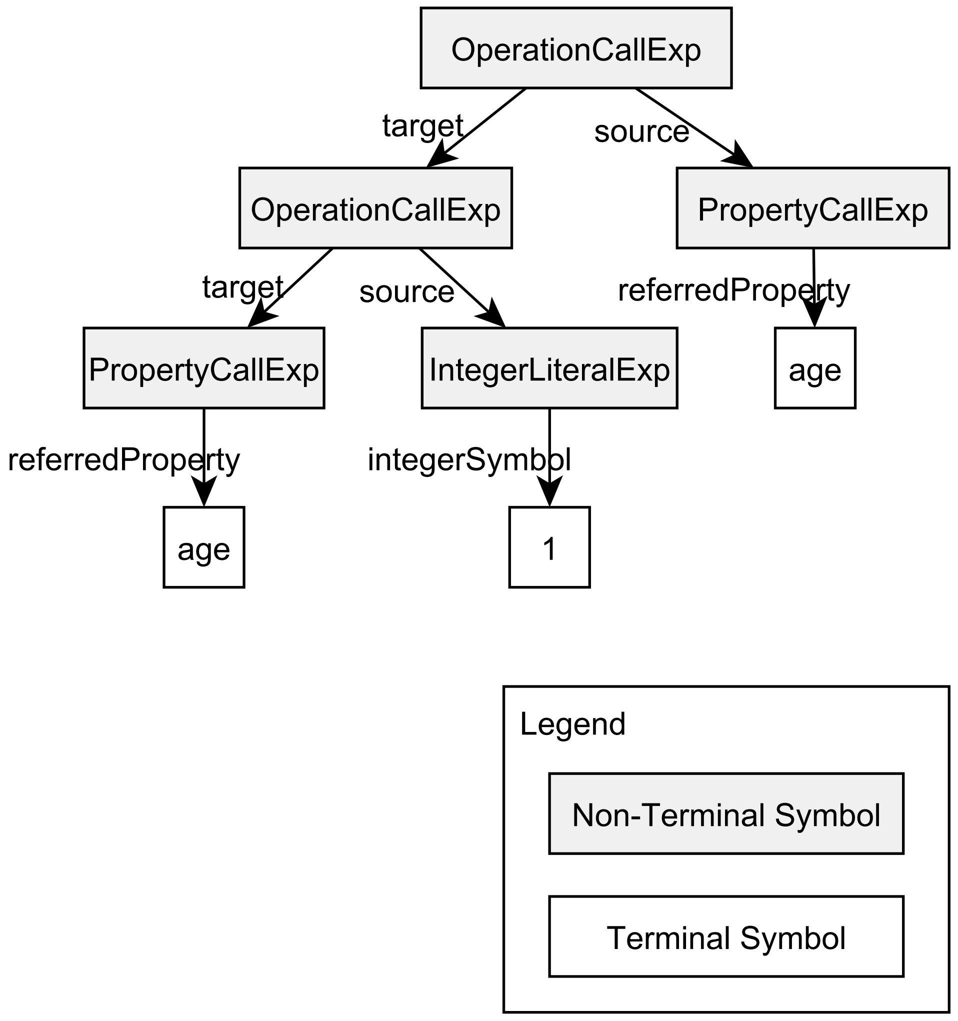


Figure 2.6: The Abstract Syntax Model of a simple invariant

3 The Eclipse Debugging Framework

The Eclipse Debugging Framework, also known as the *debug platform*, is part of the Eclipse Software Development Kit (SDK). It provides support for building and integrating debuggers into Eclipse. It supplies interfaces and classes known as the *debug model* [URL12a]. The debug model represents common artifacts in debuggable programs. This chapter will give an overview about the debug model and describes how the Eclipse user interface interacts with the debug model.

3.1 The Debug Model

The debug model provides interfaces that must be implemented by clients who want to provide debugging support. Figure 3.1 shows an overview of the debug model.

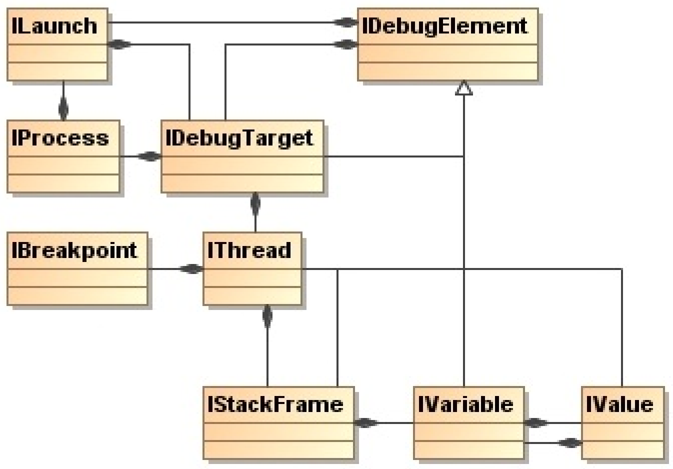


Figure 3.1: The Eclipse platform debug model architecture

The remainder of this section describes each part and its role.

ILaunch

The `ILaunch` results from launching either the debug session or a system process. Therefor it represents those process(es) launched and gives access to them.

IProcess

The `IProcess` represents a program in normal execution mode. It can be an external program being executed to run an external debugger.

IDebugElement

The `IDebugElement` represents an artifact in the program being debugged.

IDebugTarget

The `IDebugTarget` plays the central role in the debug model. It manages the other debug elements, e.g. it contains threads and give access to the process. The debug target represents either a debuggable process or a virtual machine. Thus, the abstraction provides the ability to represent either a debug target for a single debugging process, like `gdb` [URL12d] for the C language or to represent for example the *Java Virtual Machine*.

IThread

The `IThread` represents a thread in the program being debugged. It contains the stack frames and providing access to them.

IBreakpoint

The `IBreakpoint` represents a breakpoint in the program being debugged. Breakpoints are defined on lines of the program being debugged. They are "[...] capable of suspending the execution of a program at a specific location when a program is running in debug mode" [URL12b].

IStackFrame

The `IStackFrame` represents the execution stack of a thread being suspended. It contains the variables being visible at the current location of the program in execution. The stack can be used to visualize the execution path of a program. It contains access to the line, the characters of the start and end of the element in the resource (e.g. a `*.ocl` file containing the source code) to highlight the location in the source code. Figure 3.2 shows the visualization of an example stack frame in Eclipse.

IVariable

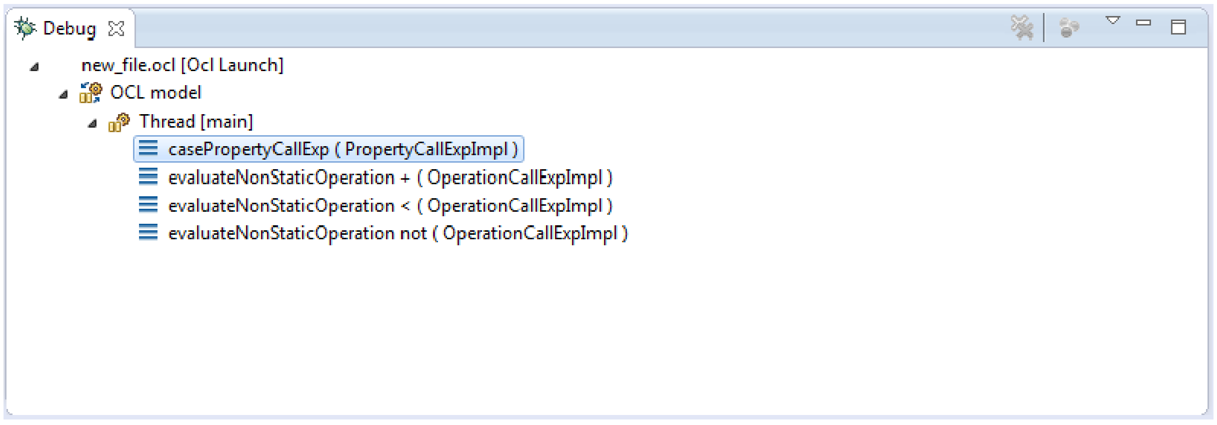
The `IVariable` represents a variable in the stack frame. The `IVariable` also supports the ability, to change a value of a variable while the program is suspended, but running (e.g. the Java Platform Debugger Architecture supports Hot Code Replacement).

IValue

The `IValue` represents a value of a variable. It can represent complex data structures and therefor contain variables itself.

3.2 Interacting with the Debug Model

All commands controlling the debugger are issued from Eclipse's user interface thread. Thus, these actions must be non-blocking. That means, they have to return just after they are called in order to not freeze the user interface. For example, if an action will result in a long running operation, the user interface's thread is waiting for the action to return and cannot render the user interface or handle user inputs anymore.


Debug

new_file.ocl [Ocl Launch]
OCL model
Thread [main]
casePropertyCallExp (PropertyCallExpImpl)
evaluateNonStaticOperation + (OperationCallExpImpl)
evaluateNonStaticOperation < (OperationCallExpImpl)
evaluateNonStaticOperation not (OperationCallExpImpl)

Figure 3.2: A Stack Frame of a suspended Debug Process

3.3 The Execution Control Commands

Figure 3.3 shows and explains the icons to send *Execution Control Commands* to the debug process and/or its threads. In the following the shown commands are explained:

- `Resume` (hit F8 when the process or thread is suspended): makes the process or thread to continue interpretation where it was stopped (either by reaching a breakpoint or by a suspend command),
- `Suspend`: suspends a process or thread in order to stop, but not terminate it,
- `Terminate`: terminates a process and end execution,
- `Step into` (hit F5): to step into the statement the thread suspended at (e.g. if the interpreter suspended by reaching an operation call a `step into` will cause the call of the operation and then a new suspend event is fired after entering the operation call),
- `Step over` (hit F6): interprets the statement the thread suspended at without stepping into and suspends again afterwards,
- `Step return` (hit F7): to return from the statement currently stepping in.

