RWTH Aachen University
Software Engineering Group

Improvements to OCL Implementation within the Monti-Core workbench

Seminar Paper

presented by

Mehlan, Ferdinand

1st Examiner: Prof. Dr. B. Rumpe

Advisor: Arvid Butting ,M.Sc.

The present work was submitted to the Chair of Software Engineering

Aachen, Januar 19, 2018

Eidesstattliche Versicherung

Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Abstract

The object constraint language (OCL) is a domain specific language (DSL) for expressing logical constraints and evaluating them on an application domain. MontiCore is a language workbench, which can realize and generate implementation of such textual DSLs. Amongst other languages, OCL/P, a specification of OCL, with syntax similar to java, was built with MontiCore. Two main improvements on this OCL implementation have been made alongside this paper and serve as deliverables. One on the back-end-side, improving structure and extensibility, and a second one adding additional semantical sensitivity, improving on the user-experience.
The first contribution includes an overhaul of the OCL abstract grammar, cleaning up old artifacts and using new features of a new MontiCore version. Thus enabling easier maintainability and extensibility at grammar-level, better performance at generation-level and a cleaner workbench for downstream tools.
The second contribution: OCL is always used in combination with another DSL, to get some statement on a problem domain. E.g. in the UML theory OCL can navigate along classdiagramm (CD) relations to build constraints. Beforehand the OCL implementation would not consider semantical correctness of CD navigation in OCL constraints. The implementation now crosschecks such relations in constraints to all imported CD after parsing level.
To present these and other features, the OCL parsing tool has been integrated into an easily usable web presentation, where users can review and checkout the presented work.

Contents

Chapter 1

Introduction

The Object Constraint Language (OCL) is a language for defining of invariants on UML models. They are used to concretize behaviour of such UML models. An invariant is a constraint which should be true for the life-time of an object. For the formal description of behaviour, OCL can use information of classdiagrams to describe behaviour. That inludes navigation on class associations. This means, that a constraint is formulated on the level of classes, but its semantics is applied on the level of objects. Listing 1.1 shows an example invariant. It constrains the behaviour of an auction object, that the start time should come before the end time. This constraints can be used to describe correct behaviour.

```
context Auction a inv:
  a.startTime.lessThan(a.closingTime)
```

Listing 1.1: An example OCL constraint.

OCL implementations should follow these principles. Interesting for this paper, is if the navigation on classdiagram attributes is integrated into OCL languages. E.g. Eclipse OCL can cross-check navigation on CD. Our in house implementation OCL/P did not yet, however this paper will present type-checking as a new feature.

Chapter 2

Grammar Improvements

MontiCore is a language workbench, which provides mechanisms to extend and compose textual DSLs. Amongst other languages Unified Modeling Language for Programming (UML/P), a variant of UML, the languages and tools were built with MontiCore. That includes OCL/P [Rum12] , the UML/P specification of OCL and a DSL for classdiagramms (CD). In the following paragraphs, when mentioning OCL or CD, the P specification and its MontiCore realization is meant.
MontiCore features a specification of a DSL, by using an abstract grammar to describe the syntax and generates parser and an abstract syntax tree from it, which can be later used to built language specific tools. The contribution in this chapter will focus on the grammar and its correct and optimized implemenation of the OCL specification[OCL].

2.1 OCL Grammar

OCL is a DSL for expressing logical constraints. Listing 2.1 shows an example OCL file, while omitting the actual expressions, only showing what head information to the constraints are needed. An OCL file consist of an (optional) `package` and `imports`, followed by the `ocl` block. The `ocl` block groups one or multiple invariants, with an (optional) `context`, in which the actual logic expressions built the constraint.

```
package example.ocl;

import example.cd.Auction;

ocl myOCLDefs {
  inv:
    // ocl expressions here

  context Auction a, b inv:
    // ocl expressions here

  context Auction inv:
    // ocl expressions here
}
```

Listing 2.1: An example OCL file showing various constraints with expressions omitted.

