University of Agder Norway

COOL - Project report

Group: COOL
Fredrik Jørgensen Olsen, Kristoffer Lauritz Larsen, Anders Tellefsen
Git Repository: https://github.com/Zedyz/COOL-MPS

IKT445
Generative Programming

Grimstad, December 13, 2024

Contents

List of Figures

List of Tables

1 Introduction

The Classroom Object-Oriented Language (COOL) is a language designed to teach and learn object-oriented programming concepts, specifically for object-oriented programmers.

This project focuses on improving the current COOL implementation by addressing issues in the existing implementation within MPS [2], without modifying the language itself. The work strictly adheres to the specifications defined in the COOL manual [1] and the language structure described in Section 2.1. The primary goal is to address predefined issues, identify new ones, and implement tests to validate the concept editors within the implementation.

Specifically, EditorTests will be used to test the concept editors by simulating user interactions with the language to ensure that the COOL editor functions as expected. There will also be an effort to implement NodeTests to verify functionality at the node level, as well as Generation tests to ensure that the template declarations generate the expected Java code.

The report documents the process of addressing issues related to the current COOL implementation, both existing and newly identified, with a focus on providing thorough documentation through screenshots and detailed explanations. The project findings of current issues and proposed next steps, outlined in Section 4.3 and 4.4, should be considered particularly useful for future groups reviewing this report and continuing work on the COOL implementation.
The report is organized as follows:

- **Chapter 2: Language** – Provides an overview of the COOL language, detailing its purpose and existing implementation.
- **Chapter 3: Implemented solutions to issues** – Describes the implementation process of solution to current issues in the implementation, as well as describing tests.
- **Chapter 4: Discussion** – The report discusses the current implementation and the successes of this project, including working examples, while also outlining existing issues in the implementation and proposing future steps for improvement.
- **Chapter 5: Plan and Reflection** – Reviews the project planning and management, analyzing time allocation, milestones, and outcomes.
- **Appendix** – Includes supplementary materials.

2 Language

2.1 Overview of COOL Language

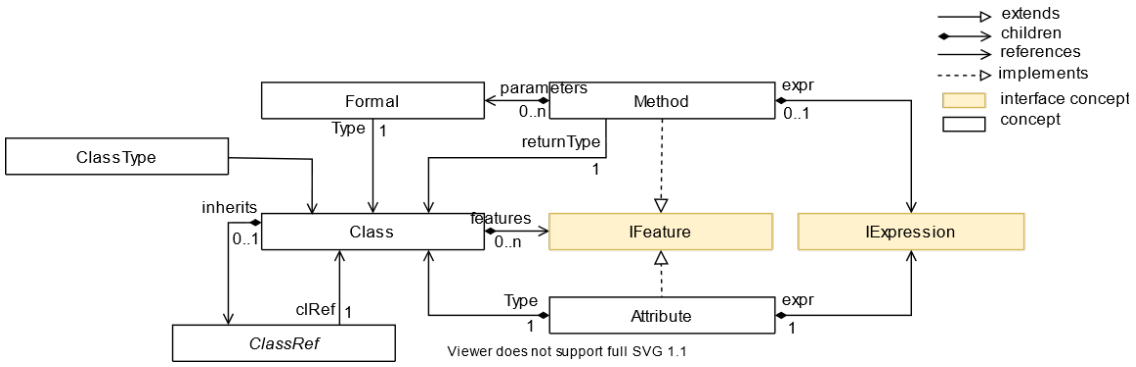

Figure 2.1: Abstract syntax tree (AST) of COOL, highlighting the hierarchical structure of classes, attributes, and methods.

The figure illustrates the foundational structure of the COOL language. At the top level, a Class node serves as the central concept. Each class may include multiple IFeature elements, which come in two principal forms: Attribute and Method. Attributes provide the data fields associated with a class, while Methods implement the functional behavior of that Class.

Within each Method, Formal parameters define the inputs required by the Method's logic. Methods also reference an IExpression, a placeholder for the executable code that implements their functionality. The ClassRef concept enables classes, Methods, and Attributes to reference each other consistently throughout the model. This reference mechanism allows complex type hierarchies and inter-class relationships to be represented explicitly.