The *Execution Control Commands* can be used to control the debugging process. They provide the ability to observe the execution in a fine-grained manner (e.g. to step into and step over) or in a course-grained way using resume and breakpoints for debugging.

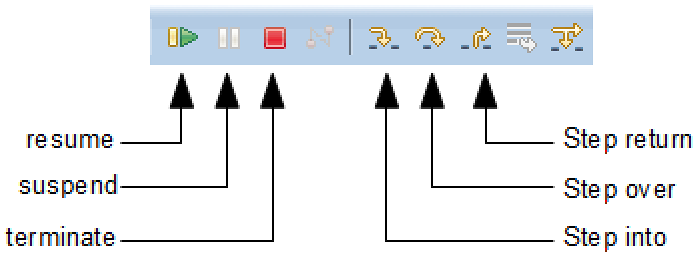


Figure 3.3: The Eclipse's User Interface for Debug Commands

4 Requirements Analysis and Related Work

This chapter presents and discusses the requirements for a step by step debugger for Dresden OCL. It presents related work in the field of model-based debugging and relates it to this work.

4.1 Requirements Analysis

This section presents and discusses the requirements for a step by step debugger for Dresden OCL. The requirements result from the analysis of the Debug Framework of Eclipse presented in Chapter 3, since the Debug Framework forms the foundation the Dresden OCL Debugger is built on top of.

Several subtasks have to be realized to implement the debugger:

/R1/ A mapping from the Abstract Syntax Model to the Abstract Syntax Tree,

/R2/ an implementation of the various interfaces of the Eclipse Debug Framework,

/R3/ extending the OCL Interpreter to provide the information for the Debug Model,

/R4/ and realizing an instrumentation of the existing OCL Interpreter.

The mapping from ASM to AST

To check if an element interpreted by the interpreter is constrained by a line breakpoint there have to be more information available. The line breakpoints are defined in the editor. Thus, they relate to elements of the AST. For EMFText-based debuggers, breakpoints can be retrieved from EMF-based AST resources. In particular, they are defined on the elements in that line. When parsing the AST, the elements of the ASM are generated. For earch element generated, its originating element of the AST must be stored, otherwise it is unclear to which expressions in the ASM each individual breakpoint relates to. Currently, there is no such mapping. The constraints defined in the OCL file are stored on the corresponding model. Therefor the model will own the information which maps an element of the ASM to the element of the AST it originated from. More specifically, we need:

/R11/ A map containing the elements of the ASM and their corresponding element in the AST they originated from,

/R12/ an algorithm computing the line a given ASM element originated from.

The Eclipse Debug Model implementation

EMFText provides debugging support for their generated editors and *Domain Specific Languages (DSL)s*. But this feature has some conditions Dresden OCL cannot fulfill. EMFText assumes that (1) the abstract syntax itself is executable; and (2) the interpreter has to be stack-based.

The first condition is unfeasible due to "[...] OCL is a modeling language in the first place, OCL expressions are not by definition directly executable" [OMG12, p. 21]. OCL's Abstract Syntax Tree is parsed to an Abstract Syntax Model using the Dresden OCL Parser. The ASM is then used during interpretation to evaluate the constraints on the model instance element (see Section 2.4). The second condition means that the interpreter uses a stack to push and pop the elements being interpreted. These preconditions render the auto-generated debugging support unusable for Dresden OCL. As a result the Debug Model has to be implemented explicitly in this work:

/R21/ Implement the interfaces provided by *Eclipse's Debug Model*,

/R22/ implement the debug user interface to provide debugging support,

/R23/ implement the launch support to launch OCL files.

As a non-functional requirement Section 3.2 already stated the need of non-blocking functions called from the UI thread:

/R24/ The functions called by the Eclipse's user interface must be non-blocking.

Since OCL is side effect free [OMG12, p. 21] (e.g. without altering the state of the model) the debugger could debug backwards. This means, the ability to step back and to reverse the last operation. This criterion will not be part of this work, but is listed here as demarcation criterion:

/R25/ The UI offers the ability to step back to issue the debugger to reverse the last operation.

The instrumentation of the Interpreter

When interpreting constraints the interpreter takes a constraint and a model instance element as input and interprets the constraint on the model instance element. The user is interested in the result of the interpretation. However, while debugging the user is interested in (parts of) information during the interpretation, e.g.:

/R31/ The value of a property of the model instance element,

/R32/ the value of a variable, e.g. as defined by a LET expression,

/R33/ the variables and their values regarding the current scope of execution,

/R34/ the result of an operation call,

/R35/ the execution sequence (stack frame).

Furthermore, a debugger provides the ability to stop and continue interpretation step by step. To stop conditionally either because of a stop condition has been triggered or a breakpoint has been hit. This requires the interpreter to provide interfaces for instrumentation. Especially, to be instrumented from outside, e.g. to suspend or resume. The interpreter has to behave differently regarding its execution mode. The execution mode can either be the debug mode or the normal run mode. The execution modes can have an impact on both the execution time the interpreter needs and the resources it uses. During debugging the execution time and the consumed resources will be higher. To sum up, the interpreter needs to supply the following:

/R41/ interfaces to make the interpreter suspend, resume or abort,

/R42/ interfaces to instruct the interpreter (e.g. the commands shown in Figure 3.3),

/R43/ interfaces to add and delete breakpoints,

/R44/ suspend if a breakpoint is hit,

/R45/ trace the execution sequence and provide a stack frame,

/R46/ save variables and their values during each execution step.

4.2 Related Work

To the best of my knowledge, there is currently no existing work where real step by step debugging is applied to OCL. In [BGHK12] debugging is understood as "term evaluation". Brünningen et al. developed an "evaluation browser" for USE[1] which shows the evaluation of invariants on a given state model (UML object diagram) in which the state model is used as a test case. This view can be filtered, e.g. to show failed evaluations only and to examine the cause of the failure. Their approach currently works for invariants only. Dresden OCL provides this feature, too. It is named *Dresden OCL Tracer* [WTFS12, p. 46] and allows to view the evaluation of any expression and the interim results of the subexpressions. Table 4.1 shows these tools and their features regarding their debugging or tracing support.

Tool	Features		
	graphical Debugging	Debugging	Trace outputs
Dresden OCL before this work	no	✓	✓
Dresden OCL after this work	✓	✓	✓
USE	no	partially	✓
OCLE	no	partially	no
Eclipse MDT/OCL	no	no	no

Table 4.1: The features of the OCL tools regarding their debugging and tracing support

"[OCLE[2] was] initially designed with the NEPTUNE[3] project [...] but development was stopped in 2005" [CODA+11, p. 8]. They offer debugging support by means of identifying erronous sub-expressions. Eclipse MDT/OCL[4] offers currently no interactive debugging support. At best, the OCL Console can be used to evaluate partial OCL expressions.

With the OCL Debugger developed in this thesis Dresden OCL will be the only OCL tool that allows graphical debugging support at this time.

[1]USE: UML-based Specification Environment, `http://sourceforge.net/projects/useocl/`
[2]OCLE: OCL Environment, `http://lci.cs.ubbcluj.ro/ocle/index.htm`
[3]IST 1999-20017 FP5 EU research project, `http://neptune.irit.fr`
[4]MDT/OCL or Eclipse OCL, `http://wiki.eclipse.org/MDT/OCL`

5 Design and Structure

This chapter presents the architecture of the developed debugger. It describes the new introduced plug-ins, classes and interfaces and how they play together. Furthermore, it describes how the debugging process is working in detail. Finally, it is presented how the debugger has been tested.

5.1 Architecture

This section introduces the architecture of the developed debugger. Specifically, the package structure and the class structure are presented.

5.1.1 Package Structure

The debugger is realized as a new plug-in of *Dresden OCL*. Since it extends the OCL interpreter, which has been introduced in Section 2.4, it inherits its dependencies. Furthermore, it is dependent on the *Model* plug-in, the *Model Instance* plug-in, the *Model Bus* plug-in and the *OCL Resource* plug-in. Figure 5.1 shows these dependencies.