Listing 2.2 shows an excerpt of the OCL grammar and how the syntax of the parts mentioned above are realized textual . Additionally each constraint can be built out of (nested) OCL expressions, e.g. simple math-expressions like `2 == 4 + 4`, which would consist of an equals and a plus-expression. More detailed explanations to MontiCore grammars can be found here [GKR+08][KRV08][KRV10] and for the OCL specification here [Rum12][OCL].

```
grammar OCL {

  CompilationUnit =
    ("package" package:(Name || ".")+ ";")?
    (ImportStatement)*
    OCLFile;

  OCLFile =
        "ocl"
        fileName:Name "{"
        OCLConstraint*
        "}"
        ;

  interface OCLConstraint;

  // (...)
}
```

Listing 2.2: An excerpt of the MC4 grammar for OCL.

2.2 Miscellaneous Improvements

Parsing empty strings: There were several syntax rules, which could parse empty strings. While this would mostly not break the grammar, it can hinder the parsing performance, since it causes trace-backs, when it first matches the empty string with the rule, but later has to go back in the tree because no matching rules can be found at that path. As a rule of thumb, one should parse as little as possible, while being in accordance with the specification and without making the grammar overly complex.
Listing 2.3 shows a rule, which should parse different types of writing constraint definitions. E.g. `Auction`, `Auction a`, `Auction a in (...)` or `a in (...)`. However it also matches the empty string, because everything is optional. By splitting the rule into two cases and being more verbose, as shown in Listing 2.4, this problem can be avoided. Now each half has a non optional part.

```
grammar OCL {

  OCLContextDefinition =
    Type? (Name ("in" expression:OCLExpression)?)?
    ;

  // (...)
}
```

Listing 2.3: **Before:** Showing a grammar rule which parses an empty string.

```
grammar OCL {

  OCLContextDefinition =
    Type Name?
    |
    Type? Name ("in" expression:OCLExpression)?
    ;

  // (...)
}
```

Listing 2.4: **After:** Now not parsing an empty string anymore.

Disparities to the specification: Some disparities between the specification and the grammar showed up. E.g. it should be possible to define multiple variables with the same type separated by comma like this: `Auction a1,a2,a3`. However this was not possible with the version of Listing 2.4. This was fixed as shown in Listing 2.5.

```
grammar OCL {

  OCLContextDefinition =
    Type (Name || ",")*
    |
    Type? (Name || ",")+ ("in" expression:OCLExpression)?
    ;

  // (...)
}
```

Listing 2.5: Now can parse multiple variables separated by comma.

2.3 Expression Extensions

The OCL invariants are built from statements and expressions, which can be nested within other expressions. Looking at infix expressions, this led to some limitations when generating with MontiCore. The following examples show how a grammar for the infix expressions for `+ - * / %` was realized with each MontiCore version and what change was made in the last version.

MontiCore 3.x: This version did not support left-recursive rules. This means when designing the expression rules, to realize a priority between different expressions they have to be connected in the right order and operators of the same priority belong into the same node. In the example at Listing 2.6 this means `ASTPlusMinusExpr` makes one node and is dependent on `ASTMultDivModExpr`. And `ASTMultDivModExpr` itself is dependent on the next higher expression and so on.
Cons: Expression nodes are dependent from each other, this hinders navigation and it is hard to introduce new expressions in the middle, since this changes the AST and breaks tools.
Pros: `ASTPlusMinusExpr` has a different AST then `ASTMultDivModExpr`, this is good for tools working on different expressions.

```
grammar OCL {

  PlusMinusExpr implements Expression =
    left:PlusMinusExpr operator:["+" | "-"] right:PlusMinusExpr
    |
    ("(" paren: PlusMinusExpr ")" | nextHigherPrec: MultDivModExpr)
    ;

  MultDivModExpr implements Expression =
    left:MultDivModExpr operator:["*" | "/" | "%"] right:MultDivModExpr
    |
    ("(" paren: MultDivModExpr ")"  | nextHigherPrec:LogicalORExpr)
    ;
}
```

Listing 2.6: **MontiCore 3.x:** Infix-Expressions.

