COOL Project Report

IKT445 - Generative Programming

Johnny Sognnes

Halvor Songøygard Smørvik

*

Department of Information and Communication Technology

Faculty of Engineering and Science

University of Agder, 2017

Contents

Chapter 1

Abstract

MPS was used to implement the COOL language based on the COOL reference manual. Structs, editor, contraints and model to model translation was used to generate java code from test programs in COOL. The language is not fully implemented, but some translations work.

Chapter 2

Introduction

COOL is a small object oriented language made for compiler courses. In this assignment the COOL language is translated to base language which is very similar to java, using model to model transformations in MPS. The language specification is the COOL reference manual[1]. Several code examples is used to test translations from COOL to base language/Java. Our language is built upon a past student project, where we also got their documentation for their work. Therefore there are some elements which is not documented in this report, but still is in the repository.

The version used is MPS 2.2. The project is uploaded at the cool_mps repository in Bitbucket. The latest commit at the time of writing was 87132ef70af at 16.11.2017.

Chapter 3

Language

"Cool is based on Sather164, which is itself based on the language Sather"[1]. COOL have may similarities with java. It have classes and methods, but is a bit restricted on creation of variables. Generally it has less functions than java.

Chapter 4

Solution

4.1 Structure

The concepts that are used in MPS are based on the syntax page in the COOL reference manual[1]. This was already included in MPS in the previous project. One concept was added, but this is probably possible to combine with an existing concept. The structure of the language, including the concepts used for the examples is added as an appendix to this PDF.

4.2 Textual Syntax

The textual syntax was added by using the editor. The form of each concept is stated in the manual, and could be made in the editor with similar notation. The syntax was already written in the MPS editor at the start, but a syntax for dispatch, and separators to method were added to this.

4.3 Constraints

Following constraints have been implemented in MPS.

- **Upper- and lowercase letters**

  Every class must start with a uppercase letter. Every method and attribute must start with lowercase letter. If this is not true then property "-> is valid" is set to false in their respective constraints.

- **Digits**

  Every class, attribute and method cannot start with a digit. This is also done in "is valid"

- **Keywords**

  Keywords (as defined by the COOL manual) cannot be used as identifiers or types. One cannot write loop : Int, for example.

- **Empty name**

  The length of a name must be 1 letter or more.

- **Legal alphabet**

  When making identifiers or types, the letters one can use is now limited to the alphabet 0-9a-zA-Z_. This is implemented by checking a regex for `\w*`, for all characters written by the user in method, attribute and class. We can't find the exclusion of characters in the COOL reference manual, but it makes sense, looking at what's standard in other programming languages.

- **Redefining**

  Classes may not be redefined. No method name may be defined multiple times in a class, and no attribute name may be defined multiple times in a class. This is implemented in the "is valid" part of the property constraints for class and method, by limiting size of objects of the same concept with the same name to 1. This constraint is in the manual, therefore included.

4.4 Semantics

We use model-to-model transformation in MPS, which translate each COOL concept into Baselanguage/Java using templates. Following concepts have transformations in MPS.

- **Program**
- **Class**
- **Method**
- **Attribute**
- **Dispatch**

- **New**
- **Constants**
- **Block and block expression**

4.5 We are proud of/worked well

The constraints implemented worked very well, as they are both logical and easy to test. Some of the transformations also worker well, examples of these can be seen in the next chapter.

4.6 Not implemented

Following points are not implemented for COOL.

- Create the rest of the reduction rules
- Sort out build errors, specifically for type node references in the generator
- If we wish to get i.e. '(new IO).out.string', we need to put parentheses around the 'new' expression without getting them
- Include 'self'
- Implement inheritance properly and get classes defined outside a program to generate.
- Make a reference to BaseAttribute. - We could get items in classes we did not inherit from.
- Make transformed methods return a value, and get arguments in the transformations.

Chapter 5

Examples

The following images shows a range of examples of transformations from COOL to Java. The COOL images also shows how the textual syntax is displayed in the code.

Program, Class and method is present in all of them, because it doesn't make any sense making a program in COOL without those.

```
class Main <no inherits> {
  main (  ) : Int {
    {
      abort (  ) ;
      1 ;
    }
  };
};
```

(a) COOL

```
public class Transf_Ex_Abort {
  public static class Main {
    public static void main() {
      {
        abort();
        1;
      }
    }
  }
}
```

(b) Generated

Figure 5.1: Dispatch with abort, block and integer constant

```
class Main <no inherits> {
  main (  ) : Object {
    false
  };
};
```

(a) COOL

```
public class Transf_Ex_Bool {
  public static class Main {
    public static void main() {false
    }
  }
}
```

(b) Generated

Figure 5.2: Boolean constant

```
class Main inherits SELF_TYPE {
  main (  ) : <no type> {
    new Main
  };
};
```

(a) COOL

```
public class Transf_Ex_New {
  public static class Main {
    public static void main() {
      new Transf_Ex_New.Main();
    }
  }
}
```

(b) Generated

Figure 5.3: Usage of new

```
class Main inherits IO {
  str : String  ;
  main (  ) : <no type> {
    out_string (  )
  };
  stringtest (  ) : <no type> {
    "this is a string"
  };
};
```

(a) COOL

```
import COOL.types.baseclasses.String;

public class Transf_Ex_String_Out {
  public static class Main {
    private static String str;
    public static void main() {out_string()
    }
    public static void stringtest() {"this is a string"
    }
  }
}
```

(b) Generated

Figure 5.4: Dispatch with IO and string constant

Chapter 6

Discussion

The transformations have been done by using a base language generator. Reduction rules for almost half of the concepts and a mapping label for Class have been created.