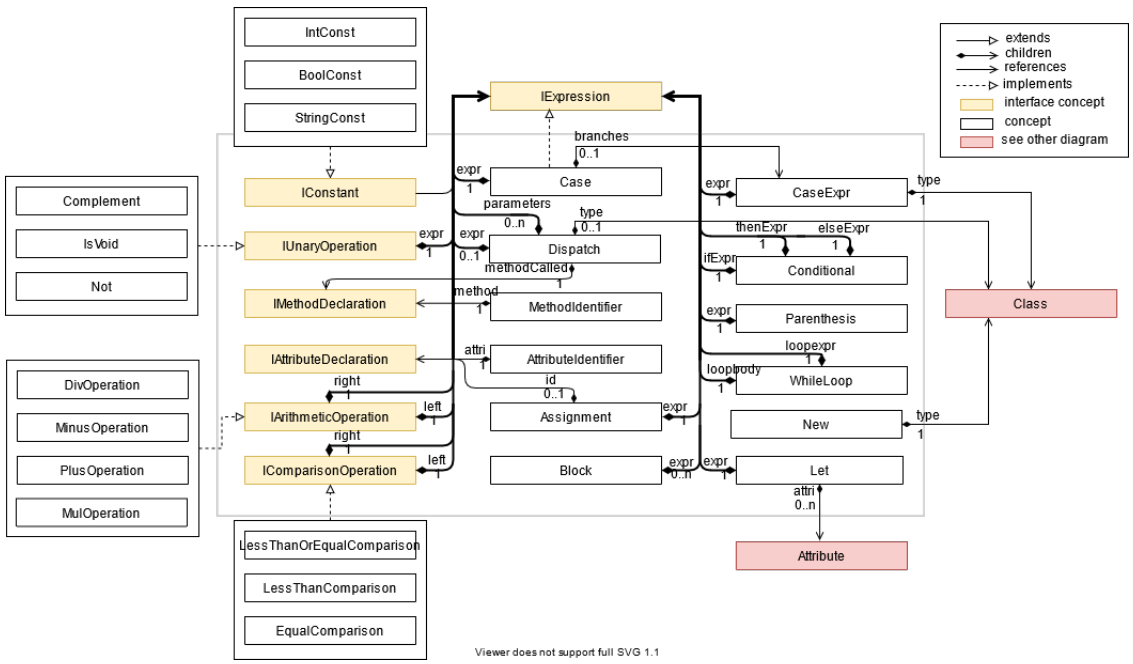

Figure 2.2: Extended structure illustrating the relationship between expressions, features, and their implementation through concepts such as interfaces.

The extended figure provides a detailed view of the internal structure of the COOL language's expression model. At its core is the IExpression concept, which serves as the basis for all constructs in

COOL. The model branches into various specialized expression types, each representing a specific aspect of the language's operational semantics.

IConstant expressions represent immutable values within the code, such as integers, booleans, and strings. Building on these basic values, IUnaryOperation and IArithmeticOperation nodes define the supported unary and binary arithmetic operations (e.g., negation, addition, multiplication) that can be applied to constants or other expressions. IComparisonOperation nodes specify relational checks like equality or less-than comparisons. IMethodDeclaration and IAttributeDeclaration nodes allow references to methods and attributes, enabling dynamic resolution of methods to be called or fields to be read. Dispatch nodes represent method calls, linking an IExpression to a corresponding method identifier and optional parameters. Assignment operations handle mutable state changes, binding new values to attributes or local variables.

Conditional nodes capture if-then-else logic, WhileLoop nodes provide iteration capabilities, and CaseExpr nodes introduce pattern-matching style branching. There are also abstractions like Block for grouping multiple sub-expressions, Let for local variable bindings, and New for object instantiation. COOL organizes all code into classes, where each class encapsulates attributes (variables) and methods (functions) that define the behavior of objects. It supports object-oriented features such as inheritance, polymorphism, and encapsulation. Classes in COOL form a single-inheritance hierarchy, with the 'Object' class serving as the root.

Characteristics of COOL:

- **Static Typing:** COOL enforces type safety at compile time, ensuring that procedures are applied only to data of the correct type, preventing runtime type errors and guaranteeing predictable program behavior.
- **Expression-Oriented Structure:** Most constructs in COOL are expressions, and every expression has a type and value, reinforcing the language's strict typing system.
- **Automatic Memory Management:** COOL uses garbage collection to automatically deallocate unused objects, removing the need for manual memory management.
- **Basic Classes:** The language includes predefined classes such as 'Object', 'Int', 'String', 'Bool', and 'IO', providing functionalities like input/output operations, arithmetic, and string manipulation.
- **Inheritance:** Classes can inherit from other classes, enabling the reuse of attributes and methods while supporting method overriding with restrictions to maintain type safety.

The COOL type system is designed to provide strong type safety while supporting object-oriented programming principles. It introduces the SELF_TYPE, which adapts to the class of the self object, ensuring that methods returning objects of the same class are correctly typed. This ensures that variables, attributes, and method return values consistently adhere to their declared types, reducing the likelihood of type-related errors.
Execution: Every COOL program must include a *'Main'* class with a *'main'* method as the entry point. This method is executed by instantiating the *'Main'* class and invoking its *'main'* method.

2.2 Purpose

COOL (Classroom Object-Oriented Language) is a small programming language specifically designed for teaching object-oriented programming, type systems, and compiler design within the scope of a single-semester course. The language is designed to be simple enough for beginners while still being complex enough to cover concepts in modern programming.

COOL's syntax and semantics are clear and has a focus on type safety to make it a practical language for students to explore object-oriented programming concepts and gain experience in compiler implementation. COOL focuses on providing students with a predefined set of constructs to develop

programs, explore object-oriented principles, and understand the implementation of programming languages, rather than modifying or expanding the language itself [1].

2.3 Current State of the Implementation

A table showing the current implemented structure of the COOL project in MPS is provided in Table A.1.

2.4 Identified issues

Table 2.1 shows the identified issues when taking on the project.

Aspect	**Issues**
Typesystem	1. Improve typechecking. 2. Fix issues with SELF_TYPE vs. Object.
Constraints	1. The name of the Main class should use a checking rule instead of a constraint on the name itself.
Editor	1. Improve editor for expressions. 2. Using the comma key may not be the best interaction method; the dispatch option should be directly in the context menu. 3. Replace the standard texts for empty and not available. 4. Improve editors to allow typing verbatim text. 5. Add meaningful intentions for refactoring. 6. When typing '<-', the transformation of the Assignment concept does not add the written text to the identifier; instead, the identifier vanishes.
Generator	1. The Let concept needs a proper reduction rule.
Other	1. Create unit tests for the editor and code generation. 2. Build a standalone version. 3. Avoid explicit dependency on runtime in the module for programs.
Documentation	1. Improve the README file to explain how to navigate, use, and modify the COOL-MPS project. 2. Explain how to use MPS. 3. Integrate the issues into the Github issues section, rather than just the readme file

Table 2.1: Categorized current issues in COOL.

Aspect	**Issues**
Constraints	1. Methods within a class can have duplicate names.
Editor	1. When typing 'Inherits' after a class name and adding an inheritance relationship in the editor, the inherited class disappears from the editor immediately after being added.
Generator	1. When generating COOL methods that contain a block with another element inside it, the transformation rule, which specifies that the block itself should not be returned, fails to trigger. As a result, the "return" statement is generated in Java at the top of the method, followed by the block's square brackets and its contents.