MontiCore 4.0: This version supports direct left-recursive rules, that means the rules can be designed differently as shown in Listing 2.7.
Pros: The grammar is more cleanly readable, by using left-recursion.
Cons: All expressions have to be in one rule, thus are not extensible at all and additionally this only generates one ASTNode to work with. Now tools can not differentiate between PlusMinus and MultDivMod, or use the visitor pattern to work with them.

```
grammar OCL {

  Expression =
    Expression operator:["*" | "/" | "%"] Expression
    |
    Expression operator:["+" | "-"] Expression;
}
```

Listing 2.7: **MontiCore 4.0:** Infix-Expressions.

MontiCore 4.5: The latest version introduces some useful features, also indirect left-recursive rules and rule priorities (since 4.5.3).
Pros: The grammar is clean and easy to read, while using left-recursion. The priorities should be set with some space in between, (e.g. in steps of 10), to allow adding new rules later. Each operator gets its own node, allowing a clean visitor pattern on each of them for tools.

```
grammar OCL {

    interface Expression;

    PlusExpression implements Expression<20> =
      Expression "+" Expression;

    MinusExpression implements Expression<20> =
      Expression "-" Expression;

    MultExpression implements Expression<30> =
      Expression "*" Expression;

    DivExpression implements Expression<30> =
      Expression "/" Expression;
}
```

Listing 2.8: **MontiCore 4.5:** Infix-Expressions.

Chapter 3

Semantic: Types in OCL

OCL is a type-less language, i.e. it is possible to define variables, but giving a type is optional and when used further correct assignment and comparison is not checked. OCL is often used with CD together which provides types and navigation across relations between classes. The idea is to import one or more CD models to an OCL file and automatically infer types and check for consistency using information from the CD models.

3.1 Concept

Figure 3.1 shows an example CD. It includes various classes with attributes and associations with cardinalities and optional role and association names.

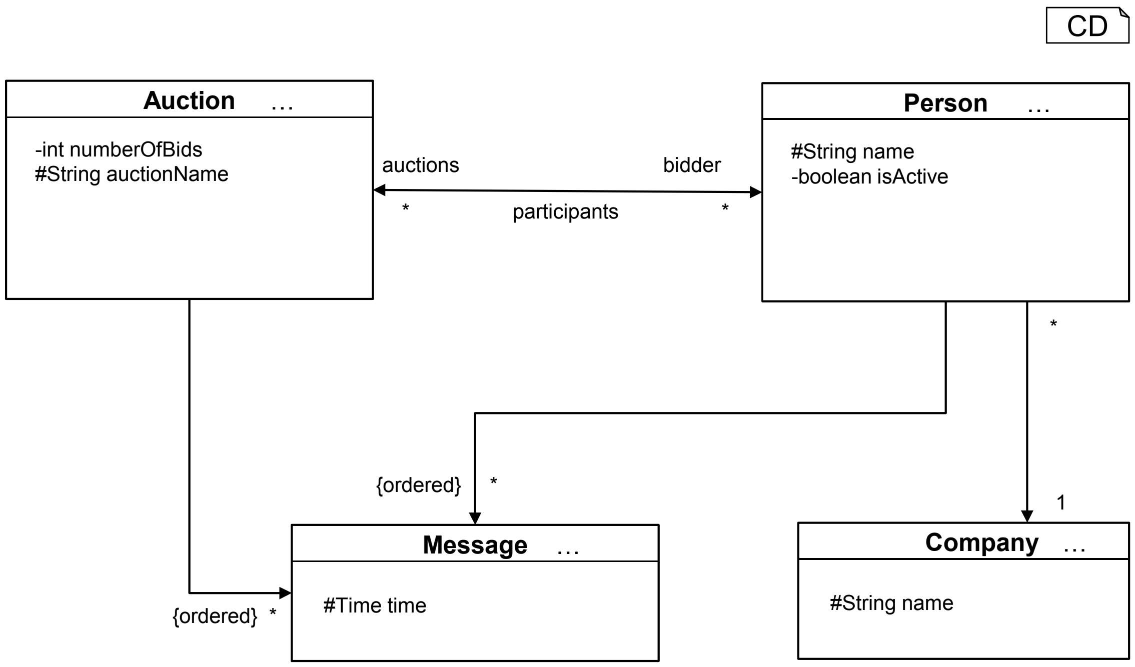


Figure 3.1: This is an example CD model.

