An Implementation of the Specification Language ACT ONE in MPS

DAT304-G
Research Project, Computer Engineering

Author
Trong Duc Truong

Supervisor
Andreas Prinz

Grimstad, May 2019

UiA Faculty of Engineering and Science

Title:	An Implementation of the Specification Language ACT ONE in MPS
Keywords:	Specification Language, Term reducing, ACT ONE, MPS

Author:	Trong Duc Truong
Supervisor:	Andreas Prinz

Distribution:	Open

Course:	DAT304-G - Research Project, Computer Engineering
University:	University of Agder
Faculty:	Faculty of Engineering and Science

Semester:	Spring 2019
Date:	Grimstad, May 2019

Address:	Jon Lilletuns vei 9, 4879 Grimstad
Phone:	+47 37 23 30 00
Telefax:	+47 38 14 10 01

Abstract

The aim of this project was to use MPS to implement a system to work with the algebraic specification language ACT ONE, tailored towards the course DAT233 held at the University of Agder.

The final product is an IDE that caters to a modified ACT ONE and its rewrite system. The primary quality attribute we prioritised for our implementation was correctness, as we deemed this important in according to DAT233.

Concluded, the final product works as intended. It allows users to specify specifications and perform term rewriting with them. It is accessible in the sense that the all components are coupled together under a single MPS project. For these reasons, the final product, although flawed, provides sufficient features to warrant its usage in future lectures in DAT233.

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgement

I would like to thank my supervisor Prof. Andreas Prinz for assisting me with the thesis in dire times where my forehead, admittedly, went banging into the wall so often that I almost must thank the wall as well, for being right around the corner whenever I needed it, akin to how Prof. Prinz was for me.

I would also like to acknowledge my brother Tam Thanh Truong as the second reader of this thesis; I am gratefully indebted for his valuable comments on this thesis.

Contents

Chapter 1

Introduction

1.1 Background

In 1983 the algebraic specification language *ACT ONE* was designed to integrate various concepts within specification languages together with rigorous semantics [11, p. 268]. During this time the *ACT ONE system*, a collection of tools to work with the language, was implemented. This included an editor, a parser, an interpreter, and libraries [11, p. 272]. Further development of this system has since ceased.

The underlying concepts of ACT ONE are still being taught in the course DAT233 held at the University of Agder. At the time of writing, it is taught without a current development tool for it, forcing both the students and professors to either write ACT ONE specifications without an accompanying tool, or to rely on other algebraic specification languages. The project at hand will try to alleviate this dilemma by developing a corresponding tool, akin to the mentioned ACT ONE system.

1.2 Problem Statement

As mentioned, the ACT ONE system is no longer available, and creating a new one, independent from the previous system, is therefore more relevant. We will make use of Jetbrains' Meta-Programming System (MPS) [17] to accommodate this task, a software we will elaborate more in Chapter 4. Hence, the problem statement for this project will be:

> *Implement the specification language ACT ONE in MPS, fitted with features inline with the course DAT233.*

The proposed implementation will be an accumulation of the following components:

1. Our own implementation of the language ACT ONE.

2. A term reduction system for it.

3. A corresponding Integrated Development Environment (IDE).

It is important that this system is accessible for students and professors alike, as it will be designed in accordance with the course DAT233. This is also the reason why the design and functionality of the final product will be heavily tailored towards it.

1.2.1 Scope

In-Scope

Items mentioned in this section are within range of relevance, and will be elaborated during the project.

The specification language ACT ONE as defined by Ehrig [9] will be the version used as a point of reference. This version will henceforth be referred to as the *original ACT ONE*.

Alternative language concepts that are not defined in the original ACT ONE may be implemented to increase the expressiveness of our version of ACT ONE.

Using MPS to implement our version of ACT ONE is a requirement for this project.

Out-of-Scope

The following items will not be elaborated during the project, mainly due to time restrictions.

A complete replica of ACT ONE is not our goal, due to the vast collection of concepts in the original definition. A more concise and limited version is more preferable.

Large-scale testing with many testers will not be inducted. All testings will instead be performed by us only.

1.3 Overview

The thesis will explore its contents in the following order:

Theory: Explores the theory behind algebraic specifications and term reduction, what the language ACT ONE entails, and what MPS is.

Method: describes the methods used to solve the aforementioned objectives.

Result: Goes into detail on the results we acquired, that is, the implementation of ACT ONE, its reduction system, and the IDE.

Discussion: Discusses the results and how well they conform with the functional requirements we outlined throughout the project.

Conclusion: Summarises the project and its future.

Chapter 2

Theory

2.1 Data Type

A *data type* specifies how a given data should be interpreted by a compiler or interpreter. More specifically, it defines a collection of data values and a set of predefined operations on those values [20, p. 260]. Examples of some common data types are `Integer` and `Boolean`, representing whole numbers and truth values, respectively. Given a data type, there are two different perspectives one can look at it:

1. From the implementer's perspective, a data type is perceived as a *data structure*: the actual implementation of the data type.
2. From the user's perspective, a data type is perceived as an *abstract data type*: a data type separated away from its implementation.

The `Stack` data type may make this distinction clearer: Users of `Stack` are interested in how they can add elements to the stack, and how they can retrieve elements from it. Conversely, the implementer worries more about how the actual implementation of the stack data type should be carried out. Should it implemented as an array or should it be implemented as a linked list?

Since abstract data types are more relevant for the coming chapters, a more complete definition is needed: An abstract data type is a theoretical model of a data type, where all notions of implementation are omitted. This view allows a discussion of specific data types without being limited to concrete boundaries like programming languages or technology [1].

2.2 Specification

In general

To assure a common understanding of a system between the implementer and the user, a *specification* of the system might prove to be useful. A specification is a description of a system explaining its design, properties, expected behaviour etc. [23].