Table 2.2: Categorized new issues in COOL.

Table 2.2 shows issues identified while working on understanding and implementing the language as described in Chapter 3.

3 Implemented solutions to issues

Aspect	Issues
Constraints and typesystem	1. The name of the Main class should use a checking rule instead of a constraint on the name itself. 2. The main class does not produce an error when not implementing a main method (a COOL requirement), only an unexplanatory name constraint. 3. Methods within a class can have duplicate names.
Generator	1. The Let concept needs a proper reduction rule.
Editor	1. When typing 'Inherits' after a class name and adding an inheritance relationship in the editor, the inherited class disappears from the editor immediately after being added.
Tests (Other)	1. Create unit tests for the editor.

Table 3.1: Selected issues in COOL successfully solved.

Table 3.1 shows the issues addressed in this implementation of COOL, derived from Table 2.1 and 2.2.

3.1 Structure

The structure is fully implemented in accordance with the COOL language specifications when comparing the overview of the COOL structure shown in Section 2.1 (Figures 2.1 and 2.2) with the current state of the implementation table in Appendix A.1. Therefore no new implementations were made to the structure itself, including the relationships and references.

3.2 Constraints and typesystem

Issue 1 addresses the need to use a checking rule (Typesystem aspect) instead of imposing a constraint on the class name itself, as shown in figure 3.1.

Class_Constraints

```
property {name}
  get <default>
  set <default>
  is valid (propertyValue, node)->boolean {
    // Must start with uppercase letter and then must contain only letters, numbers and underscore
    if (!propertyValue.matches /[A-Z] ([A-Z]|[a-z]|[0-9]|_)*/) { return false; }

    // The model with the baseclasses does not need a Main
    string[] baseClasses = {"Bool", "IO", "Int", "Object", "SELF_TYPE", "String"};
    if (!baseClasses.asSequence.contains(propertyValue)) {
      // There must be a class named Main in the model
      if (!propertyValue.equals("Main") && node.model.roots(Class).
          where({it => it:Class.name.isNotEmpty && it:Class.name.equals("Main"); }).size != 1) { return false; }
    }
```

Figure 3.1: Class constraint which was removed from the implementation and replaced by the Main Class checking rule shown in figure 3.2.

```
    string[] baseClasses = {"Bool", "IO", "Int", "Object", "SELF_TYPE", "String"};

    if (!baseClasses.asSequence.contains(c.name)) {
      if (!c.name.equals("Main") && c.model.roots(Class).where({it => it.name.isNotEmpty && it.name.equals("Main"); }).
          size != 1) {
        error c.name + " is missing Main class in the root node" -> c;
      }
    }
  }
```

Figure 3.2: Main Class checking rule.

The implemented Main Class checks for the following:

- The current class name is not *'Main'*
- the current class name is not part of the defined baseClasses.

- checks if any other class in the same model root is named Main.

If no class named Main is found, an error is raised as shown in Figure 3.4.

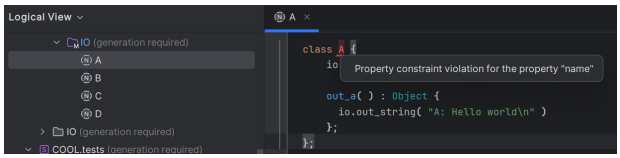

Figure 3.3: Class name constraint without a Main class in the model.

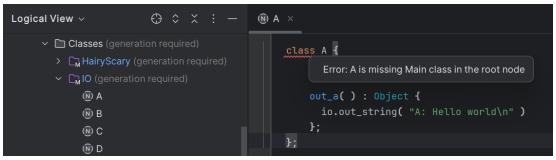

Figure 3.4: Class name checking rule without a Main class in the model.

Issue 2 addresses how the *'Main'* class does not produce an error when not implementing a *'main'* method (a COOL requirement), only an unexplanatory name property constraint.

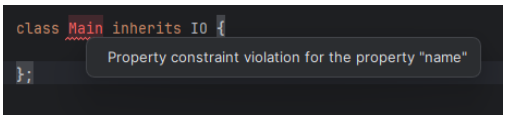

Figure 3.5: *'Main'* Class without a main Method giving a name property constraint violation.

Figure 3.6: Implementation of *'main'* Method *'Main'* Class checking rule.

Figure 3.6 shows the implementation of the *'main'* method *'Main'* class checking rule, which checks whether the current class name is *'Main'*. It then examines its node features; if any feature has the name *'main'* and is part of the concept alias Method, and if there is not exactly one such feature, an error is thrown as shown in figure 3.7.

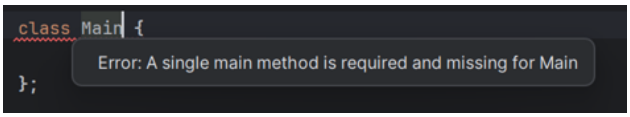

Figure 3.7: *'Main'* Class without a *'main'* Method giving a descriptive error produced by the checking rule.

A Class constraint was implemented to enforce that a *'main'* method was automatically created in the Class Editor given that the class name equaled *'Main'*, as show in Figure 3.8 and 3.9.

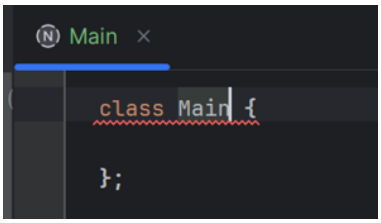

Figure 3.8: *'Main'* class without a *'main'* method.