The example in Listing 3.1 then imports the classdiagram and uses the types in the OCL constraint. Several variables are declared: `b = a.bidder` where `b` is a `Set<Person>`. Next `m = b.message` the variable is of the type `List<Message>` using navigation and implicit flattening on the CD. Tools using OCL could need this information, especially generators generating java-code from OCL [Meh17], since in java correct typing is needed. Additionally CD or OCL files can get quite big, where it isn't obvious if correct navigation is applied. Also there are many different ways a classdiagram can be navigated, which will be explored in further examples. All this makes it a useful addition to implement some sort of type inferring mechanism after parsing the OCL model.

```
package example.ocl;

import example.cd.Auction;

ocl myOCLDefs {

  context Auction a inv:
    let
      b = a.bidder;  // Set<Person>
      m = b.message;    // List<Message>
      t = {mess.time | mess in m}; // Collection<Time>
    in
      b == this.message

}
```

Listing 3.1: This OCL constraint contains several variable declarations without a type given.

Now it is from interest to shortly look into, what kinds of navigation on CD are possible and need to be considered. In general OCL can navigate across CD in specification mode, meaning direction of associations and visibility of attributes does not need to be considered. OCL can access them all. Listing 3.2 shows multiple ways of navigating across the same association.
First: `a.participants` uses the association name to get a `Set<Person>`. The association name also can be used from both sides, i.e. `Person.participants` to navigate to Auction.
Second: `a.bidder` uses a role name to get the same set. The role name here is specific to one side, giving a special name to the target. For the other side one would need the other role name: `Person.auctions`.
Third: The last possible way is to get the target by using its lower-case name: `a.person`. This is always possible and useful when no association name or role name is specified.
Or the same in the fourth example, but showing the specification mode and its possible navigation against association direction, `c.person` is a proper call and gives a `Set<Person>`.
After that, all these examples navigate from `Set<Person>` to the person attribute `name`. This is possible because OCL supports implicit flattening, this would finally return a `Set<String>`. A more detailed specification of implicit flattening of container types can be found here [OCL].

```
package example.ocl;

import example.cd.Auction;

ocl myOCLDefs {

  context Auction a inv:
    a.participants.name != Set{}

  context Auction a inv:
    a.bidder.name != Set{}

  context Auction a inv:
    a.person.name != Set{}

  context Company c inv:
    c.person.name != Set{}

}
```

Listing 3.2: Shows different possible navigation ways across an association

Listing 3.2 shows how the `this` variable is automatically set, if only one variable is set in the context, or only one type without variable is set in the context, then a variable named `this` is implicitly declared and can be used in the constraint.
The third constraint in that example does not have any context at all, but shows another alternative. `Auction.participants`, where `Auction` is the short-form for `Auction.getAllInstances()` and thus gives a set of auctions to work on.