Although specifications provide a great range of usage, this thesis will exclusively discuss their application with abstract data types. The specification of an unbounded stack, presented in [25, ch. 8.2], is an example of such:

Example 1 (A specification of an unbounded stack of natural numbers)

`init` : *An operation which creates an initial and empty stack.*

`push` : *An operation which adds a natural number to the top of an existing stack to produce a new stack.*

`pop` : *An operation which takes a stack as input and produces a new stack with the top-most natural number removed.*

`top` : *An operation which takes a stack as input and returns the natural number at the top of the stack.*

`isempty` : *An operation which takes a stack as input and returns* `true` *if the stack is empty, and* `false` *otherwise.*

Despite only describing the available operations for a stack, this specification describes the essence of the data type. However, this specification has some caveats due to its reliance on textual descriptions. One such caveat is the lack of apparent rules for the structure of the descriptions, allowing inconsistent structure of descriptions across multiple specifications and, perhaps even worse, within the same specification. Another caveat is that the specification will be language-dependent, as the descriptions must be written in one language or another. A more consistent, yet precise, method of specification is needed. Given the mentioned caveats, two qualities are sought here:

Uniformity: the structure of specifications should be the same for all specifications, independent of the system to be described.

Formality: the language used in the specification must be independent of both programming language and written language.

Formal Specification

We introduce the idea of *formal specifications*, a subset of specifications that encompasses the above-mentioned qualities. Uniformity is achieved by using universally agreed upon methods to describe systems, whereas formality is realised through the usage of mathematical language or other formal languages [14, ch. 2]. Depending on the system to be described, various structures of formal specifications exist that may be more suitable than others. These methods includes, but are not limited to: Algebraic Specification, Transition-Based Specification, and Model-Based Specification [13]. Only algebraic specifications, however, will be elaborated further in this thesis.

2.3 Algebraic Specification

The possibility of considering a data type as an algebra, as a means to specify abstract data types, was realised by Zilles in 1974 [26]. Said algebras, namely a *many-sorted algebras*, provides the foundation for algebraic specifications. The theory of many-sorted algebras will be touched upon, but not elaborated, in this thesis, whereas the elaboration of algebraic specifications will be kept simple and acts more like a general overview. For a more comprehensible discussion of many-sorted algebras and algebraic specifications, see [24]. Hereinafter, many-sorted algebras shall be referred to as just algebras.

The concept of algebraic specifications may seem daunting at first due to its mathematical nature. To fully understand algebraic specifications one needs an understanding of the underlying concepts: Signatures and algebras. To make matters worse, these underlying concepts houses further, underlying concepts that need to be understood to see the complete picture. For that reason, this section serves as an explanation of all the pieces that constitutes an algebraic specification, explained in an incremental fashion.

2.3.1 Syntax and Semantics

Both *syntax* and *semantics* are prevalent terminologies when discussing the concept of specifications, and are defined as follows:

Syntax: The arrangement of words in sentences.

Semantics: The meaning of an arrangement of words.

2.3.2 Signature

Before we define a signature, we begin with an explanation of the concept of *sorts*, a fundamental piece that ties all the aforementioned concepts together.

Sorts

Unlike the case of unsorted algebras, many-sorted algebras introduce the concept of *sorts*. This notion of sort may feel abstract and therefore, be difficult to grasp for many, which motivates a thorough explanation in this thesis. Formally, a sort represents a domain of elements, or a category of elements. Examples of such domains of elements are the set of all integers $\mathbb{Z}$ or the boolean domain $\mathbb{B}$. Note, however, that a sort only *represents* a domain, meaning that **a sort in itself is nothing but a placeholder**. This will be more apparent when we compare the definitions of signature and algebra with each other.

Operation Symbol

Paraphrased from [9], we have the following definition for an operation symbol:

Definition 2.3.1 *An operation symbol declaration*

$$N : s_1, \ldots, s_n \to s \quad (n \geq 0)$$

consists of an operation name N, a list of input sorts $s_1, \ldots, s_n$*, and an output sort s.*

Similarly to sorts, an operation symbol is also just a placeholder but for operations. Note that an operation symbol can have an unlimited number of input sorts, but only a single output sort.

Signature

From [9], we have the following definition of a signature:

Definition 2.3.2 *A signature*

$$\Sigma = (S, OP)$$

consists of a set S of sorts and a set OP of operation symbols.

Considering the definition for signature, it can be concluded that it acts as a collection for the two previous concepts, namely sorts and operation symbols. Figure 2.1 show this relation in a graphical way. An example of a signature that fits the specification given in Example 1 can be expressed as:

```
Sorts:
    Stack
    Nat
    Bool
Operations:
    init    :           -> Stack
    push    : Stack Nat -> Stack
    pop     : Stack     -> Stack
    top     : Stack     -> Nat
    isempty : Stack     -> Bool
```

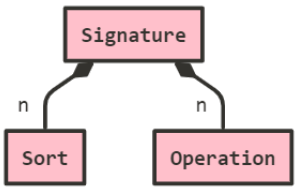
Listing 1: A signature that conforms with Example 1

Note that this is just one of many signatures that conforms with the specification at hand. This is due to sorts and operation symbols only being placeholders; the exact names do not matter other than being identifiers. This cannot be stressed enough! To illustrate this, we present the following signature which also conforms with the specification:

```
Sorts:
    Dog
    Cat
    Mouse
Operations:
    cow     :           -> Dog
    horse   : Dog Cat   -> Dog
    donkey  : Dog       -> Dog
    alpaca  : Dog       -> Cat
    tiger   : Dog       -> Mouse
```

To sum up, signatures provide a structure that embodies sorts and operation symbols. They do not, however, provide any context on the behaviour of these two components. We know the expected input and output of each operation symbol, but nothing about their behaviour and relation with other operation symbols. In that sense, signatures provide the syntax, but not the semantics. In the following sections we will see that the semantics can be settled when we use signatures with *specifications*.