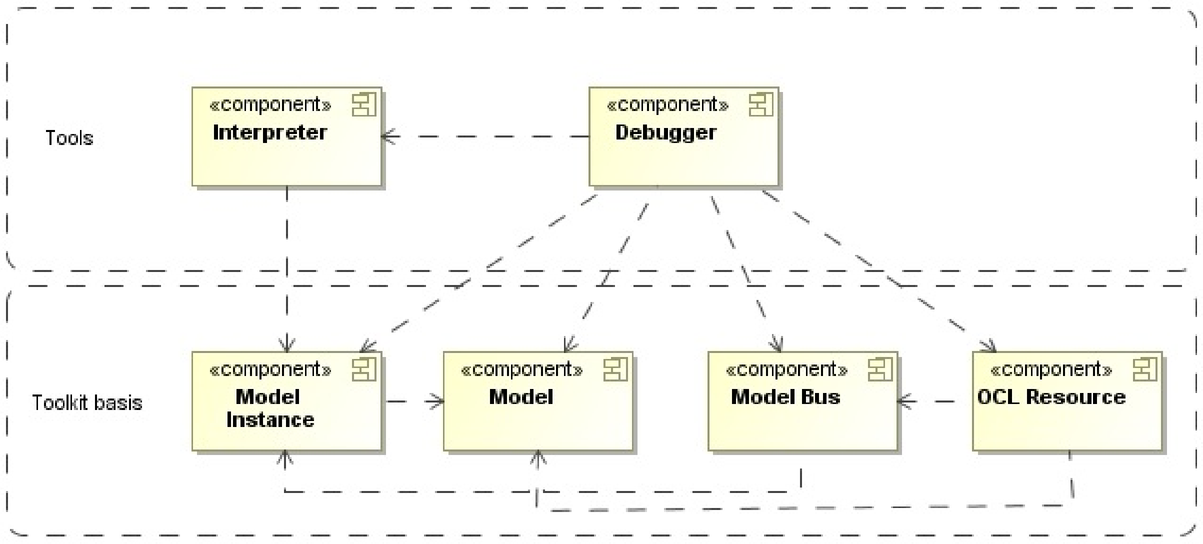


Figure 5.1: Dependencies of packages involved with the debugger

5.1.2 Class Structure

This section describes some classes and interfaces that have been implemented for the debugger.

Public Interfaces

The debugger plug-in defines one public interface that shall be used to refer to a debugger. It is called `IOclDebuggable` and is described in Figure 5.2. It extends the `IOclInterpreter` and offers therefor the ability to use the debugger wherever the interpreter is used as type or argument.

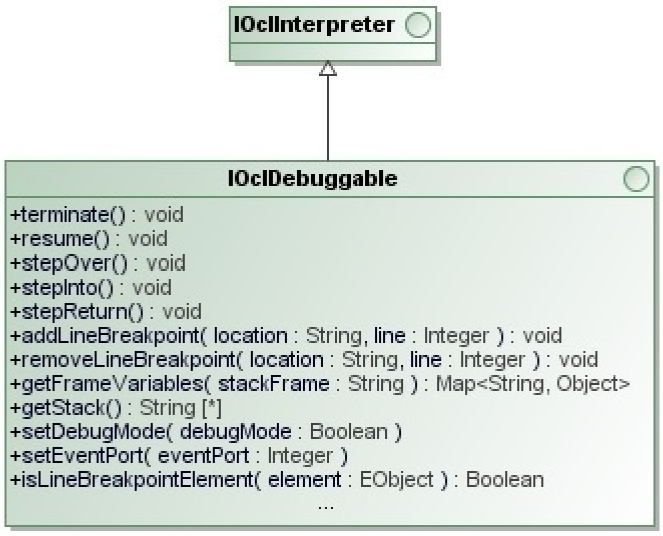

IOclInterpreter
IOclDebuggable
+terminate() : void
+resume() : void
+stepOver() : void
+stepInto() : void
+stepReturn() : void
+addLineBreakpoint(location : String, line : Integer) : void
+removeLineBreakpoint(location : String, line : Integer) : void
+getFrameVariables(stackFrame : String) : Map<String, Object>
+getStack() : String [*]
+setDebugMode(debugMode : Boolean)
+setEventPort(eventPort : Integer)
+isLineBreakpointElement(element : EObject) : Boolean
...

Figure 5.2: The interface of the OclDebuggable

The `IModel` does now provide the methods `getAllMappings()` in order to get the current mapping from the elements of the ASM to the AST. `setAllMappings(Map)` sets the mapping, respectively.

5.2 The OCL Debug Model

The central component of the debugger is the implementation of the various interfaces provided by the *Eclipse Debug Model* introduced in Section 3. An important aspect of the implementation of all those interfaces is the coercion to thread safe programming, including classes that must be single threaded, because of the Eclipse Platform is throughout multi-threaded.

The `OclDebugTarget` is the central class of our debug model. This is, because it "[...] is the root of the debug element hierarchy" [URL12c]. Launching a debug session will always result in the creation of a debug target. It is responsible to keep track of all used threads, the breakpoints and the event dispatcher. Debugging OCL is done in a single threaded environment; the OCL debugger. The `OclDebugger` runs in parallel in its own thread and is issued with commands from the UI or the other debug elements (see Section 3.3).

The `OclDebugProxy` is responsible for sending the commands to the debugger. It does not directly communicate with the debugger, but uses the `OclDebugCommunicationHelper` to asynchronously send requests and sometimes wait for their results. The proxy sends the requests to the request port . The `OclDebuggerListener` provides a server and forwards the requests to the `IOclDebuggable`. The debugger proccesses the request and responses with an event, if neccessary. The `EventDispatchJob` in the `OclDebugTarget` uses the `OclDebugCommunicationHelper` to receive those events and to notify the listeners to these events: the `OclDebugProcess`, the `OclDebugThread` and the `OclDebugTarget` itself. Figure 5.3 shows the correlation between all these classes and clarifies the calls between them exampled with a terminate command issued from the user interface.

In Section 3.2 the coercion to thread safe programming and non-blocking operations has been explained. This requirement led to the implementation of a non-blocking communication mechanism using sockets. There are two ports used for communication. One port for *requests* and the other for *events*. Requests are used to send commands to the debugger, e.g. to suspend or to terminate the debugger. Events are used to listen for incoming commands to be executed or answered. The use of sockets over a simple listener is that they are buffering messages and therefor queuing them. Figure 5.4 shows the sequence of events when starting the debugging process. Using listeners, after the third call the fifth could happen before the fourth. The message would be lost. With the use of sockets the message is kept and queued eliminating race conditions. Furthermore, they can be used to wait for requested answers. If the `OclStackFrame` is asked for variables it contacts the `OclDebugProxy` to get the variables belonging to it. Then a request is issued and the socket reader is immediately *blocked* until the answer returns. A simple listener could not realize that.

The `OclDebugCommunicationHelper` therefor defines three functions for the different types of communication patterns: (1) `sendEvent()` to send a message into the `PrintStream`[1]. The stream is synchronized. This means, it can be written just once at a time. (2) `receive()` to receive a message from the reader. Calling this message blocks until a message is received. (3) `sendAndReceive()` to do both; send and receive as explained before.

5.3 The Mapping from ASM to AST

Section 4.1 discussed the fundamental reasons for a mapping from the ASM to the AST. This section will explain how this mapping was implemented and discusses the decisions made. Changes are made to the *OCL staticsemantics checker*, in particular to the `OclParseTreeToEssentialOcl` (see Listing 5.1). While creating the element of the ASM, it also stores the AST's element in a map. The resulting mapping is promoted via the model. Currently, the `IModel` holds the constraints defined for that model. Thus, the `IModel` will hold the mapping, too. The line of an element of the ASM can be determined by computing the line of the corresponding AST's element (cf. Listing 5.2).

```scala
case c : CollectionLiteralPartsOclExpCS => {
  (computeOclExpression(c.getOclExpression)).flatMap { oclExpressionEOcl => {
    val ci = factory.createCollectionItem(oclExpressionEOcl)
    allMappings.put(ci, c)
    Full(ci)
    }
  }
}
```

Listing 5.1: An excerpt of the mapping from ASM to AST

Listing 5.1 shows a Scala code snippet of the `OclParseTreeToEssentialOcl trait`[2]. In

[1] A print stream is a stream you can write into. The sink must not be known.

[2] "A trait encapsulates method and field definitions, which can be resued by mixing them into classes" [OSV10, p. 258]