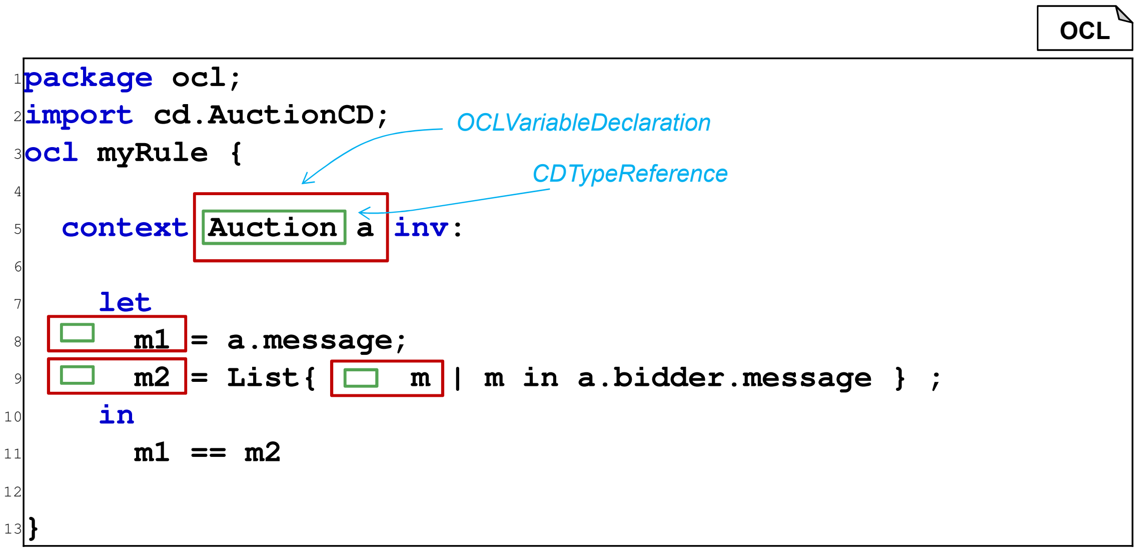
```
package example.ocl;

import example.cd.Auction;

ocl myOCLDefs {

  context Auction a inv:
    this.participants.size > 0

  context Auction inv:
    this.participants.size > 0

  inv:
    Auction.participants.size > 0

}
```

Listing 3.3: Shows examples of using `this` or Class.getAllInstances()

In summary OCL can navigate in specification mode across classdiagrams using properties of associations and classes. Additionally there are some implicit declarations of variables and flattening of containers. The task now is integrating these semantic implications into the implementation and warn the user of semantically false navigations to improve constraint quality.

3.2 Implementation

First lets look at how MontiCore generates DSL and their tools. There are two main concepts, which are worked with here, the abstract syntax tree (AST) and the symboltable. In depth explanations can be found here [OCL].
As already shown in chapter 2, MontiCore takes a grammar and generates the language parser and other tools. This works by parsing the grammar and generating the AST from it, where each grammar rule gets an ASTNode. Also a parsing tool is generated, which can take a string of an OCL model and parse it, meaning generating the AST from it. An ASTNode then is an object consisting of parsed information, like names and other sub-nodes. E.g. looking at the `PlusExpression` rule in Listing 2.8 the string `"2 + 4 * 3"` would parse to an ASTPlusExpressionNode, which has two data entries, some sort of primary AST node for `2` and an ASTMultExpression for the `4 * 3`. This is all automated by MontiCore and is useful because it also enables using a visitor pattern on each AST node, but until now mostly concerns syntactic stuff.
Alongside the AST the symboltable can be built. It supports features of symbols, scopes



```
package ocl;
import cd.AuctionCD;
ocl myRule {

  context Auction a inv:

    let
        m1 = a.message;
        m2 = List{    m | m in a.bidder.message } ;
    in
      m1 == m2

}
```