Figure 2.1: A diagram showing the composition of a signature.



2.3.3 Term

The operation symbols of a signature can be composed together into *terms* by using the output of an operation symbol as the input for other operation symbols [10]. The following terms can be derived from Listing 1:

```
init
isempty(init)
pop(push(init,4))
top(push(pop(push(init,6)),3))
```

Observe that operation symbols such as `init` and `4` are denoted without parentheses. This is due to them being *operation constants*, operation symbols without any input sorts.

An important quality of a signature is its implications of correct syntax and correct semantics: All the terms in the previous example are both syntactically correct and semantically correct according to Listing 1. However, erroneous terms exist: From the same signature, these derived terms are invalid:

```
in(n)           % Error due to the usage of unrecognised tokens
push(init)      % Error due to missing arguments
isempty(4)      % Error due to type error
init(3)         % Error due to excessive number of arguments
```

Remark 1 *The difference between syntactical errors and semantic errors is not definite. An error which some authors would deem a semantic error, could be considered a syntactical error by others, depending on whether it is caught by the parser before the compiler does or not (see [2, 22, 19]). Due to the static nature of specifications, the thesis classifies the difference like this:*

Syntactical errors: *lexical errors, referencing to a undeclared operation, and type errors.*

Semantic errors: *unintended behaviour errors.*

Before proceeding further, we must acknowledge that both examples above contain some inaccuracies, namely the numerical constants `3`, `4` and `6`. We have not explicitly defined these constants as valid operations of the signature, meaning that they, in reality, do **not** exist. For now, we assume that all natural number constants exist for all signatures.

2.3.4 Specification

Before delving directly into the definition of a specification we must explain the concept of *axiom*.

Axiom

Consider this scenario: We have a stack of natural numbers, of which we push a natural number to. If we now proceed to check the current natural number on top of the stack, what natural number can we expect to find? Without a doubt the recently pushed natural number. In fact, this is always the case for any given stack s or any natural number n. We can denote this scenario as

$$\forall n \in \texttt{Nat} \quad \wedge \quad \forall s \in \texttt{Stack} \implies top(push(s, n)) = n$$

or the equivalent

$$\forall n : \texttt{Nat} \quad \wedge \quad \forall s : \texttt{Stack} \implies top(push(s, n)) = n$$

This scenario presents an example of an *axiom*, a statement that is regarded as being self-evidently true [3]. A formal definition of an axiom is [25]:

Definition 2.3.3 *An axiom $a = \langle V, L, R \rangle$ consists of a set V of variables, L and R where both are terms.*

In the aforementioned scenario, the variables were n:`Nat` and s:`Stack`, whereas $L = top(push(s, n))$ and $R = n$.

Specification

Paraphrased from [9], we have the following definition for a specification:

Definition 2.3.4 *An algebraic specification*

$$SPEC = (S, OP, A)$$

consists of a set S of sorts, a set OP of operation symbols, and a set A of axioms.

Moreover, Definition 2.3.4 can be merged with Definition 2.3.2, giving us the following alternative:

Definition 2.3.5 *An algebraic specification*

$$SPEC = (\Sigma, A)$$

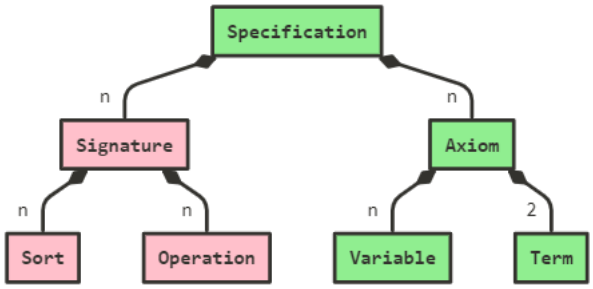
consists of the signature $\Sigma = (S, OP)$ and a set A of axioms.

Essentially, an algebraic specification is a signature, but accompanied with axioms. The signature provides the syntactical component of the specification; it outlines which operation symbols that are within the context of the said specification. Additionally, the axioms describe the semantic component of the specification; how

two or more operation symbols behave with each-other. In conclusion, an algebraic specification of an abstract data type provides us a formal description of both the syntactical component and the semantic component of the abstract data type.

Applying the new concepts to Figure 2.1 results in a more complete picture, as seen in Figure 2.2.

Figure 2.2: A diagram showing the composition of a specification.



2.4 ACT ONE

Up to this point, we referred to ACT ONE as an *algebraic specification language* without explaining what that really means. In the previous sections we explored the formal definitions of algebraic specification and its underlying concepts. These formal definitions describes how all algebraic specifications behave in general. An algebraic specification language, however, is a language used to express actual instances of algebraic specifications. Different languages have different rules on how one express certain concepts of a specification. The `BOOL`-specification in Listing 2 and the `NATSTACK`-specification in Listing 3 are both written in according to the original ACT ONE's syntax.

```
type BOOL is
sorts
    Bool
functions
        vars b, b1: Bool;

        func non: Bool -> Bool;
                non(true) = false;
                non(false) = true;

        func _and_: Bool,Bool -> Bool;
                true and b = b;
                false and b = false;

        func _or_: Bool, Bool -> Bool;
                true or b = true;
                false or b = b;

        func _xor_: Bool, Bool -> Bool;
                true xor b = non(b);
                false xor b = b;
endtype
```

Listing 2: A specification `BOOL` [9] written in original ACT ONE.

```
type NATSTACK is
sorts
        Stack
        Nat

constructors
        init: -> Stack;
        push: Stack Nat -> Stack;

functions
    vars
            s : Stack;
            n : Nat;

    func top: Stack -> Nat;
            top(push(s,n)) = n;

    func pop: Stack -> Stack;
            pop(push(s,n)) = s;

    func isempty: Stack(D) -> Bool;
            isempty(init) = true;
            isempty(push(s,n)) = false;

endtype
```

Listing 3: A specification `NATSTACK` [9] written in original ACT ONE.