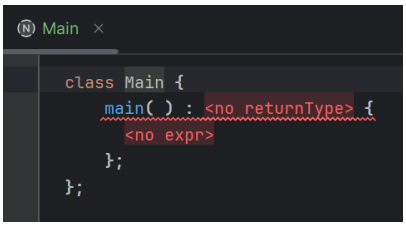

Figure 3.9: A *'main'* method automatically added to *'Main'* class when trying to edit the Class body.

```
// The Main class will enforce a main method to be created
if (propertyValue.equals("Main") && node.children.ofConcept<IMethodDeclaration>.
    where({it => it:IMethodDeclaration.name != null && it:IMethodDeclaration.name.equals("main"); }).size < 1) {
  node<Method> mainNode = new node<Method>();
  mainNode.name = "main";
  node.features.set(0, mainNode);
  return true;
}
```

Figure 3.10: main method Main Class constraint.

Figure 3.10 shows the implemented Class constraint, which checks if the current node's property value equals *'Main'* and verifies if its child nodes are of the Method concept, where the child node is not null and the method name equals *'main'*. If the count is less than 1, a new instance of a Method node is created, given the name *'main'* which is then added to the Class features, and returned.

Issue 3 was identified during the empirical testing of the language: it was possible to create several methods within a class with the same name without violating a naming constraint.

```
property {name}
  get <default>
  set <default>
  is valid (propertyValue, node)->boolean {
    //  Methods cannot have the same name
    if (node.siblings.ofConcept<Method>.where({it => it:Method.name != null && it:Method.name.equals(
          propertyValue); }).size > 0) { return false; }
    return true;
  }
```

Figure 3.11: Duplicate Method name property constraint.

Figure 3.11 shows the implementation of the duplicate method name property constraint, which checks all siblings of the current node of the Method concept. It checks where the sibling has a name and where the name equals the current node's name property value. If the count is more than 0, it means that there is another sibling node with the same name property. In other words, a method with the same name exists within the same class scope.

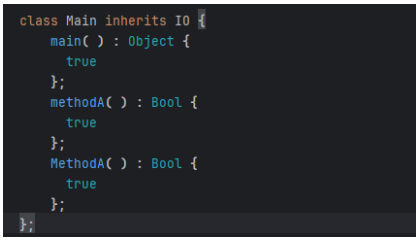

Figure 3.12: Before implementing the constraint.

Figure 3.13: After implementing the constraint.

3.3 Generator

```
print_10_comparison( ) : Object {
  {
    let num : Int <- 0; in
      {                    ;
        while num <= 10 loop
          {
            out_int( num );
            num <- num + 1;
          }
        pool;
      }
  }
};
```

Figure 3.14: Let example in COOL.

```
public Object print_10_comparison() {
  return
  {
    private int num = 0;


    while ( <= 10) {

      {
        out_int();
         =  + 1;
      };
    };
  };
}
```

Figure 3.15: Let generation example in Java with variable scope issue.

Issue 1, shown in Figure 3.14 and 3.15, presents an example where Let is used as an expression inside a method body, containing a While loop, a function call, and a variable assignment. Figure 3.15 shows the corresponding Java code generated by applying the reduction rule for the Let. In figure 3.15 the generator creates a variable `private int num = 0` inside the block (`{ }`). Due to the scope of the generated private variable, the generated While loop, its conditions and expressions are not able to reference the private variable outside the scope, hence no variables are generated after the private variable in the Java code.

```
template reduce_Attribute
input    Attribute

parameters
<< ... >>

content node:
public class Type {
  <TF [$IF$[private ->$[Type] $[attr];]] TF>
  <TF [$IF$[private ->$[Type] $[attrInitialize] = $COPY_SRC$[new Type()];]] TF>
}
```

Figure 3.16: Attribute reduction rule.

The issue was traced to the implementation of the `Attribute` concept in COOL. It was concluded that the `Let` construct itself was functioning as intended; however, the `Attribute` concept was not working correctly when invoked in the attribute initialization section of the `Let`. Figure 3.16 shows that attributes are by default, initialized as `private` variables. Those variables are permitted in Java within the class scope but cannot be declared inside methods. Doing so violates Java syntax and renders the variables unreferenceable within the method scope, as illustrated in Figure 3.15.

```
template reduce_Attribute_Local
input    Attribute

parameters
<< ... >>

content node:
public class Type {
  public void Method() {
    <TF [->$[Type] $[x] = $COPY_SRC$[new Type()]; ] TF>
  }
}
```

Figure 3.17: Local attribute reduction rule implementation.

To mitigate the issue related to Figure 3.16, a reduction rule for local Attribute initialization was implemented as shown in Figure 3.17. The generator template declaration enabled the creation of local attributes when initialized within a `Let` concept. The node hierarchy used to implement this followed the structure of a standard variable initialization in Java, using dynamic type setting via `node.type.name` reference (`->$[Type]`), dynamic naming through the `node.name` reference (`$[x]`), and the $COPY_SOURCE[`new Type()`] referencing the expression after the = sign inside the Template Fragment (<TF TF>).

Figure 3.18: Main mapping configuration implementation for the Attribute concept.

The `Attribute` concept within the main mapping configuration implementation, as shown in Figure 3.18, in the generator was changed. The first case expression was implemented to evaluate `true` if `node.parent.concept.isExactly(Let)`, ensuring that the `reduce_Attribute_local` template would only trigger when the attribute was initialized inside a `Let` concept. This change ensured that it did not interfere with other type-specific reduction templates, which are responsible for initializing private class variables.

```
print_10_comparison( ) : Object {
  {
    let num : Int <- 0; in
      {                    ;
        while num <= 10 loop
          {
            out_int( num );
            num <- num + 1;
          }
        pool;
      }
  }
};
```

Figure 3.19: Let example in COOL.

```
public Object print_10_comparison() {
  return
  {
    int num = 0;

    while (num <= 10) {

      {
        out_int(num);
        num = num + 1;
      };
    };
  };
}
```

Figure 3.20: Let generation example in Java without issue.