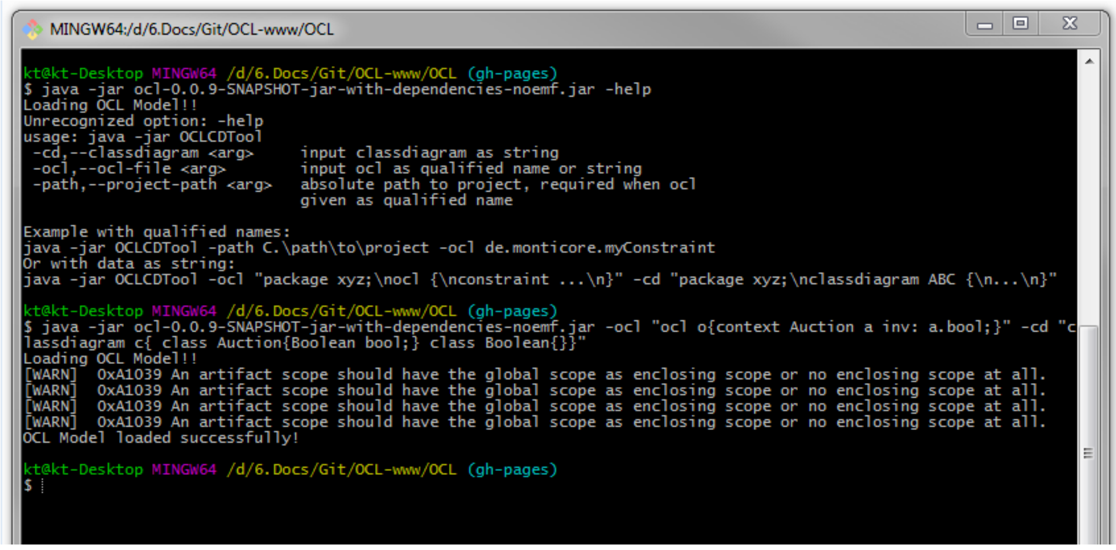
Figure 3.2: An example OCL constraint. Red marks `OCLVariableDeclarations`, green `CDTypeReferences`.

and shadowing, and resolving these symbols in their scopes. For the task here interesting, as each variable defined in OCL should be represented as an OCLVariableSymbol, which holds data about its name and type. This is useful e.g. for a java generator, which then just resolves the variable name as an OCLVariableSymbol and gets the type. However the problem in OCL was that the type was not always specified, so it has to be inferred while building the symboltable.
Figure 3.2 holds an example OCL constraint with various variables. Red boxes mark where OCL variables are defined and OCLVariableSymbols need to be created. They include type and name (the scope depends on the AST context). Green boxes mark where

the type should be. However only the first variable has its type declared, the others have to be inferred.
This is done using a visitor pattern on the defining expression of the variable. E.g. `m1 = a.message` where `a.message` is the expression defining the type of `m1`. In this case there is a qualified name expressing navigation. The Inferring visitor then recursively infers the types of the names.

- First it resolves a as OCLVariableSymbol and get its CDTypeReference, which in this case is a CD class named Auction.
- Second it tries to resolve message as an attribute or association on that CD class. This is possible since with importing the AuctionCD, all of this model was parsed and all elements built into the symboltable automatically. The OCL symboltable creator uses this now to built its own symboltable.

3.3 Running Example



```
MINGW64:/d/6.Docs/Git/OCL-www/OCL

kt@kt-Desktop MINGW64 /d/6.Docs/Git/OCL-www/OCL (gh-pages)
$ java -jar ocl-0.0.9-SNAPSHOT-jar-with-dependencies-noemf.jar -help
Loading OCL Model!!
Unrecognized option: -help
usage: java -jar OCLCDTool
 -cd,--classdiagram <arg>     input classdiagram as string
 -ocl,--ocl-file <arg>        input ocl as qualified name or string
 -path,--project-path <arg>   absolute path to project, required when ocl
                              given as qualified name

Example with qualified names:
java -jar OCLCDTool -path C.\path\to\project -ocl de.monticore.myConstraint
Or with data as string:
java -jar OCLCDTool -ocl "package xyz;\nocl {\nconstraint ...\n}" -cd "package xyz;\nclassdiagram ABC {\n...\n}"

kt@kt-Desktop MINGW64 /d/6.Docs/Git/OCL-www/OCL (gh-pages)
$ java -jar ocl-0.0.9-SNAPSHOT-jar-with-dependencies-noemf.jar -ocl "ocl o{context Auction a inv: a.bool;}" -cd "c
lassdiagram c{ class Auction{Boolean bool;} class Boolean{}}"
Loading OCL Model!!
[WARN]  0xA1039 An artifact scope should have the global scope as enclosing scope or no enclosing scope at all.
[WARN]  0xA1039 An artifact scope should have the global scope as enclosing scope or no enclosing scope at all.
[WARN]  0xA1039 An artifact scope should have the global scope as enclosing scope or no enclosing scope at all.
[WARN]  0xA1039 An artifact scope should have the global scope as enclosing scope or no enclosing scope at all.
OCL Model loaded successfully!

kt@kt-Desktop MINGW64 /d/6.Docs/Git/OCL-www/OCL (gh-pages)
$
```

Figure 3.3: The OCL parser integrated into a command-line tool.