2.5 Term Rewriting

Example of term rewriting

Given an empty stack, if you push an element (e.g. the value 7) into it and ask the stack to retrieve the next element in line, we expect to retrieve the recently-pushed element. We will use this scenario to explain the concept of term equivalence. Recall the `NATSTACK`-specification from Listing 3. We can use it to express the scenario above as

$$top(push(init, 7)) \tag{2.1}$$

Furthermore, we can show that the expression reduces to the value 7 with one of the axioms provided by the specification, which is defined as:

$$top(push(s, n)) = n \tag{2.2}$$

We can see that the structure of expression 2.1 fits the left-hand side of the axiom Equation (2.2) with the corresponding unifier

$$s \leftarrow init \quad ; \quad n \leftarrow 7 \tag{2.3}$$

which is valid as `init` is a `Stack`-value and therefore conforms with variable `s`, and likewise for the `Nat`-value `7` and the variable `n`. As such, we arrive to the result

$$top(push(init, 7)) = 7 \tag{2.4}$$

as expected.

Definition

A definition of term rewriting [9] is as follows:

> Rewriting is a subsequent left to right application of the equations until no more equation is applicable. The equations are used as rewrite relations.

Remark 2 *In the literature the definitions of* term rewriting *and* term reduction *have been used interchangeably (see [9, 25, 15]). For the sake of consistency, we will use term rewriting only throughout this thesis.*

2.5.1 Axiom equivalence

Axioms of a specification can be interpret in two different ways. They can be treated as equations in the mathematical sense, or as rewrite relations [25, ch. 10.11]. The

difference between these two interpretations is subtle but important. From mathematics, we know that equality have the innate properties:

For any object α, β, θ:

$$\alpha = \alpha \qquad \text{(Reflexive)}$$

$$\text{if } \alpha = \beta \text{ then } \beta = \alpha \qquad \text{(Symmetric)}$$

$$\text{if } \alpha = \beta \text{ and } \beta = \theta \text{ then } \alpha = \theta \qquad \text{(Transitive)}$$

If the axioms were to be treated as equations, the equality relation "=" is reflexive, symmetric, and transitive. If the axiom is treated as a rewriting rule however, the equality operator "=" will be interpreted as a rewrite relation "$\rightarrow$" instead, such that an axiom is represented as

$$L \rightarrow R \tag{2.5}$$

where L, R denote the left and right components of the rule [25, ch. 10.12]. Unlike the equality relation, the rewrite relation is transitive, but not symmetric nor reflexive. Another take on the difference, is to think of the rewrite relation as a unidirectional relation in compared to the bi-directional equality relation. Note that this is about the interpretation of the axiom specifically, and not altering the actual context of axiom. The following example may make it easier to grasp this distinction:

Given the equation

$$x + 0 = x \tag{2.6}$$

If we were to treat this as the rewriting relation

$$x + 0 \rightarrow x \tag{2.7}$$

Equation (2.6) will, obviously, remain true. Equation (2.7) is merely a deduction of Equation (2.6). The introduction of the rewrite relation may seem redundant, as it seems like a more restricted form of the equality relation. In fact, it is this level of strictness that makes it useful. Perhaps the following example illustrates this aspect better. Given the Equation (2.6), the following equation, due to the symmetric property of equality, is true as well:

$$x = x + 0 \tag{2.8}$$

The fact that this is true is not a problem in itself, but the implication of it may be problematic in term rewriting. If we substitute the x in the right-hand side with the equation itself, we get

$$x = (x + 0) + 0 \tag{2.9}$$

It is clear that this process can be repeated indefinitely, resulting a infinitely long expression on the right-hand side. Computing an infinitely-expanding expression is seldom good and should be avoided. Treating the equation as a rewriting relation is a way to accommodate this. If the Equation (2.6) was treated as the rewrite relation 2.7 instead, then the Equation (2.8) and the rewrite relation

$$x \to x + 0 \tag{2.10}$$

are no longer implicitly true.

In conclusion, even though an axiom with an equality operator is correct, it may allow unwanted divergence during term rewriting in some cases. Treating it as a reduction relation instead is a means to avoid it.

2.5.2 n-Step-Rewriting and Exhaustive-Rewriting

We introduce two forms of rewriting: *Step-Rewriting* and *Exhaustive-Rewriting*. A n-Step-Rewriting rewrites a given term n times or until no further rewrites are possible. After n rewrites, it terminates regardless of whether there exist further possible rewrites or not. An Exhaustive-Rewriting, however, rewrites a given term until no further rewrites are possible, and will therefore never prematurely terminate. Given the definition, Exhaustive-Rewriting is equivalent with x-Step-Rewriting, when $x \to \infty$.

2.5.3 Breadth-First Rewriting and Depth-First Rewriting

A rewriting can be done in two different orders: *Normal-order evaluation* and *Applicative-order evaluation* [25, ch. 9.10]. The difference between these two orders can be illustrated with an example. Using the `NATSTACK`-specification from Listing 3, we are given the term:

```
top(push(pop(init),3)
```

With normal-order evaluation, this term is evaluated from the "outside-in", meaning that the rewriting begins with `top(push(*),3)`, where `*` is a placeholder for terms. Furthermore, we see that the specification defines the axiom

```
top(push(s,n) = n)
```

which our term conforms with, resulting `3` as the rewritten term.