Figure 3.19 shows an example of the Let used in COOL, while Figure 3.20 shows the generated Java code. Figure 3.20 illustrates how the `int num = 0;` can be referenced inside both the `While` condition and the `While` body. However, the generated Java code is not entirely correct, as it fails to place the method's return statement in the correct location. This issue is related to the Method generator and not the Let itself (see Section 4.3).

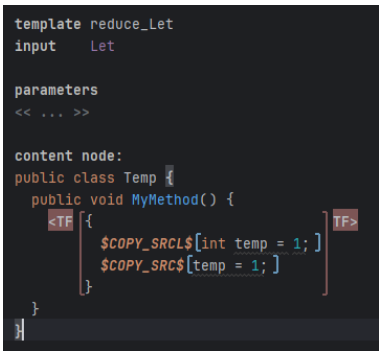

Figure 3.21: reduce Let Generator rule implementation.

Figure 3.21 shows the `reduce_Let` generator rule generating the Java code in Figure 3.20. The `$COPY_SRCL` references the list of nodes (Attributes) in the implementation of `Let` in COOL, using the same node hierarchy as `int temp = 1` ('type', 'name' = 'expression'). Similarly, `$COPY_SRC` references the nodes in the Let expression body, using the same node hierarchy as `temp = 1` ('name' = 'expression').

3.4 Editor

Issue 1 addresses an issue with performing inheritance directly in the Class editor, where it would be removed immediately after being added, as shown in Figures 3.22 and 3.23.

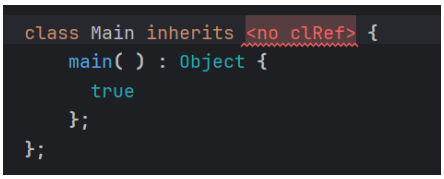

Figure 3.22: One step before adding an inheritance class in the `<no clRef> field`.

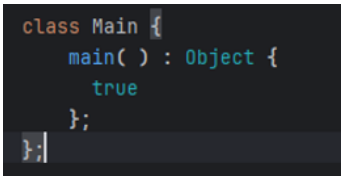

Figure 3.23: Editor after adding it (removed and inherits node detached).

The implementation issue occurred with the reference to the `name` property of the `clRef` referencing the `inherits`. When adding an `inherits` node, it would set the current node to null. This violated the editor rule implemented in the entire `?^[inherits (%inherits%->(%clRef%->{name}))-]` block, which states that if `node.inherits` is null, it hides the block and detach any `inherits` node. The issue was resolved by implementing a direct reference to the `inherits` node itself, as shown in Figure 3.25, instead of going through its `name` property.

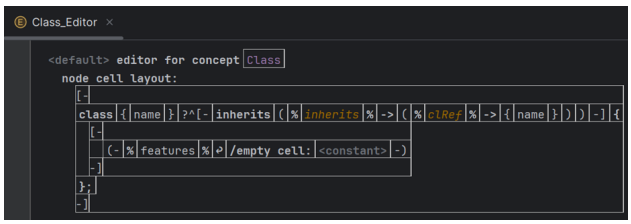

Figure 3.24: Class editor before making any changes to the implementation.

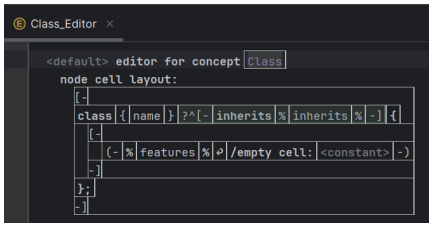

Figure 3.25: Class editor after making changes to the implementation.

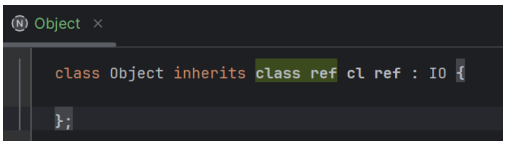

Figure 3.26: An example class in COOL inheriting from the IO class.

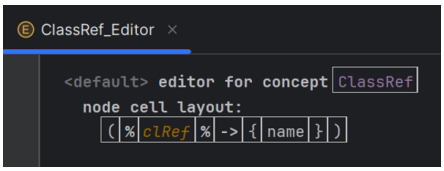

Figure 3.27: Implementation of the ClassRef editor.

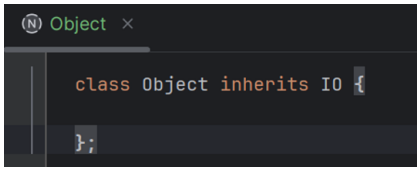

Figure 3.28: After ClassRef editor implementation.

To align the implementation with the COOL syntax, a class editor was added to the clRef to prevent it from appearing as shown in Figure 3.26 and instead display only the name property of the clRef concept, as shown in Figure 3.28.

3.5 Tests

3.5.1 EditorTests

EditorTests for all concept editors were implemented, demonstrating a before-and-after scenario where the text in the 'type' field is typed in the `<cell>` during the before scenario, expecting the results to match the after scenario, as shown in Figures 3.29 and 3.30.

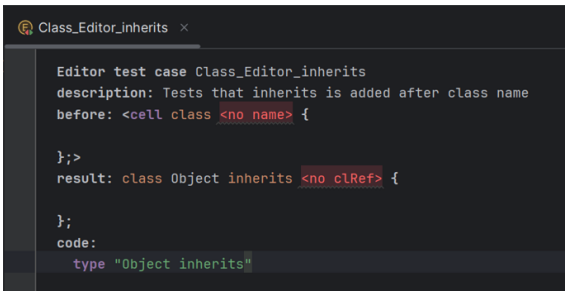

Figure 3.29: Example 1 of EditorTest implementation.