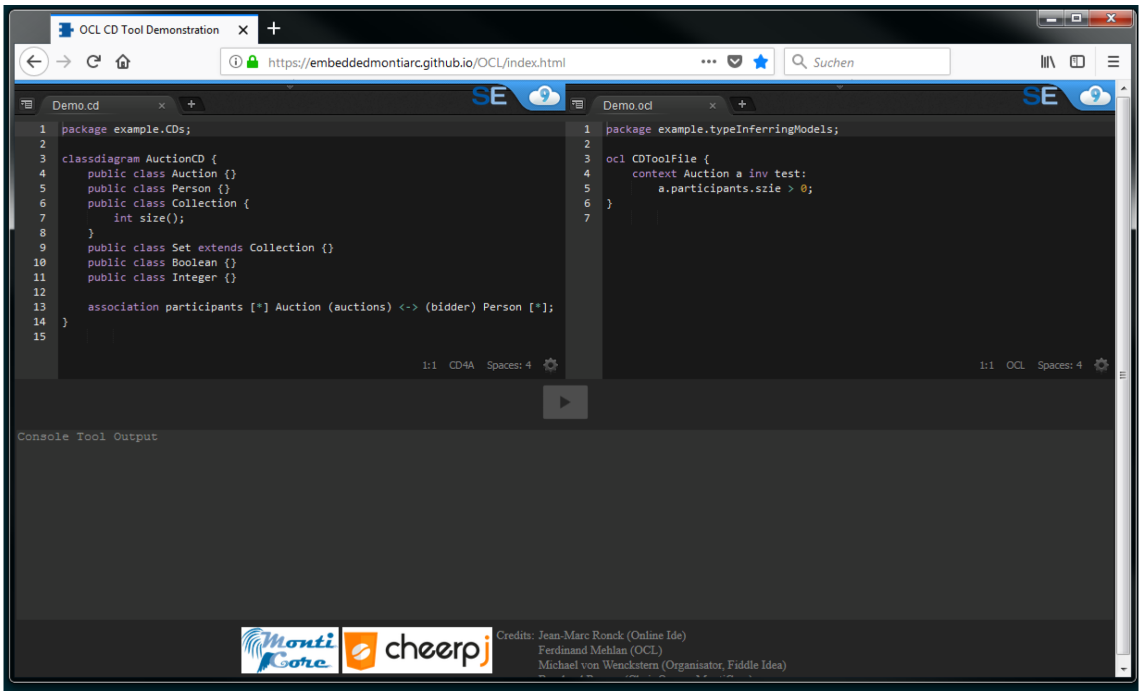
While the implementation was fairly successful, these features of automatically loading CD and cross-checking types are fairly back-end in their nature. To get a better view of it and easier access to test examples, a command-line tool `OCLCDTool` functions as an user-interface to easily check inputs. Figure 3.3 shows the CLI and how it is used. The tool and some examples can be found here:
`https://github.com/EmbeddedMontiArc/OCL/tree/master/OCLCDTool`
The tool, when used on a file system, is called with two options:

- `-path` : which takes the absolute path to the parent folder of the project, where the CD and OCL files are present.
- `-ocl` : which takes a qualified name to the ocl file.

If the file system is not accessible, like in one of our use-cases, the tool can also be called directly with strings of the models:

- `-cd` : which takes the CD model string.
- `-ocl` : which takes the OCL model string.