With applicative-order evaluation, however, the evaluation begins with the innermost arguments of the term. Given the same example, applicative-order evaluation will evaluate firstly `init`, of which there exist no axioms for. Next evaluation becomes `pop(init)`, which should result an error, as popping an empty stack is not possible. However, since we have not declared any error-handling with our specification, the evaluation proceeds to `3`, followed by `push(pop(init),3)`, and lastly followed by `top(push(*),3)`, which we saw earlier results to `3`. Although they end with the same results here, this is not always the case, as seen in [25, ch. 9.10].

In Chapter 4, we refer to normal-order evaluation and applicative-order evaluation as *BFS* and *DFS*, respectively.

2.6 Model-Driven Development

Model-Driven Development (MDD) is a software development process where development progress is achieved through the use of *models* rather than computer programs [21]. Models here being representations of the actual implementation units. MDD is often associated with two advantages:

Abstraction: Models encapsulates the actual implementation which, in other words, makes it more simplified for developers to work with. Models can become more domain-specific, making it more apprehensible for the domain experts.

Automation: Creating models and, from it, have code automatically generated is possible and often preferable for MDD [5].

The philosophies of MDD can be observed in MPS.

2.7 MPS

When creating a new computer language one may make use of language workbenches, software tools designed to help users develop new computer languages. JetBrains' Metalanguage System (MPS) is an example of a language workbench [8]. This section will elaborate some of the features of MPS that will be important for later chapters.

2.7.1 BaseLanguage

As the name suggests, creating a language in MPS requires programming. MPS provides a programming language called *BaseLanguage* one can use to define the needed logic of the language to be created. BaseLanguage is essentially an extended

version of Java 6, as it shares similar set of structures from it. However, BaseLanguage has additional features that were designed to work with the underlying concepts of MPS [4].

2.7.2 Projectional Editor

During the compilation or interpretation of some source code, the code get parsed into *abstract syntax trees* (AST), data structures that represent the syntactical structure of the program [20]. It is through analysing these ASTs where syntactical errors can be found, as the parser have some predetermined rules on how well-formed ASTs should look like. Without valid ASTs the program cannot be further compiled or interpreted. Considering how a syntax is determined by the parser, it is clear that languages with different syntax cannot conform to the same parser, as it does not know how to parse the given input into ASTs.

One of the features of MPS is its *projectional editor*, an user interface which allows users to interact with the ASTs directly. An advantage of working with ASTs directly in contrast to not, is that the user is shielded away from the involvement with the parser itself, effectively allowing users to create languages without worrying about the parser. This is possible because all languages that are created with MPS is, in reality, written in another language MPS can parse [8].

2.7.3 Models

As mentioned earlier, MPS follows some of the philosophies of MDD. Programs in MPS are organised into models [18]. These models are further organised into the following *modules*:

Language: The language definition and features. Models in this module form a language and the corresponding IDE for it.

Solution: The container for the end users' code. Users use languages defined in language-modules to create code they can store in this module as models.

Generator: Contains models that defines how MPS should transform a language into another language. It is here one can define the bridge between models to actual implementation.

2.7.4 Aspects

MPS defines a language as an accumulation of *aspects*, including, but not limited to:

`Structure:` The only mandatory aspect of a language in MPS. Defines how the structure of the language's AST should be. Models within this aspect are called *concepts*.

`Editor:` Aspects that define how the concepts should be projected in the editor, effectively defining the syntax of the language.

`Constraints:` Aspects that define restrictions for the language. An example of a constraint is restricting the usage of some concepts if a certain criteria is not met [8].

`Intentions:` Defines the behaviour of the possible suggestions the IDE can provide you.

Of the mentioned aspects, `Structure` and `Constraints` have a connection to the syntax component of the language due to their roles, whereas `Editor` and `Intentions` defines usability, often associated with IDE features.

Chapter 3

Method

The aim of this project can be divided in two:

1. Implement the specification language ACT ONE.
2. Implement the IDE for our implementation of ACT ONE.

For that reason, the methods used in this project were assessed in light of both objectives. This chapter lists the said methods and the reasons behind each of them.

3.1 MPS

Implementing a language can be costly in terms of time, let alone implementing an IDE for it. Using a language workbench like MPS alleviates this cost in both areas, as languages and IDEs are implemented simultaneously in MPS [16]. Having every component of the implementation in one, accumulated MPS project makes it easier to continuously test the interoperability of the components. Moreover, MPS being a free software is another, welcomed advantage.

MPS uses a generative approach, in the sense that MPS generates compilable code from the code units created by the user. By default, the generated code is in Java, but MPS can also generate code into C, LaTeX, JavaScript and more. Generation of Java code is used in this project due to the fact that MPS is more stable with Java [16].

3.2 Requirements Analysis

To keep the project within a consistent course and not stray away from the scope at hand, the project was bound to *requirements* that we outlined ourselves. When

defining these requirements, we used the philosophies on requirements proposed by Bass [5, ch. 4.1] as the reference point. While he present these philosophies in light of software architecture, the same philosophies serve as a sufficient method to set empirical expectations for this project, warranting objective discussions for the future chapters.

3.3 Test-Driven Development

Although following any conventional software development methodology is not a necessity to arrive to our conclusion, it greatly influences the process of the implementation, and therefore deserves a mention. For the implementation part of the project, we followed some of the philosophies found in *Test-Driven Development* (TDD), a methodology realised by Beck [7]. A version of TDD follows this sequence of steps:

1. Add a test that tests a certain aspect of your system. Make sure that this test is correct and does not break any of the existing tests.
2. Write some production code where your aim is to pass the test added in step 1.
3. Refactor the code, readying for future tests.
4. Repeat steps.

We went with a less restrictive version of this methodology, as the aforementioned steps imply that additional production code should only be written with the intention to pass a certain test. We allowed additional production code to be added without having a corresponding test as writing a suitable test can, in some cases, be more challenging than writing the actual solution for it.