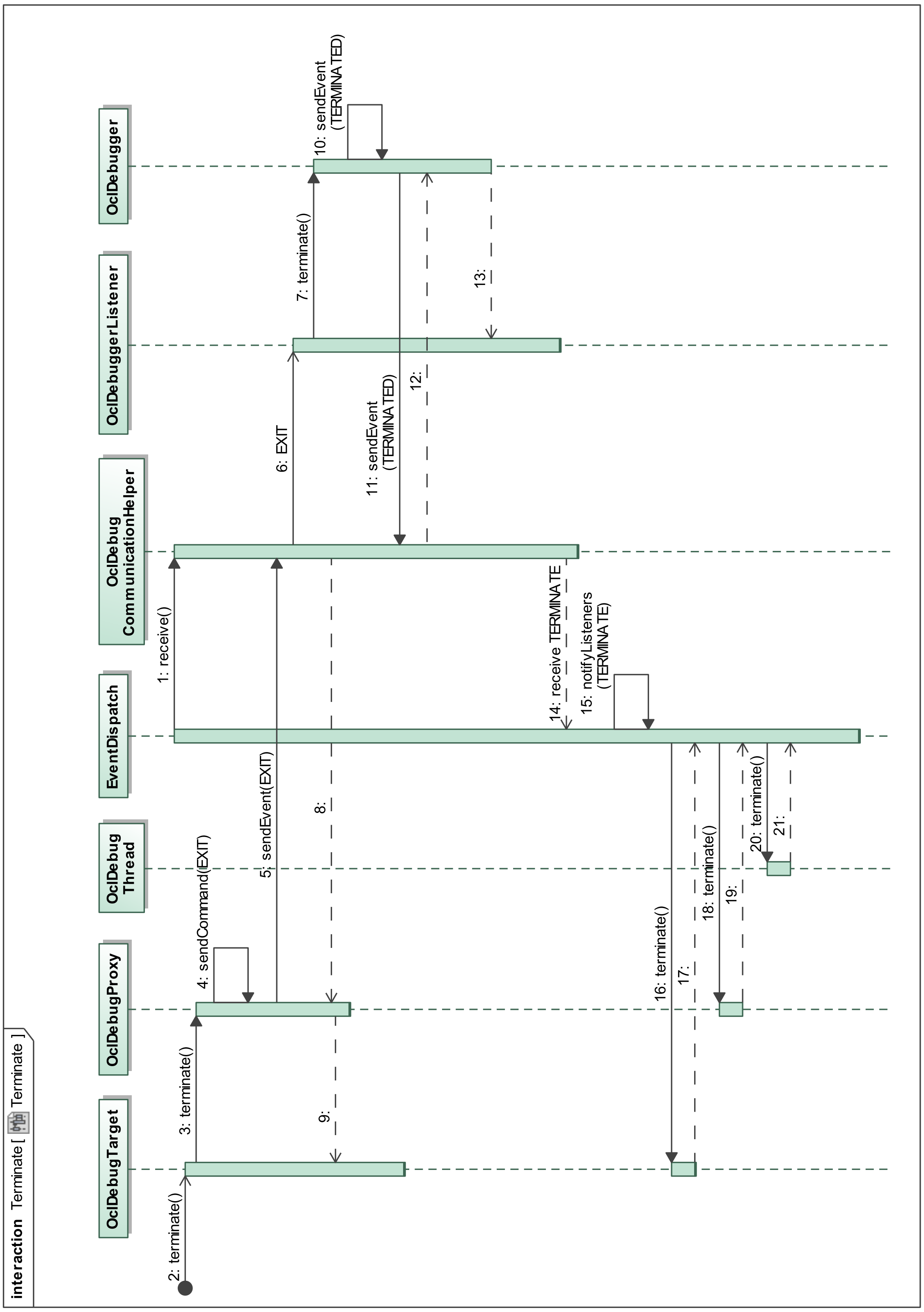


Figure 5.3: Synchronous and asynchronous calls after issuing a terminate command from the UI

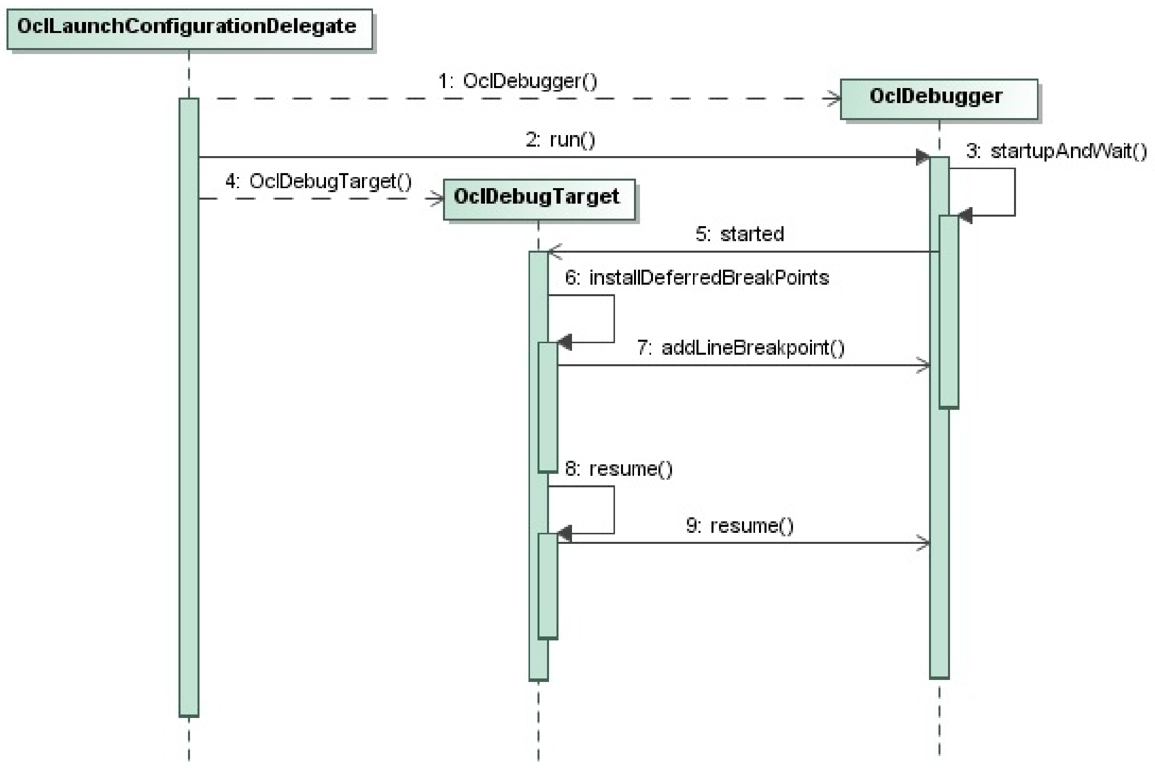


Figure 5.4: The Interaction between the Launch, Debugger and the Debug Target

case of a `CollectionLiteralPartsOclExpCS` (e.g. `Set = {1, 2}` the `1` and `2` are the expressions stating a simple `IntegerLiteralExpCS`) the OCL expression gets computed in line 2, and the `CollectionItem` will be created. This `CollectionItem` is then mapped in line 4 on the `CollectionLiteralPartsOclExpCS` it originated from.

```
1      /**
2       * Computes the line of the EObject in the containing resource.
3       * @param element the EObject element
4       * @return the line the element was defined in the resource
5       */
6      protected int getLine(EObject element) {
7  
8          EObject e = m_currentMappings.get(element);
9          OclResource resource = (OclResource) e.eResource();
10         int line;
11         do {
12             line = resource.getLocationMap().getLine(e);
13             e = e.eContainer();
14         } while (line == -1 && e != null);
15         return line;
16     }
```

Listing 5.2: The function computing the line number of an element of the ASM

The function presented in Listing 5.2 is defined in the `OclDebugger`. The function gets an `EObject` passed as argument. That is a super type the ASM elements inherited from. In lines 11-15 the line of the AST element in the OCL editor is determined and returned.

As discussed in Section 4.1, the mapping from the ASM to the AST is essential to realize OCL debugging for Dresden OCL. However, its implementation turned out not to be a big task and could be realized rather easily.

5.4 The OCL Debugger

This section introduces the debugger realized in this work and how it has been tested.

5.4.1 The Implementation of the Debugger

In fact the debugger is a specialization of the interpreter. It does indeed interpret, but collects a lot more information at the same time. Section 4.1 stated all the requirements the debugger has to fulfill. In order to achieve this, the debugger must have access to the internals of the already implemented `OclInterpreter`: First, it needs to change the startup process of the interpreter. Second, the debugger has to call more functions during interpetation. Third, it has to collect the information during each step and send it to the event socket. Last, the debugger has to provide interfaces to e.g. suspend the interpretation. This would result in a lot of conditional checks for every method (e.g. checks for the execution mode). Therefor, the `OclInterpreter` has been extended by the `OclDebugger` implementing the `IOclDebuggable` (cf. Figure 5.2). This solution comes with the advantage that the original implementation of the interpreter does not suffer in time and space needed for interpretation.

As mentioned before, the `OclDebugger` implements the `IOclDebuggable`. It also extends the `OclInterpreter`. On startup it starts the server socket and suspends in order to listen to incoming commands. The debugger encapsulates the `caseXXX` methods of the interpreter (e.g. `caseOperationCallExp(OperationCallExp)`) and checks if the ASM element is constrained by a breakpoint. If it is constrained by a breakpoint the debugger suspends, otherwise it resumes the interpretation unmodified. Listing 5.3 shows how this is realized for the `caseOperationCallExp` function. The debugger checks and eventually suspends before and after the call (depending on the current stepping) to the super implementation in order to show the stack frame and variables before and after the call, e.g. after returning from an `OperationCall` the parameters and the source of the `OperationCall` are presented as variables (actually the variables are named as *source* and as *param*).

```java
@Override
public OclAny caseOperationCallExp(OperationCallExp operationCallExp) {

    String operationResultName =
            "result of " + operationCallExp.getReferredOperation().
                getName();

    OclAny result = super.caseOperationCallExp(operationCallExp);
    myEnvironment.setVariableValue(operationResultName, result);
    stopOnBreakpoint("caseOperationCallExp", operationCallExp);
    popStackFrame();
    myEnvironment.deleteVariableValue(operationResultName);
    return result;
}
```

Listing 5.3: The function caseOperationCallExp of the OclDebugger

To support *step return* and *step over* commands, the `OclDebugger` needs to log every step at any time during interpretation. When in step through mode (by issuing step into after suspending), before and after every case-method call a *virtual* breakpoint have to be set. These virtual breakpoints have to be connected to the method. When *step return* gets issued the next *virtual* breakpoint after leaving the method currently stepping in will be the next step. Since every function call of the `OclInterpreter` is encapsulated before and after with breakpoint checks, issuing step return would result in suspending the debugger when the latter check is done and the debugger has returned from that function. This means, that on every first check the function has to be marked in order to know where to stop. However, *step over* requires to execute the next case-method call without stepping into it, but execute the case-method call and immediatelly suspend after returning from it.