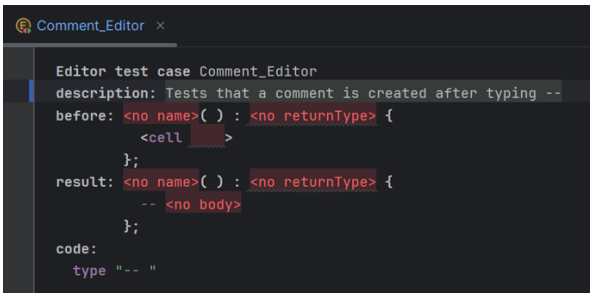

Figure 3.30: Example 2 of EditorTest implementation.

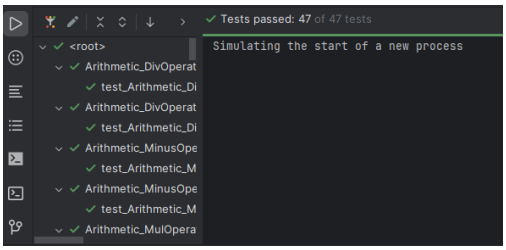

Figure 3.31: All implemented EditorTest test run.

3.5.2 NodeTests

A NodeTest were implemented, as shown in Figure 3.32, which in this case tests if the node produces an error or not. The test in this case produces an error as the class has no name property defined, hence test passes as shown in Figure 3.33

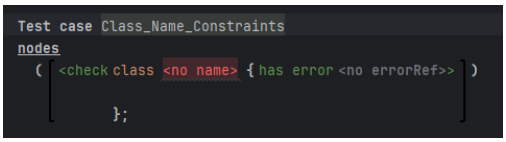

Figure 3.32: Example 1 of NodeTest.

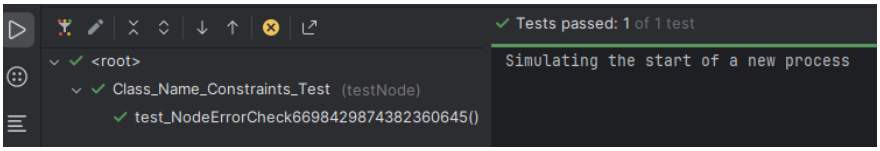

Figure 3.33: NodeTest example test passed.

4 Disucssion

4.1 Implementation and successes

The current implementation, in addition to our contribution to the identified issues, covers a significant portion of the COOL language as defined by its language specifications. The structure, constraints, and type system are implemented so that all core concepts such as class features, expressions, and references function as expected.

The successes of the implementations in this project include the successful implementation of the Let rule for the Generator, the checking rule to ensure that a model contains a *'Main'* class, and the implementation of inheritance functionality in the Class editor.

The Let rule for the generator initially appeared to be an issue with the current implementation of the rule itself, but was ultimately identified as a Java scoping issue caused by the Attribute concept in COOL and its transformation template implementation which forced private variables, making the variable inaccessible after generation of the Java code. Before changing the implementation of the Attribute concept and its transformation rule, there were some attempts which involved passing the attribute through the parameters of a method. This makes the variable accessible within the method scope. However, since Template Fragments in MPS must share the same parent node, it was not possible to create separate Template Fragments for the method parameters and method expressions, leaving out the rest of the method syntax. This forced us to include the entire method syntax. Hence the actual generation would not turn out as correct Java syntax as shown in figure 4.1. Here the Generated initialized variable is accessible to the assignment inside the scope.

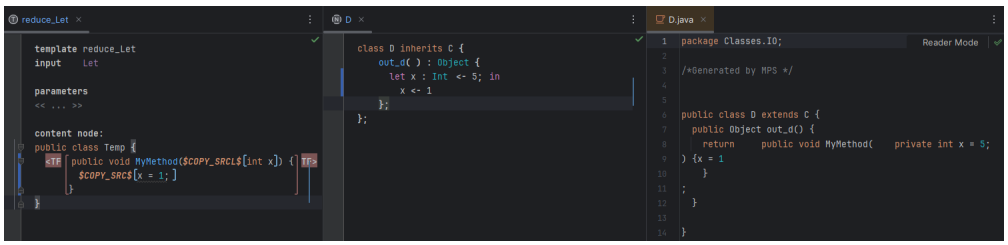

Figure 4.1: Example of early Let rule issue fix implementation attempt.

4.2 Empirical evidence

Several test cases were made testing the concept editors, as described in Section 3.5, showing the correct user functionality. Figure 4.2 provides an example of functional class inheritance and the use of inherited classes. It also demonstrates a properly functioning Main class, including the use of methods and the initialization of attributes.

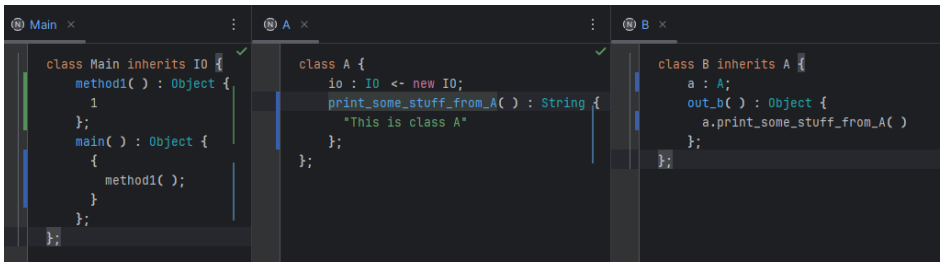

Figure 4.2: Working Class inheritance example with attributes and methods.

```
class Main inherits IO {
    and( b1 : Bool, b2 : Bool ) : Bool {
      if b1
      then b2
      else b1
      fi
    };

    pal( s : String ) : Bool {
      if s.length( ) < 2
      then true
      else and( s.substr( 0, 1 ) = s.substr( s.length( ) - 1, s.length( ) ), pal( s.substr( 1, s.length( ) - 1 ) ) )
      fi
    };

    i : Int;

    main( ) : SELF_TYPE {
      {
        i <- ~1;
        out_string( "Please enter a string:\n" );
        if pal( in_string( ) )
        then out_string( "That was a palindrome.\n" )
        else out_string( "That was not a palindrome.\n" )
        fi;
      }
    };
};
```

Figure 4.3: Working Palindrome example.