The work was done by using the methods we learned in class and in review meetings. The rest of the work was done by trial and error, because there are no examples (that we found) applicable for this type of project.

Unfortunately, we did not have much time at the end of of the project to look through the unsolved challenges in MPS. As the project is now, it is not usable in terms of running or translating all of what COOL should be able to do.

A proposed solution where the baseClasses were made into a library has been considered, and we could have tried doing this if we had more time and knowledge of the software.

In the beginning of the project, we chose to build upon the prior student project. This was because we tough this could help us save some time and to help us understand how a language is implemented in MPS. The structure they used has been modified, and have better documentation now, then what we got. For structure, the Interfaces where used instead of abstract classes, since they allready was included in the erlier student project.

Some examples have been included in the chapter before this one, showing that some aspects of our implementation of COOL works.

Future steps would be to work ous sevral bugs and implement everything mentioned in chapter 4.6

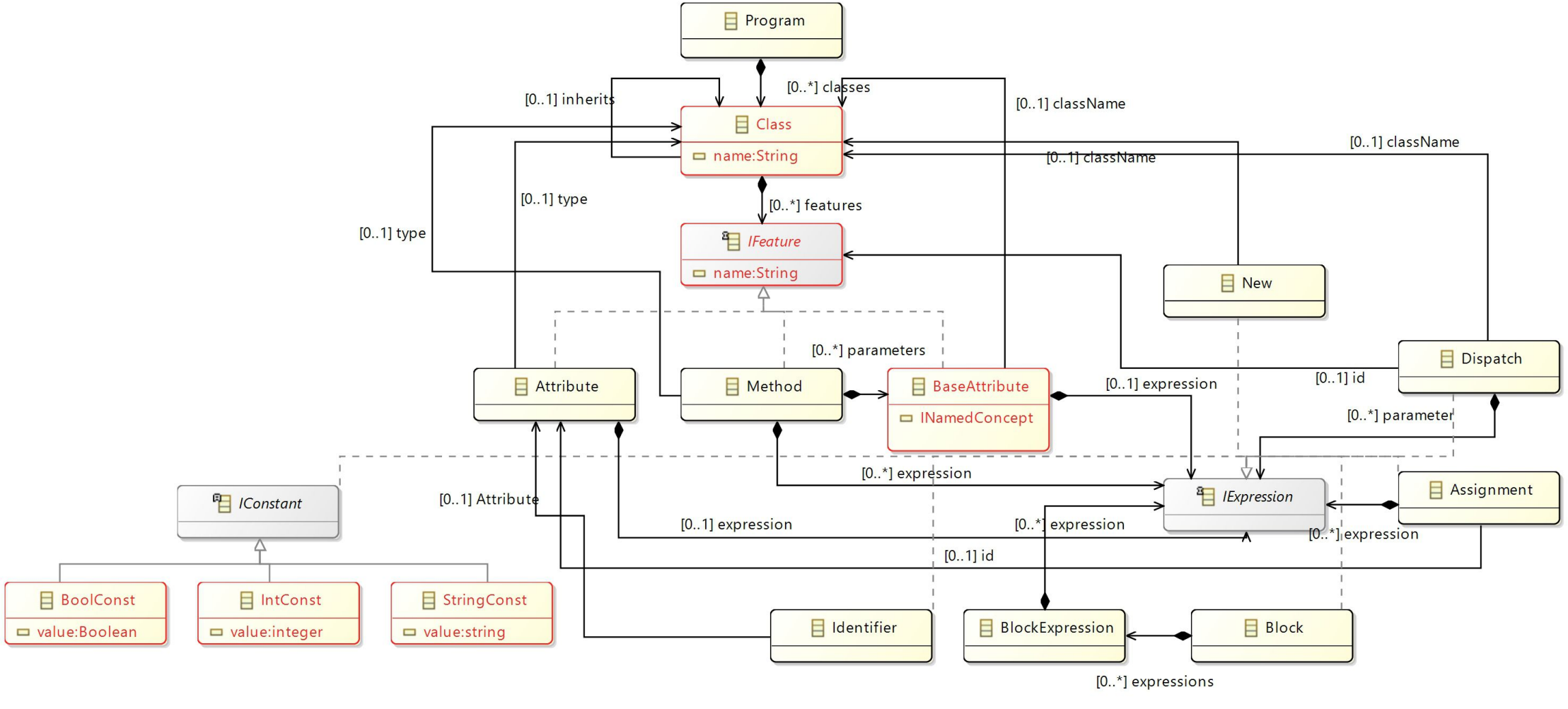
Chapter 7

Plan and reflection

This project was worked on by setting one to two days off every time a deadline of the mandetory assignments for the project was up. In the last days before turning this project raport in we gathered two days. Seeing how we didn't manage to make a complete COOL implementation, more time should have been given to this project, to get the expected result of a full implementation.

Bibliography

[1] *The Cool Reference Manual*. [Online]. Available: `https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf` (visited on 05/12/2017).


Program
[0..*] classes
[0..1] inherits
[0..1] className
Class
name:String
[0..1] className
[0..1] className
[0..1] type
[0..*] features
[0..1] type
IFeature
name:String
New
[0..*] parameters
Dispatch
[0..1] id
Attribute
Method
BaseAttribute
INamedConcept
[0..1] expression
[0..*] parameter
[0..*] expression
IConstant
[0..1] Attribute
IExpression
Assignment
[0..1] expression
[0..*] expression
[0..*] expression
[0..1] id
BoolConst
value:Boolean
IntConst
value:integer
StringConst
value:string
Identifier
BlockExpression
Block
[0..*] expressions