The variables and their values are stored in the `OclInterpreter` in a data structure called `InterpretationEnvironment` that holds the visible variable values hierarchically. Thus, reaching a new scope while interpreting the `InterpretationEnvironment` creates a new empty environment that links to its parent. So there is an unambiguous assignment of the variables during interpretation. The debugger uses this environment to determine the variables visible in every stack frame.

5.4.2 Testing the Debugger

The `OclDebugger` was tested extensively by hand. This means, constraints were debugged and the overall output was observed manually. However, to allow further development and enhancement as well as bugfixing of the OCL debugger developed during this work, unit tests are required to test the debugger in a systematic and automated manner. Thus, first unit tests testing the `OclDebugger` and the most important features, e.g. starting and suspending or to suspend on breakpoint hit have been developed. They found the basis for the development of further, systematic unit tests testing the functionality of the OCL debugger for all kinds of expressions being part of OCL. But, due to the limited time for the completion of this work, there are currently no integration tests. Integration testing can be done by mocking the whole OCL debug model in the package `org.dresdenocl.debug.model`. Designing further unit tests as well as integration tests is a definite taks for future works.

6 Graphical User Interface Implementation

This chapter describes how the OCL Debugger realized in this work may be used. Dresden OCL in general was introduced in Chapter 2. With the OCL Debugger you can debug `.ocl` files with Dresden OCL. In the following, we use the `allConstraints.ocl` (cf. Listing A.1) of the *Simple Example (UML/Java)* (see Figure 6.1) provided with Dresden OCL's Examples.

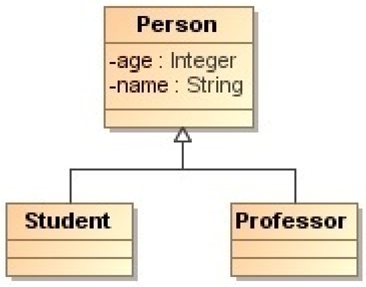


Figure 6.1: The Class Diagram of the Simple Example

6.1 The Dresden OCL Debug Perspective

This section shows the *Debug Perspective* and each of the views being part of it. This is the *Launch Configuration Dialog*, the *Variables View*, the *Breakpoints View* and at last the *Stack Frame View*.

The Dresden OCL Launch Configuration Dialog

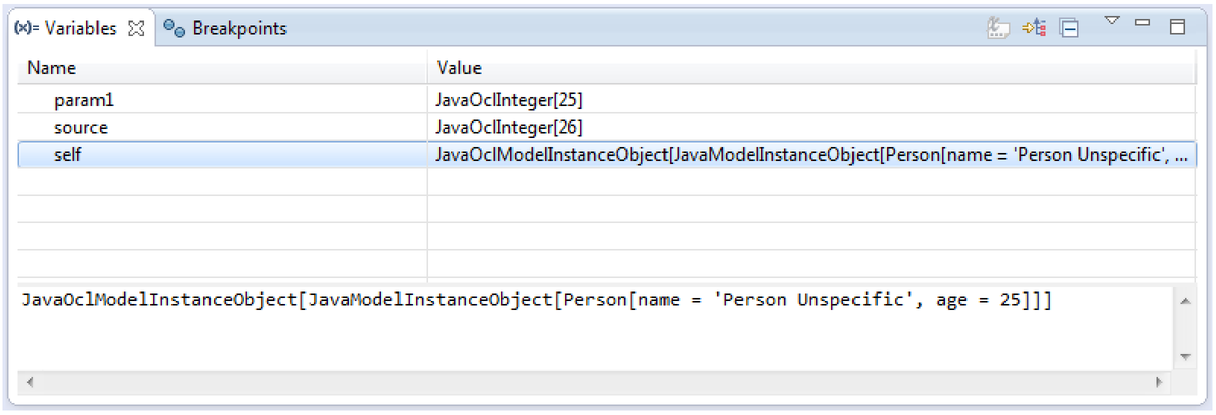
The Dresden OCL *Launch Configuration Dialog* shown in Figure 6.3 allows to configure options for the debugging process. Currently, the resource URL can be changed.

The Dresden OCL Variables View

The *Variables View* (see Figure 6.2) allows to see the variables in the current selected stack frame. Switch the stack frame selection in order to see the variables of the other stack frames. While debugging, the latest stack frame get selected automatically and the latest variable values are presented.

The Dresden OCL Breakpoints View

The *Breakpoints View* shows all breakpoints defined. The colored dot before the breakpoint entry shows, whether the breakpoint is active or not. If the dot is filled blue, it is active. Is it a blue circle with white body it is inactive. Figure 6.4 shows a breakpoint view with active breakpoints.



Name	Value
param1	JavaOclInteger[25]
source	JavaOclInteger[26]
self	JavaOclModelInstanceObject[JavaModelInstanceObject[Person[name = 'Person Unspecific', ...

```
JavaOclModelInstanceObject[JavaModelInstanceObject[Person[name = 'Person Unspecific', age = 25]]]
```

Figure 6.2: The Variables view showing the variables of a selected stackframe

The Dresden OCL Stack Frame View

The *Stack Frame View* shows each thread being executed in the current debugging process and the call stack for every single thread. Figure 3.2 shows a stack frame while debugging. The call stack of each thread can be collapsed using the arrow before the thread entry.

6.2 Using the Debugger

This section explains how the debugger is used in order to debug a set of OCL expressions on *Model Instance Elements*. First, you have to load a *Model*. Then, you have to load the *Model Instance*. At last you have to start the debugging process.

6.2.1 Selecting a Model

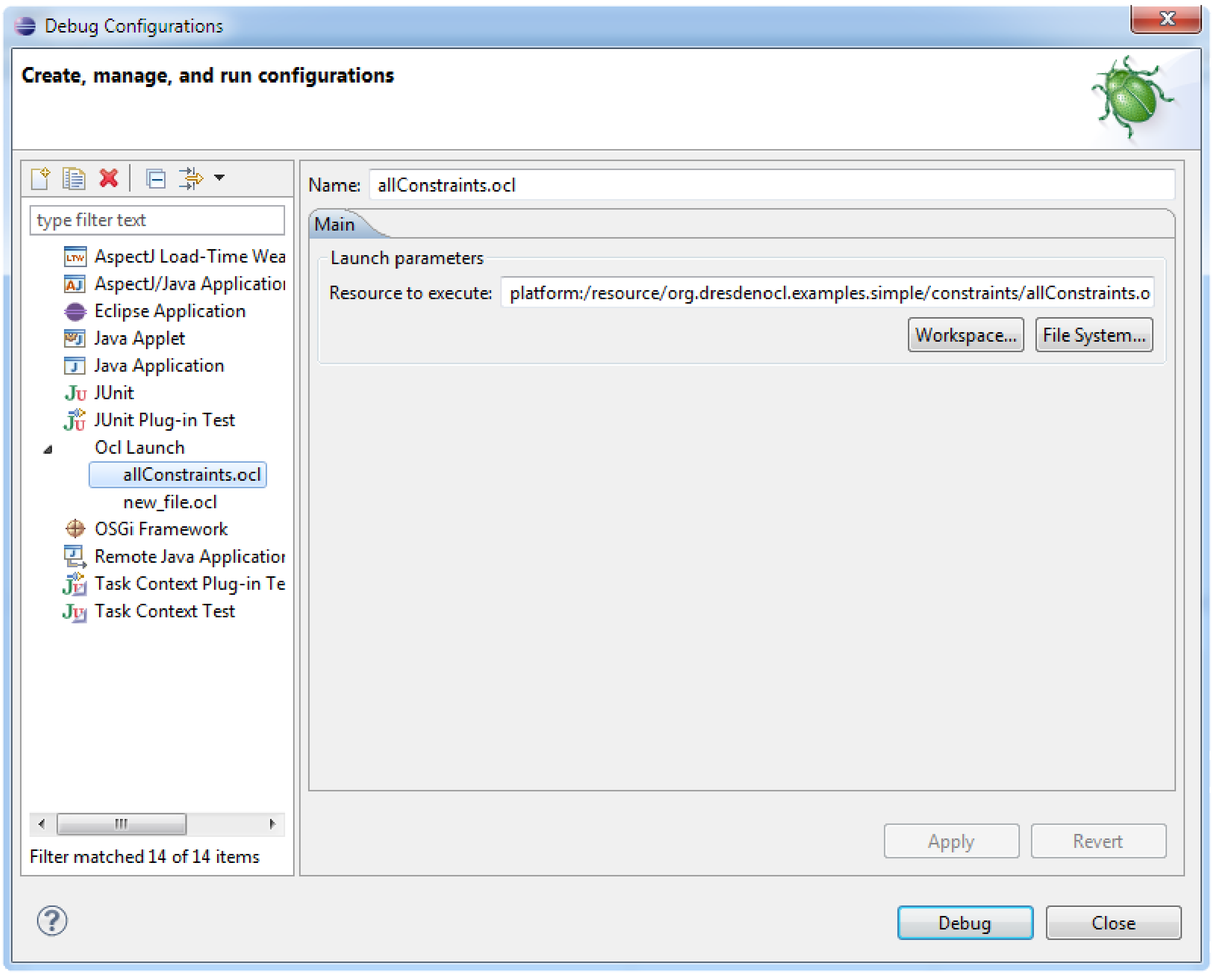
To select a model use the context menu entry under *Dresden OCL > Load as Model ...* or better use the `@model{}` annotation at the very beginning of the constraint file (as comment before the `package` command) in order to load the model automatically.

6.2.2 Selecting a Model Instance

To select a model instance use the context menu entry under *Dresden OCL > Load as Model Instance ...* on a proper *Model Instance* file.