As mentioned earlier, MPS can generate Java code, however, the generation plan must be defined beforehand by the user for MPS to generate. One of the two objectives mentioned earlier, namely "implementing the specification language ACT ONE", can be further divided into two:

1. Implement the syntax and semantics of ACT ONE.
2. Implement the executable term rewriting logic.

This further division of objectives reveals an important fact of this project, specifically, that the syntax and semantics of ACT ONE are independent from the term rewriting logic. As mentioned in the theory chapter, the concept of term rewriting is not exclusive for ACT ONE, but shared among specification languages.

3.3.1 Unit Tests

Due to the importance of correctness of the term rewriting logic, testing it in multiple ways was vital. This was accomplished by creating a draft of the term rewriting logic, independent from MPS and ACT ONE. This draft was written in pure Java. Alongside with the draft, we created multiple unit tests with the JUnit framework [6], each testing a certain aspect of the term rewriting logic. When the draft reached a satisfactory level, we imported it to the MPS project. Due to how similar BaseLanguage and Java is, we could import the final version of the draft to MPS without too many alterations.

3.3.2 Test Cases

In the later stages of the project the MPS project became an accumulation of three components:

1. The specification language ACT ONE.
2. The IDE for our implementation of ACT ONE.
3. The rewrite system.

To test correctness of this accumulated project, we tested it against various test cases, ranging from simple specifications to more convoluted specifications. Following the same principles as earlier, we introduced more test cases incrementally, gradually increasing in complexity. More elaboration of these test cases and its results will find place in the coming chapter.

Chapter 4

Result

4.1 Requirements

4.1.1 Functional Requirements

Functional Requirements are expectations of the system's behaviour, how it *must* respond to certain run-time stimuli. Our functional requirements are listed in Table 4.2.

Code	Full
ACT	ACT ONE Syntax and Semantic
IDE	IDE Features
PRE	Predefined Features
RED	Reduction Features

Table 4.1: All code abbreviations.

Code	#	Must	Description
ACT	1	YES	Implement Prefix Operations
ACT	2	NO	Implement Infix Operations
ACT	3	NO	Let-Declaration
ACT	4	NO	If-Then-Else
ACT	5	NO	Allow specification extension
ACT	6	NO	Implement Operator-Precedence
ACT	7	NO	Implement Associative Property
ACT	8	NO	Implement Commutative Property
ACT	9	YES	Support UTF-8-compatible Syntax
ACT	10	NO	Implement Subsort
ACT	11	NO	Implement Universal Sort
ACT	12	NO	Parameterised Specification
IDE	1	NO	Overloaded Operations Check
IDE	2	NO	Duplicate Operations Check
IDE	3	NO	Parameter Type Check
IDE	4	NO	Parameter Number Check
PRE	1	YES	Predefined Boolean
PRE	2	NO	Predefined Integer
PRE	3	NO	Predefined List
RED	1	YES	Step-Reduction
RED	2	YES	Exhaustive Reduction
RED	3	YES	BFS Reduction
RED	4	YES	DFS Reduction
RED	5	NO	Tuple Reduction
RED	6	NO	Infinite-Reduction-Handler

Table 4.2: Our functional requirements. Code abbreviations is listed in Table 4.1. The column *Must* indicates if the the functional requirement is mandatory (YES) or optional (NO).

4.1.2 Quality Attributes Requirements

Quality Attributes Requirements are expected qualifications of the overall product. Our project are concerned with, mainly, the following two:

Correctness (Primary): How *correct* term reductions in our system are, is perhaps the most important concern of our project. If you think of our term reduction as a function, the correctness is the number of input terms minus the number of expected output terms; Zero being the perfect score.

Usability (Secondary): [5] describes *Usability* as follows: "How easy it is for the user to accomplish a desired task (...)". In our case, usability is the usability of the IDE.

4.1.3 Constraints

The constraints for this project are connected to the scope of this project, of which we elaborated in the beginning of this thesis. We reiterate it again: The end-product must be created in MPS.

4.2 Implementation

As mentioned in the previous chapter, our implementation can be divided in three components:

1. The specification language ACT ONE.
2. The IDE for our implementation of ACT ONE.
3. The reduction system.

Before we explore more about each of these three components, we must reaffirm the fact that these three components are, in fact, interconnected as one in MPS. This fact is vital to understand how the implementation actually operates.

4.2.1 ACT ONE

A language can be divided into smaller syntactical components, each having a relation to one or more components. When designing a language in MPS, designating these components and their relations are what makes a language. These syntactical components are called *concepts* in MPS. Like how we differentiate between classes and objects in Java, we must differentiate between *concepts* and *nodes* in MPS. Concepts are templates that describe properties and relations, whereas nodes are actual instances of a specific concept. In other words, concepts are static components, while nodes are run-time instances.

In our implementation of ACT ONE, the concepts form an intricate structure, as seen in Appendix B. This section will explain this structure in segments to allow more precise explanations where we can highlight certain concepts and their role in the language.

INamedEntity

In many programming languages, one can declare a variable and later reference back to it by its name. The possibility to identify nodes by name is an important trait in MPS. Enabling names for concepts, and implicitly its derived nodes as well, is achieved by implementing the inbuilt, interface concept `INamedEntity`. *Interface concepts* are *abstract* in MPS, meaning that no nodes can be instantiated from

them directly. Like Java, all future descendants of a concept that implement the said interface will inherit the naming trait as well. This interaction is illustrated in Figure 4.1.

Figure 4.1: In this example `Concept_A` is implementing `INamedEntity` and therefore gains the naming property. Since `Concept_B` extends `Concept_A` it will inherit this property as well.

Specification

The structure as seen in Figure 2.2 establishes a good foundation for the corresponding concept `Specification` in our implementation, as seen in Figure 4.2. Note that the notion of signature is removed whereas the variable is detached from axioms, resulting a shorter hierarchy. Moreover, all concepts except for `Axiom` implements `INamedEntity`.