Figure 4.3 shows a working Palindrome example that utilizes the runtime base classes inherited through the IO class, specifically the out_string and in_string methods. These methods, as shown in Figure 4.4, leverage the Java Scanner class for input and println for output.

```
public class IO extends CoolObject {

  public IO out_string(String x) {
    System.out.println(x);
    return this;
  }

  public IO out_int(int x) {
    System.out.println(x);
    return this;
  }

  public String in_string() {
    Scanner sc = new Scanner(System.in);
    return sc.nextLine();
  }

  public int in_int() {
    Scanner sc = new Scanner(System.in);
    return sc.nextInt();
  }
}
```

Figure 4.4: IO baseclasses.

4.3 Current Issues and Limitations

All issues mentioned in issue Table 2.1 and 2.2, which are not addressed in the implemented solutions Chapter 3 of this project, remain current issues and limitations. However, many of these are more focused on improving usability for users, such as refining the editor and its interactions. One notable issue is related to the Generator. In many cases, it generates Java code that is not

necessarily syntactically correct. For instance, when generating a function containing a Block concept with another concept, such as Let or while, the generated code places a return statement at the beginning of the Java method, rendering the generated code unusable. This occurs despite the template declaration for the Method concept implementing an 'IF' condition that checks whether its expressions are instances of concepts such as Block or While. In such cases, no return statement should be placed at the beginning of the generated code, as shown in Figure 4.5.

Figure 4.5: Wrongly generated Java code.

There is also a limitation when it comes to creating Generation tests. For example, as shown in Figure 4.6, which shows a COOL example of a Class and its corresponding Java code, it is evident that the generated code for both is identical. However, since Java includes a ClassifierType concept, which is added to classes as a child node, the comparison of the generated Java code fails. This is because COOL's implementation of the ClassType concept only establishes a one-way relationship with the Class concept which is not referencing it back. As a result, the Class concept in COOL does not implement a superclass as a child node of the class node object. Consequently, comparing the two models in a Generator test fails at the node level. This is an issue because, when using the language, a class must serve as the root node. As a result, all Generation tests fail before the code within the class can be processed and compared to the corresponding generated Java code.

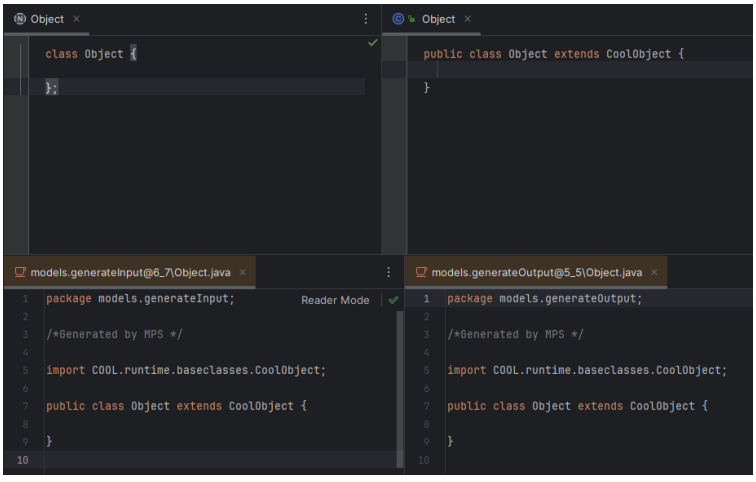

Figure 4.6: Generated Java code for COOL and Java.

There is also a limitation regarding inheritance. When inheriting from a class, the user must create a new object of the class before being able to access any of the inherited class's methods. Ideally, these methods should be accessible directly through inheritance without the need to instantiate a new object of the class.

4.4 Future steps

Future steps of the project should primarily focus on improving the Generator to ensure that all its transformation templates produce Java code that is both functional and syntactically correct. Addressing the issue with Generation tests would be an important step toward achieving this goal while refining the Generator. Although reviewing the list of current issues should also be considered, prioritizing work on the Generator is more impactful than simply adjusting the editor or implementing quality-of-life improvements. Lastly, if time permits, significant effort should be directed toward improving the project's documentation. This would save future project groups considerable time, as opening a project like COOL in MPS for the first time can feel overwhelming. This has been documented by groups from previous years and also applies to this year. The documentation should focus specifically on the project within MPS, as there is already a manual[1] that primarily covers the language specifications.

5 Plan and Reflection

5.1 Project management

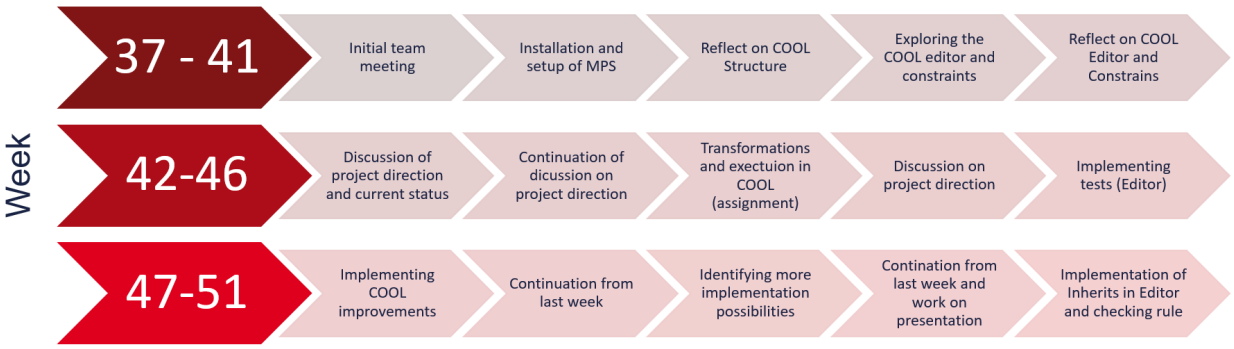

Figure 5.1: Timeline of the COOL Project, outlining major milestones and activities across the weeks.