6.2.3 Debugging

Figure 6.5 shows the context menu entry to start a debug session. Eclipse will ask whether to switch to the *Debug Perspective* if it is not already the active perspective. The *Debug Perspective* is shown in Figure 6.6. Since the debugger is embedded in the *Eclipse Debug Framework* the standard shortcuts (F5-F8) may be used in order to work with the debugger (i.e. to issue the execution control commands). The execution control commands issuable from the UI are explained in Section 3.3.


Debug Configurations

Create, manage, and run configurations

type filter text

- AspectJ Load-Time Wea
- AspectJ/Java Applicatio
- Eclipse Application
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Ocl Launch
  - allConstraints.ocl
  - new_file.ocl
- OSGi Framework
- Remote Java Applicatio
- Task Context Plug-in Te
- Task Context Test

Filter matched 14 of 14 items

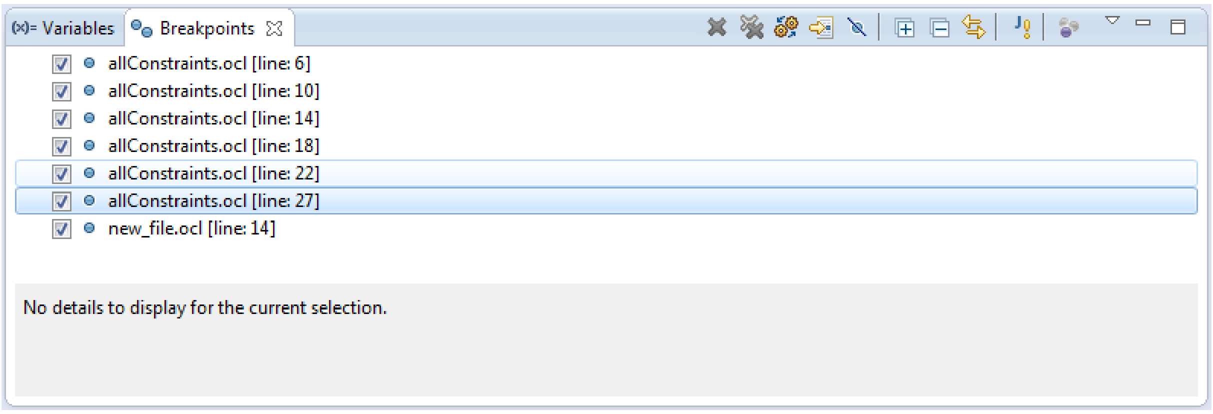
Name: allConstraints.ocl

Main

Launch parameters

Resource to execute: platform:/resource/org.dresdenocl.examples.simple/constraints/allConstraints.o

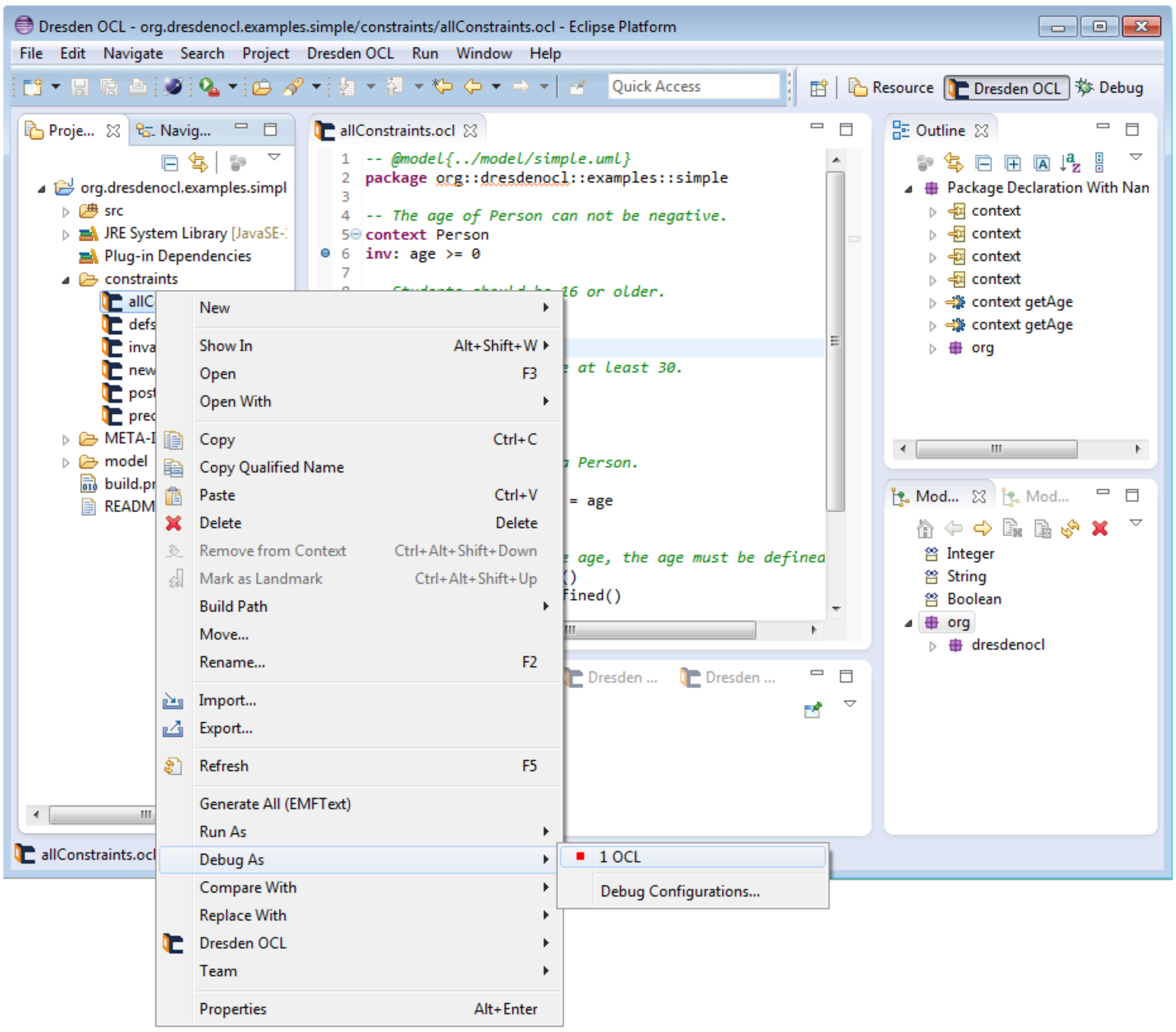
Figure 6.3: The Launch View for launching the Dresden OCL Debugger



- [✓] allConstraints.ocl [line: 6]
- [✓] allConstraints.ocl [line: 10]
- [✓] allConstraints.ocl [line: 14]
- [✓] allConstraints.ocl [line: 18]
- [✓] allConstraints.ocl [line: 22]
- [✓] allConstraints.ocl [line: 27]
- [✓] new_file.ocl [line: 14]

No details to display for the current selection.

Figure 6.4: The Breakpoints view showing the currently declared breakpoints and their files



```
-- @model{../model/simple.uml}
package org::dresdenocl::examples::simple

-- The age of Person can not be negative.
context Person
inv: age >= 0

-- Students should be 16 or older.
```

```
 at least 30.
```

```
 Person.
```

```
= age
```

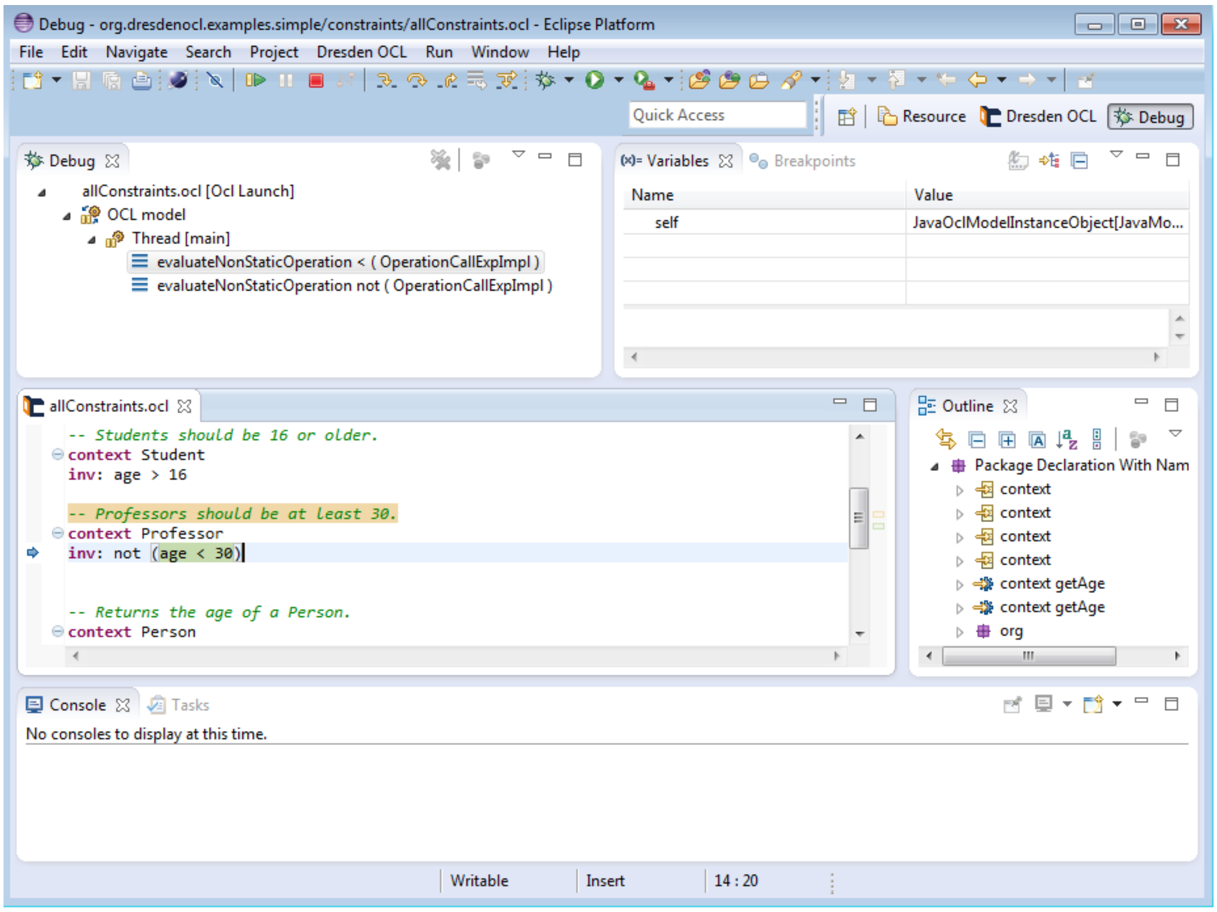
```
 age, the age must be defined
()
fined()
```