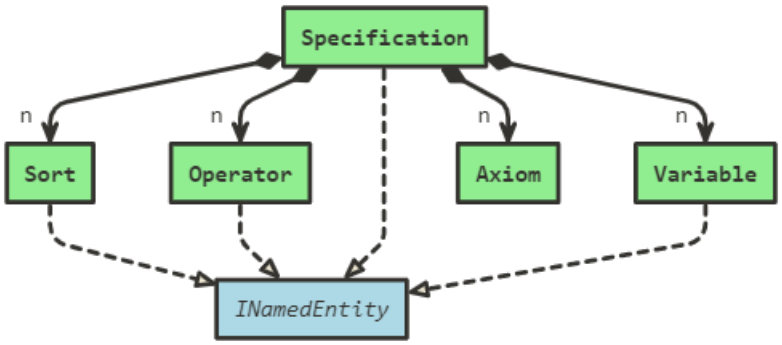


Figure 4.2: The `Specification`-concept and its relations to various concepts.

Operation

The structure for the `Operation`-concept, as seen in Figure 4.3, is derived from the Definition 2.3.1. The `Operation`-concept have input-sorts and an output-sort which are, in actuality, references to some existing sorts. Such references are represented with an intermediate which, in this case, is the `SortRef`-concept. Although these concepts are of the same type in MPS, this thesis will make a distinction between concepts in according to their role in the language. Concepts like `SortRef` will henceforth be known as *reference concepts*, while concepts like `Sort` will be just *concepts*.

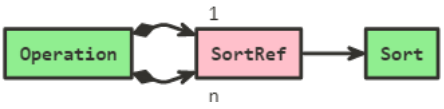


Figure 4.3: The `Operation`-concept and its relations. A `SortRef`-concept represents the relation between the `Operation`-concept and `Sort`-concept.

Variable

Similar to the `Operation`-concept, the `Variable`-concept is realised through the usage of the reference concept `SortRef` as well, as seen in Figure 4.4.

Figure 4.4: The `Variable`-concept and its relations. A `SortRef`-concept represents the relation between the `Variable`-concept and `Sort`-concept.

Axiom

Definition 2.3.3 states that an axiom is an equation $L = R$, where L and R are terms. The parent-child-relation between an axiom and its two terms is reflected in the implementation, as seen in 4.5. As illustrated in the figure, `ITerm` is an interface concept. The reason for this will be elaborated in the next section.

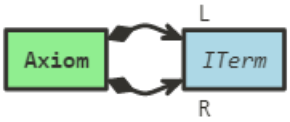


Figure 4.5: The `Axiom`-concept and its two `ITerm`-concepts. `ITerm` is an interface concept.

ITerm

`ITerm` is an interesting concept due to its recursive nature; a term can be parent of multiple terms. Instantiating an `ITerm` directly should not be possible due to the ambiguity a term presents; a term is either an operation call or a reference to a variable. Hence `ITerm` being an interface concept. This ambiguity leads to the structure seen in Figure 4.6.

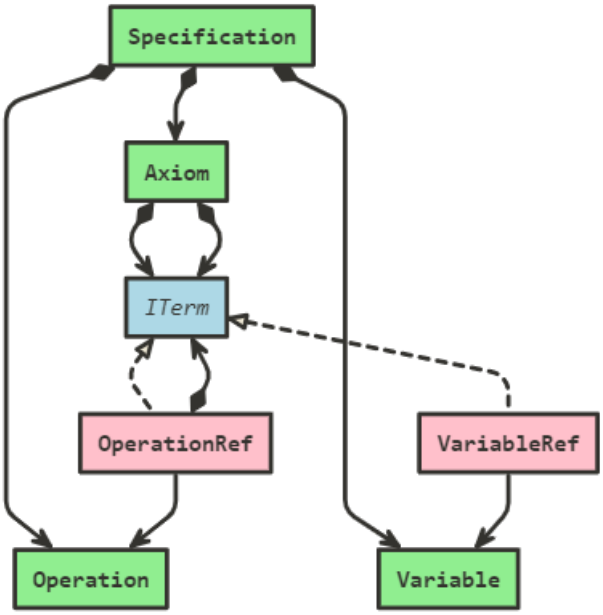


Figure 4.6: An extended version of Figure 4.5 showing the `ITerm`-concept and its associated relations. The cyclic relation between the `OperationRef`-concept and the `ITerm`-concept enables the recursive trait of `ITerm`.

Imports

As there are no limit for how big specifications can become, an effective way to organise them is a valuable feature. A such feature in our implementation is the notion of *importing*: specifications can import other specifications. This feature gives the users an incentive to import smaller specification into one, accumulated specification. This feature is realised with the reference concept `SpecificationRef` with the `Specification`-concept, as seen in Figure 4.7.

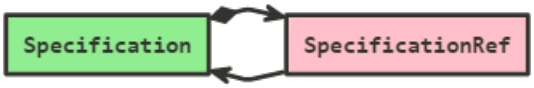


Figure 4.7: The cyclic relation between `Specification` and `SpecificationRef` enables specifications to import other specifications.

IAction

Outside of defining a specification, there are two things users can do with the said specification:

1. define and store terms, realised by the concept `Let`.
2. initiate term rewriting, realised by the concept `Reduce`.

Both actions are realised by implementing the interface concept `IAction`, as seen in Figure 4.8.

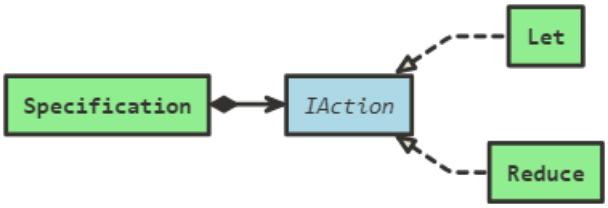


Figure 4.8: `Let` and `Reduce` are implementing the interface concept `IAction`. The concept `Specification` can parent multiple concepts that implements `IAction`.