We had weekly group meetings lasting 1–2 hours, starting from Week 37 to Week 50, where we discussed assignments related to the COOL implementation and the project. These meetings helped us understand MPS and COOL better, with team discussions being very important to the learning process. Additionally, course seminars provided opportunities to consult with the supervisor on project focus. Initially, much time was spent understanding MPS and the latest implementation of COOL. Around Week 45, we decided to focus on writing tests for the current implementation. This approach allowed us to align efforts and achieve meaningful contributions. Team contributions were equal, as most tasks were completed collaboratively during physical and online meetings to enable real-time discussions and idea testing.

5.2 Questions for the MPS Learning Quiz

Question 1: How should you think about MPS as a tool?

Suggested Answer: Do not think of MPS as an IDE or text editor. Instead, view it as a tool for structuring and interacting with nodes within node hierarchies.

Question 2: Does the whole project need to generate/compile without errors upon starting/pulling the project from Git?

Suggested Answer: No, MPS is a DSL workbench and not a typical IDE. You can still work on the project even if there are many errors.

Question 3: How can you see more details about something in the MPS editor?

Suggested Answer: Mark or select the element you want to see more details about and choose "Show Reflective Editor for the Subtree."

A Appendix - Table

Structure Element	Extends	Implements	Properties	Children	References
DivOperation	BaseConcept	IArithmeticOperation	Undefined	Undefined	Undefined
IArithmeticOperationNo	IExpression	Undefined	Undefined	left: IExpression, right: IExpression	Undefined
MinusOperation	BaseConcept	IArithmeticOperationNo	Undefined	Undefined	Undefined
MulOperation	BaseConcept	IArithmeticOperationNo	Undefined	Undefined	Undefined
PlusOperation	BaseConcept	IArithmeticOperationNo	Undefined	Undefined	Undefined
EqualComparison	BaseConcept	IComparisonOperation	Undefined	Undefined	Undefined
IComparisonOperation	IExpression	Undefined	Undefined	left: IExpression, right: IExpression	Undefined
LessThanComparison	BaseConcept	IComparisonOperation	Undefined	Undefined	Undefined
LessThanOrEqualComparison	BaseConcept	IComparisonOperation	Undefined	Undefined	Undefined
BoolConst	BaseConcept	IConstant	Undefined	Value : boolean	Undefined
IConstant	IExpression	Undefined	Undefined	Undefined	Undefined
IntConst	BaseConcept	IConstant	Undefined	value : integer	Undefined
StringConst	BaseConcept	IConstant	Undefined	value : string	Undefined
Assignment	BaseConcept	IExpression	Undefined	expr : IExpression	id : IAttributeDeclaration
AttributeIdentifier	BaseConcept	IExpression	Undefined	Undefined	attr : IAttributeDeclaration[1]
Block	BaseConcept	IExpression	undefined	exprs : IExpression[1..n]	Undefined
Case	BaseConcept	IExpression, ScopeProvider	Undefined	mainExpr : IExpression[1], branches :	Undefined
CaseExpr	BaseConcept	IAttributeDeclaration	Undefined	expr : IExpression[1]	type : Class[1]
Conditional	BaseConcept	IExpression	Undefined	IExpression[1], elseExpr : IExpression[1]	Undefined
Dispatch	BaseConcept	IExpression, ScopeProvider	showStaticTypeField : Boolean	parameters : IExpression[0..n], expr : IExpression[0..n]	methodCalled : IMethodDeclaration[1], type : Class[0..1]
IAttributeDeclaration	ICoolName	Undefined	Undefined	Undefined	Undefined
IExpression	Undefined	Undefined	Undefined	Undefined	Undefined
IMethodDeclaration	ICoolName	Undefined	Undefined	Undefined	Undefined
Let	BaseConcept	IExpression, ScopeProvider	Undefined	attri : Attribute[1..n], expr :	Undefined
MethodIdentifier	BaseConcept	Undefined	Undefined	Undefined	method : IMethodDeclaration[1]
New	BaseConcept	IExpression	Undefined	Undefined	type : Class[1]
Parenthesis	BaseConcept	IExpression	Undefined	expr : IExpression[1]	Undefined
WhileLoop	BaseConcept	IExpression, ScopeProvider	Undefined	loopExpr : IExpression[1], loopBody :	Undefined
Attribute	BaseConcept	IFeature, IAttributeDeclaration	Undefined	expr : IExpression[0..1]	type : Class[1]
IFeature	ICoolName	Undefined	Undefined	Undefined	Undefined
Method	BaseConcept	IFeature, IMethodDeclaration, ScopeProvider	Undefined	expr : IExpression[1], parameters :	returnType : Class[1]
Class	BaseConcept	ICoolName, ScopeProvider, IMainClass	Undefined	features : IFeature[0..n], inherits :	Undefined
ClassRef	BaseConcept	Undefined	Undefined	Undefined	clRef : Class[1]
ClassType	BaseConcept	Undefined	Undefined	Undefined	cls : Class[1]
Comment	BaseConcept	IExpression	body : string	Undefined	Undefined
Formal	BaseConcept	IAttributeDeclaration	Undefined	Undefined	type : Class[1]
ICoolName	INamedConcept	Undefined	Undefined	Undefined	Undefined

Figure A.1: Current state of the AST implementation of COOL in MPS

References

[1] [Online; accessed 10. Dec. 2024]. Jan. 2011. URL: https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf.

[2] *COOL-MPS*. [Online; accessed 10. Dec. 2024]. Dec. 2024. URL: https://github.com/uiano/COOL-MPS.