Figure 6.5: The context menu entry for *Debug as …*

After switching to the *Debug Perspective* the debugging process starts automatically. Thus, either the suspend command is issued from the UI on the process or thread (depends on the selection in the *Stack Frames View*, see Figure 3.2) or the debugger has to hit an active breakpoint in order to suspend; or it finishes interpretation and terminates. On suspending, the last stack frame is selected and the current variables are shown in the *Variables View*. On breakpoint hit, it is possible to either step into (F5), step over (F6), step return (F7) or to resume (F8) until all constraints are interpreted on all *Model Instance Elements*. Then, the debugger terminates.

6.3 Summary

This chapter described the *Debug Perspective* with all its views, which are the *Launch Configuration Dialog*, the *Variables View*, the *Breakpoints View* and at last the *Stack Frames View*. It explained how to load a *Model* and *Model Instance* and how to start debugging. Furthermore, it described how to control the debugger in order to get the appropriate information of the debug process (e.g. variable values at a special stack frame).


Debug - org.dresdenocl.examples.simple/constraints/allConstraints.ocl - Eclipse Platform
File Edit Navigate Search Project Dresden OCL Run Window Help
Quick Access
Resource
Dresden OCL
Debug
Debug
Variables
Breakpoints
allConstraints.ocl
Outline
Console
Tasks
Writable
Insert
14 : 20

```
allConstraints.ocl [Ocl Launch]
  OCL model
    Thread [main]
      evaluateNonStaticOperation < ( OperationCallExpImpl )
      evaluateNonStaticOperation not ( OperationCallExpImpl )
```

Name	Value
self	JavaOclModelInstanceObject[JavaMo...

```
-- Students should be 16 or older.
context Student
inv: age > 16

-- Professors should be at least 30.
context Professor
inv: not (age < 30)


-- Returns the age of a Person.
context Person
```

```
Package Declaration With Nam
  context
  context
  context
  context
  context getAge
  context getAge
  org
```

No consoles to display at this time.

Figure 6.6: The debug perspective of Dresden OCL Debugger

7 Evaluation and Future Work

This chapter evaluates the implementation presented in Chapter 5 and checks whether the requirements identified in Section 4.1 were fulfilled. At first, the tasks of this work are discussed. Then, the features realized are presented. Finally, this chapter presents tasks that were not implemented and should be addressed by future works.

7.1 Evaluation

This section will evaluate whether the developed OCL debugger fulfills the requirements analyzed in Section 4.1. The tasks of this work have been the realization of a graphical OCL debugger for Dresden OCL that is integrated into the *Eclipse Debug Framework* and the consequent subtasks. This includes the implementation of the various interfaces provided by the *Eclipse Debug Framework*, namely the *Eclipse Debug Model*. Furthermore, it encompasses the introduction of a mapping from the *Abstract Syntax Model* to the *Abstract Syntax Tree* enabling the visualization of the interpretation process. Besides, the implementation had to be tested and documented.

The result of this work is the implementation of a graphical OCL debugger for Dresden OCL which is integrated into the *Eclipse Debug Framework*. There is still further work to do in order to release the debugger with Dresden OCL. This includes further testing and the implementation of all step modes. The resulting OCL debugger has been tested with unit tests. Integration tests have yet to be done. In addition, it has been tested manually intensively.

The requirements for the realization of an OCL debugger were analyzed in Section 4.1. Table 7.1 lists these requirements and checks whether they have been realized during this work. All requirements have been fulfilled besides the demarcation criterions. They should be addressed by future works.

7.2 Future Work

Despite the debugger implemented in this work there are some features missing. Some are evaluated in Section 4.1. These can be realized in future works. Some of these tasks are:

- Currently, on mouse hover the `OclEditor` does not show the values of variables during debugging. This can be provided with implementing a debugging-aware `OclHoverText-Provider`. This means, i.e. on hovering a variable definition in the OCL file the current value will be shown.

- Currently, the `OclEditor` does give not any feedback whether breakpoints can be hit or not. Especially, when breakpoints are defined on comments, these breakpoints can never be reached. Future work could address this feature by integrating checks on breakpoint reachability into the static semantics plug-in of Dresden OCL or with another post processor.

No.	Requirement Description	Now Supported
/R1/	**A mapping from the Abstract Syntax Model to the Abstract Syntax Tree consisting of:**	
/R11/	A map containing the elements of the ASM and their corresponding element in the AST they originated from.	✓
/R12/	An algorithm computing the line a given ASM element originated from.	✓
/R2/	**An implementation of the various interfaces of the Eclipse Debug Framework. This includes the following requirements:**	
/R21/	Implement the interfaces provided by Eclipse's Debug Model.	✓
/R22/	Implement the debug user interface to provide debugging support.	✓
/R23/	Implement the launch support to launch OCL files.	✓
/R24/	The functions called by the Eclipse's user interface must be non-blocking.	✓
/R25/	The UI offers the ability to step back to issue the debugger to reverse the last operation.	x
/R3/	**The OCL Interpreter must provide the information for the Debug Model. This includes:**	
/R31/	The value of a property of the model instance element.	✓
/R32/	The value of a variable, e.g.as defined by a LET expression.	✓
/R33/	The variables and their values regarding the current scope of execution.	✓
/R34/	The result of an operation call.	✓
/R35/	The execution sequence (stack frame).	✓
/R4/	**Realizing an instrumentation of the existing OCL Interpreter.**	
/R41/	Interfaces to make the interpreter suspend, resume or abort.	✓
/R42/	Interfaces to instruct the interpreter.	✓
/R43/	Interfaces to add and delete breakpoints.	✓
/R44/	Suspend if a breakpoint is hit.	✓
/R45/	trace the execution sequence and provide a stack frame.	✓
/R46/	Save variables and their values during each execution step.	✓

Table 7.1: The requirements for the Debugger for Dresden OCL

- Breakpoints may be conditional. This means, they are either constrained by a hit-counter (triggering by a defined hit count) or a condition that must hold. Both is not implemented in this work. The latter would require (1) to write the condition in OCL; (2) to parse and interpret the condition and evaluate it during debugging.
- OCL is side-effect free. This means, it does not alter the state of a model. Thus, it is possible to debug in *reverse* order using the *drop to frame* possibility of the *Eclipse Debug Framework* to go back to a specified stack frame and replay the steps. Such functionality could be realized in future works, e.g. using a memento pattern [GHJV94, p. 283ff].
- *Dresden OCL* already provides tracing support with the *Dresden OCL Tracer* and the `TracerView`. It may be helpful to see the interim results of the expressions and sub-expressions in the view of the tracer during debugging. These both views may be synchronized with the debugging process in future works.
- Enable the possibility to change the value of variables during debugging in the Debug View. The Eclipse Debug Model supports this feature in its API.
- The *Java Debugger* (jdb) allows to change the implementation of methods, but not their interfaces or parameters, during runtime. This may be a challenging task since the constraints need to be re-parsed and the effected ASM elements need to be exchanged. The references every single old element needs to be exchanged with the new references.

8 Summary and Conclusion

This work presented and implemented a graphical debugger for *Dresden OCL* that can be used to debug OCL expressions using the Dresden OCL integration for Eclipse. The result is embedded in the Eclipse Debug Framework and offers the ability to run and debug OCL expressions on models and their instances.

The work presented how the Eclipse Debug Model can be implemented for OCL and how the interpretation process can be visualized in the user interface. The interpretative way *Dresden OCL* choose to evaluate the constraints on model instance elements is very comfortable for this purpose since it required no refactoring on the interpreter implementation to achieve the debugging support.

As shown in Section 4.2 there is currently no graphical debugging support for OCL in other tools. Despite the non-existent graphical debugging support there are some attempts to support a simple kind of debugging: to expose and visualize faulty sub-expressions, e.g. so does *USE*. The debugger realized in this work is the first to provide graphical debugging support in the field of OCL.