The CLI is an initial step, but not always convenient for users. Additionally there are often problems with wrong java versions and falsely set environment variables. A novel trend is software in the web, while using a browser to access and input it. The goal for a better presentation would be to built a simple front-end which then feeds the CLI. This can be easily realized with JavaScript and some dynamic web elements. The tool however is written and built in Java and because of performance and security reasons it is generally thought of a bad practice, running a JVM and Java code in a browser.
The solution here would be to automatically translate the Java CLI tool to JavaScript


OCL CD Tool Demonstration
https://embeddedmontiarc.github.io/OCL/index.html
Suchen
Demo.cd
Demo.ocl

```
package example.CDs;

classdiagram AuctionCD {
    public class Auction {}
    public class Person {}
    public class Collection {
        int size();
    }
    public class Set extends Collection {}
    public class Boolean {}
    public class Integer {}

    association participants [*] Auction (auctions) <-> (bidder) Person [*];
}
```

```
package example.typeInferringModels;

ocl CDToolFile {
    context Auction a inv test:
        a.participants.szie > 0;
}
```

Credits: Jean-Marc Ronck (Online Ide)
Ferdinand Mehlan (OCL)
Michael von Wenckstern (Organisator, Fiddle Idea)

Figure 3.4: Showing a browser driven GUI for the OCL tool

and integrate that into the web page. For this a python tool named **cheerpj** by the company learningtech[CPJ] was used. It emulates the java standard libraries in JS and transpilles your java code, given as packed jar. This process was not flawless at first. Executing the script on windows using python3, the process would fail, because of some UTF charset problem. However using a current linux distribution would solve this issue. Additionally working together with learningtech we could find a bug in the emulation of a java standard library, which was impeding the execution of the CLI. In the end the process was successful and a JavaScript version of the tool is integrated into an example website. Additionally the online-ide by [Ron17] was integrated into the site providing an overall pleasant user experience. The examples from the beginning of the paper can be tested there.

The tool can be found at:
`https://embeddedmontiarc.github.io/OCL/index.html`
It works by editing or inputting a classdiagram on the left editor field, then one or more OCL constraints in the right editor field. When pressing the play button, both models are parsed, then the types in the OCL file are cross checked to the classdiagram. Any errors from any step, meaning parsing errors or miss matched types will be printed out at the bottom console window.

Chapter 4

Conclusion

This paper presented improvements on the implementation of OCL/P. The improvements on th grammar promised an increase on the parser-performance and a cleaner workbench to develop tools. The parser performance was not measured in exact test-cases, because this was not the main objective, but on a subjective level, parsing-test were finishing faster. The cleaner grammar however, using the new MontiCore features to specify expressions lead to having a single AST for a single expression. This was perceptible, when developing the type-checking tool. Using the visitor pattern, handeling each ASTNode became clearer und the code got easier to comprehend.
The second contribution was about cross-checking CD types used in OCL constraints and if the navigation across CD types is not illegal according to the specification mode of OCL/P. Using OCL it mostly makes sense, to use it within a context domain. And with UML/P OCL/P is mentioned together with CD. However until now the MontiCore implementation lacked a connection between the two and relied on correct user inputs. The extension of the OCL/P implementation with automatic type checking fills the gap here.
This was beneficial to another in house project and substantially raised the quality there. The project had a generated OCL file with 200+ constraints and a related CD. The type-checking mechanism managed to find multiple inconsistencies there, which then could be easily fixed.
The translation of the OCL implementation to JavaScript and the integration into an online interface was so far successful and seems promising to easier provide examples and language features to the outside world. This can be probably used for presentation examples for research staff or for teaching purposes.

Bibliography

[CPJ] CheerpJ a Java compiler for Web applications: `https://www.leaningtech.com/cheerpj/`.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[Meh17] Ferdinand Mehlan. Verification of non-functional Properties on Component and Connector Models. Bachelor Thesis, RWTH Aachen University, 2017.

[OCL] OCL/P specification in the chapter OCL of Modellierung mit UML: `http://www.mbse.se-rwth.de/book1/index.php?c=chapter3`.

[Ron17] Jean-Marc Ronck. Creation of a Multi-User Online IDE for Domain-Specific Languages. Bachelor Thesis, RWTH Aachen University, 2017.

[Rum12] Bernhard Rumpe. *Modellierung mit UML*. 2012.