Comparing the debugger realized in this work with other existing debuggers (e.g. the *Java Debugger*) highlights what still can be done. For example the Java Debugger allows to change the implementation of methods, but not the interface or parameters, during runtime. The debugger realized in this work offers not the feature to do this.

Since debugging was one of the most-wanted features identified in a survey among the community [COD10] (among others, e.g. refactoring and auto-completion), Dresden OCL made a big step towards an `IDE4OCL`.

List of Figures

List of Abbreviations

API	Application Programming Interface
ASM	Abstract Syntax Model
AST	Abstract Syntax Tree
CS	Concrete Syntax
DOT	Dresden OCL Toolkit
DOT4Eclipse	Dresden OCL2 for Eclipse
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
Eclipse MDT	Eclipse Modeling Development Tools
EMF	Eclipse Modeling Framework
GDB	GNU Project Debugger
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDB	The Java Debugger
JVM	Java Virtual Machine
MDT	Modeling Development Tools
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
SDK	Software Development Kit
UML	Unified Modeling Language
URL	Uniform Resource Locator
XSD	XML Schema Definition
XMI	XML Metadata Interchange
XML	Extensible Markup Language

A Simple Example

```
-- @model{../model/simple.uml}
package org::dresdenocl::examples::simple

-- The age of Person can not be negative.
context Person
inv: age >= 0

-- Students should be 16 or older.
context Student
inv: age > 16

-- Professors should be at least 30.
context Professor
inv: not (age < 30)


-- Returns the age of a Person.
context Person
def: getAge(): Integer = age


-- Before returning the age, the age must be defined.
context Person::getAge()
pre: not age.oclIsUndefined()


-- The result of getAge must equal to the age of a Person.
context Person::getAge()
post: result = age

endpackage
```

Listing A.1: The allConstraints.ocl of the Simple Example (UML/Java)

Bibliography

[ABB+12] Assmann, Uwe ; Bartho, Andreas ; Bürger, Christoff ; Cech, Sebastian ; Demuth, Birgit ; Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Polowinski, Jan ; Reimann, Jan ; Schroeter, Julia ; Seifert, Mirko ; Thiele, Michael ; Wende, Christian ; Wilke, Claas: DropsBox: the Dresden Open Software Toolbox. In: *Software & Systems Modeling* (2012), 1-37. `http://dx.doi.org/10.1007/s10270-012-0284-6`. – ISSN 1619–1366

[BD08] Bräuer, Matthias ; Demuth, Birgit: Models in Software Engineering. Version: 2008. `http://dx.doi.org/10.1007/978-3-540-69073-3_20`. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 978–3–540–69069–6, Kapitel Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support, 182–193

[BGHK12] Brüning, Jens ; Gogolla, Martin ; Hamann, Lars ; Kuhlmann, Mirco: Evaluating and Debugging OCL Expressions in UML Models. Version: 2012. `http://dx.doi.org/10.1007/978-3-642-30473-6_13`. In: Brucker, AchimD. (Hrsg.) ; Julliand, Jacques (Hrsg.): *Tests and Proofs* Bd. 7305. Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–30472–9, 156-162

[Bra06] Brandt, Ronny: *Java-Codegenerierung und Instrumentierung von Java-Programmen*, Technische Universität Dresden, Diplomarbeit, April 2006

[Bra07a] Brandt, Ronny: *Ein OCL-Interpreter für das Dresden OCL2 Toolkit basierend auf dem Pivotmodell*, Technische Universität Dresden, Diplomarbeit, November 2007

[Brä07b] Bräuer, Matthias: *Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment*, Technische Universität Dresden, Diplomarbeit, May 2007

[COD10] Chimiak-Opoka, Joanna ; Demuth, Birgit: A Feature Model for an IDE4OCL. In: *Electronic Communications of the EASST* 36 (2010)

[CODA+11] Chimiak-Opoka, Joanna ; Demuth, Birgit ; Awenius, Andreas ; Chiorean, Dan ; Gabel, Sébastien ; Hamann, Lars ; Willink, Edward: Workshop on OCL and Textual Modelling (OCL 2011). In: *Electronic Communications of the EASST* 44 (2011)

[DeM79] DeMarco, T.: Classics in software engineering. Version: 1979. `http://dl.acm.org/citation.cfm?id=1241515.1241539`. Upper Saddle River, NJ, USA : Yourdon Press, 1979. – ISBN 0–917072–14–6, Kapitel Structured analysis and system specification, 409–424

[FH11] Fischer, P. ; Hofer, P.: *Lexikon Der Informatik.* Springer, 2011 `http://books.google.de/books?id=8wZUvN5af_AC`. – ISBN 9783642151262

[Fin99] Finger, Frank: *Java-Implementierung der OCL-Basisbibliothek*, Technische Universität Dresden, Diplomarbeit, July 1999

[Fin00] Finger, Frank: *Design and Implementation of a Modular OCL Compiler*, Technische Universität Dresden, Diplomarbeit, March 2000

[Fre11] Freitag, Björn: *Reengineering-Konzept für das DeclarativeCodeGenerator-Framework von Dresden OCL*, Technische Universität Dresden, Diplomarbeit, March 2011

[GHJV94] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1994 `http://books.google.de/books?id=6oHuKQe3TjQC`. – ISBN 9780321700698

[Hei05] Heidenreich, Florian: *SQL-Codegenerierung in der metamodellbasierten Architektur des Dresden OCL Toolkit*, Technische Universität Dresden, Diplomarbeit, August 2005

[Hei06] Heidenreich, Florian: *OCL-Codegenerierung für deklarative Sprachen*, Technische Universität Dresden, Diplomarbeit, March 2006

[Huß00] Hussmann, Heinrich: *Formal Specification of Software Systems.* `http://st.inf.tu-dresden.de/fs/slides/fss5a-sl.pdf`. Version: 2000

[OA99] Offutt, Jeff ; Abdurazik, Aynur: Generating Tests from UML Specifications. Version: 1999. `http://dx.doi.org/10.1007/3-540-46852-8_30`. In: France, Robert (Hrsg.) ; Rumpe, Bernhard (Hrsg.): *≪UML≫'99 — The Unified Modeling Language* Bd. 1723. Springer Berlin Heidelberg, 1999. – ISBN 978–3–540–66712–4, 416-429

[Ock03] Ocke, Stefan: *Entwurf und Implementation eines metamodellbasierten OCL-Compilers*, Technische Universität Dresden, Diplomarbeit, June 2003

[OMG12] Object Management Group (OMG): *Object Constraint Language.* Version: 2.3.1, January 2012. `http://www.omg.org/spec/OCL/2.3/`

[OSV10] Odersky, M. ; Spoon, L. ; Venners, B.: *Programming in Scala, 2/e.* Artima Press, 2010 (Artima Series). `http://books.google.de/books?id=ZNo8cAAACAAJ`. – ISBN 9780981531649

[Pre10] Press, Oxford U.: *Oxford Dictionaries.* `http://oxforddictionaries.com/definition/english/debug?q=debugging`. Version: April 2010

[Sch98] Schmidt, Alexander: *Abbildung OCL auf SQL*, Technische Universität Dresden, Diplomarbeit, April 1998

[SSJ+03] Shlyakhter, I. ; Seater, R. ; Jackson, D. ; Sridharan, M. ; Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003. – ISSN 1938–4300, S. 94 – 105

[Thi08] THIEME, Nils: *Reengineering des OCL2-Parsers*, Technische Universität Dresden, Diplomarbeit, January 2008

[URL12a] Eclipse Foundation: *Eclipse Documentation for Debug Model*. `http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/core/model/package-summary.html#package_description`. Version: 2012

[URL12b] Eclipse Foundation: *Eclipse Documentation for Interface IBreakpoint*. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fdebug%2Fcore%2Fmodel%2FIBreakpoint.html`. Version: March 2012

[URL12c] Eclipse Foundation: *Eclipse Documentation for Interface IDebugTarget*. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fdebug%2Fcore%2Fmodel%2FIDebugTarget.html`. Version: March 2012

[URL12d] Free Software Foundation: *The GNU Project Debugger*. `http://www.gnu.org/software/gdb/`. Version: March 2012

[URL13a] *Website of MDT/OCL*. Project website. `http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl`. Version: March 2013

[URL13b] *Website of Systematic Quality Assessment of Models Website*. Project website. `http://squam.info/?p=142`. Version: March 2013

[URL13c] *Website of the Dresden OCL Toolkit*. Project website. `http://www.dresden-ocl.org/`. Version: March 2013

[Wie00] WIEBICKE, Ralf: *Utility Support for Checking OCL Business Rules in Java Programs*, Technische Universität Dresden, Diplomarbeit, December 2000

[Wil09] WILKE, Claas: *Java Code Generation for Dresden OCL2 for Eclipse*, Technische Universität Dresden, Diplomarbeit, February 2009

[WTFS12] WILKE, Claas ; THIELE, Michael ; FREITAG, Björn ; SCHÜTZE, Lars: *Dresden OCL Manual*. `http://141.76.65.213/dresdenocl_updatesite/manual.pdf`. Version: March 2012

[WTW10] WILKE, Claas ; THIELE, Michael ; WENDE, Christian: Extending Variability for OCL Interpretation. Version: 2010. `http://dx.doi.org/10.1007/978-3-642-16145-2_25`. In: PETRIU, DorinaC. (Hrsg.) ; ROUQUETTE, Nicolas (Hrsg.) ; HAUGEN, Øystein (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 6394. Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–16144–5, 361-375

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, March 26, 2013