Reduce

The *reduction*-action can be regarded as an unary operation that takes in a term and returns the reduced version of it. Figure 4.9 illustrates this how rewriting is implemented in the conceptual level. Why the name of the concept is `Reduce` instead of `Rewrite`, is due to prior re-factoring mishap. From the figure alone, one can infer that the concept `Reduce` parents an `ITerm`, namely the input-term, but not that it outputs a term. The fact that reduction outputs a term cannot be illustrated in conceptual-level. It is, however, handled in run-time, when the generated code is executed.

Figure 4.9: `Reduction` parents an `ITerm`, the input-term. `Reduction` is one of two implementations of `IAction`, the second being `Let`.

Let

Let-declarations works in similar veins to how variables in Java works. Both allow declaration by name, assignment of a value, and usage by reference. One can assign any term to a let-declaration and later refer back to it by name. As mentioned in previous section, a reduction outputs a term. This means that a reduction's output can be assigned to a let-declaration as well. This does not, however, imply that `Reduce` is an implementation of the interface concept `ITerm`. Instead, it implies

that `Reduce` can be a child of `Let`, just like how `ITerm` can be. This leads to the necessary intermediary `IStorable`, as seen in Figure 4.10.

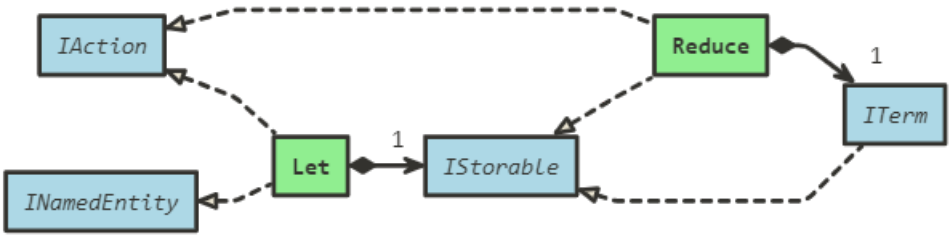


Figure 4.10: Extending Figure 4.8 and Figure 4.9 results this diagram of `Let`. The interface concept `IStorable` is introduced to support assignment of both `ITerm` and `Reduce`.

To allow assigned let-declarations to be referenced and used as a term, an additional concept must be introduced, namely the `LetRef`, as seen in Figure 4.11.

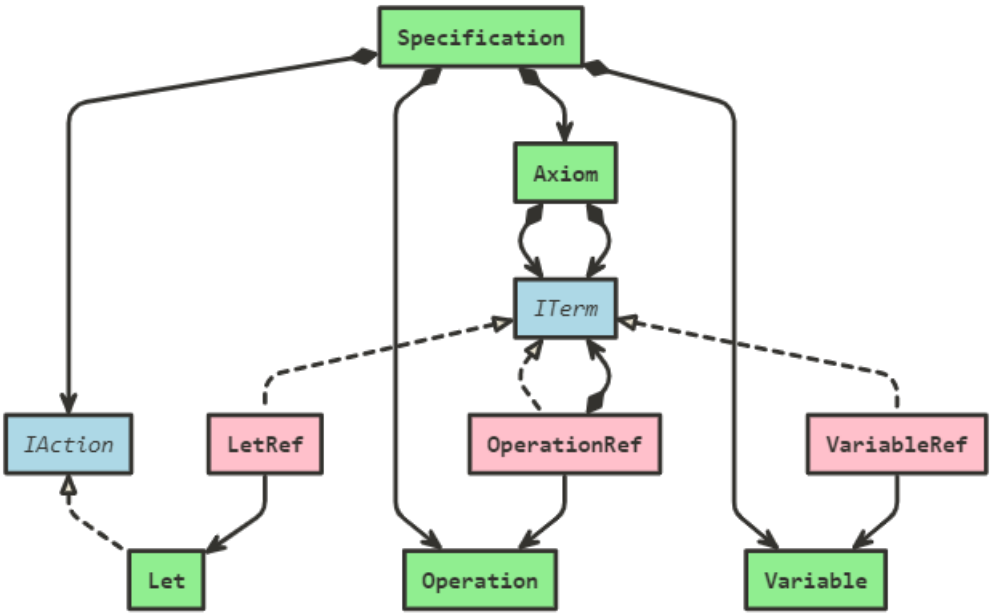


Figure 4.11: Extending Figure 4.6 brings the role of `LetRef` to light. Since `LetRef` implements `ITerm`, all instances of `LetRef` can be used as if it was any term.

Syntax

Given all the above concepts, a syntax for the language can be derived, as seen in Listing 4. Note that this is how the syntax looks like in plain-text. How the syntax is projected by the IDE will be different, something that will be elaborated in future sections.

```
type NAT is
imports: BOOLEAN
sorts
    new Nat
operators
    ctor: O   :          -> Nat
    oper: S   : Nat      -> Nat
    oper: add : Nat Nat -> Nat
variables
    var m: Nat
    var n: Nat
axioms
    axiom: add (O(), n)  = n
    axiom: add (S(m), n) = S(add(n, m))
reductions
    reduce: add (S(O()), O())
    reduce[BFS]: add(add(S(O()), S(O())), S(S(O())))
    let a = S(O())
    let b = S(S(O()))
    let c = add(a, b)
    reduce: c
```

Listing 4: An example of the syntax of our implementation of ACT ONE.

4.2.2 IDE

Due to how MPS-projects operate, the implementation of our IDE is intertwined with the implemented language ACT ONE. In fact, in MPS a language-module is both the language definition and the IDE at the same time. All concepts elaborated in previous section therefore form the IDE of our project, which is depicted in Figure 4.12. This section will list out the features our IDE has to offer.

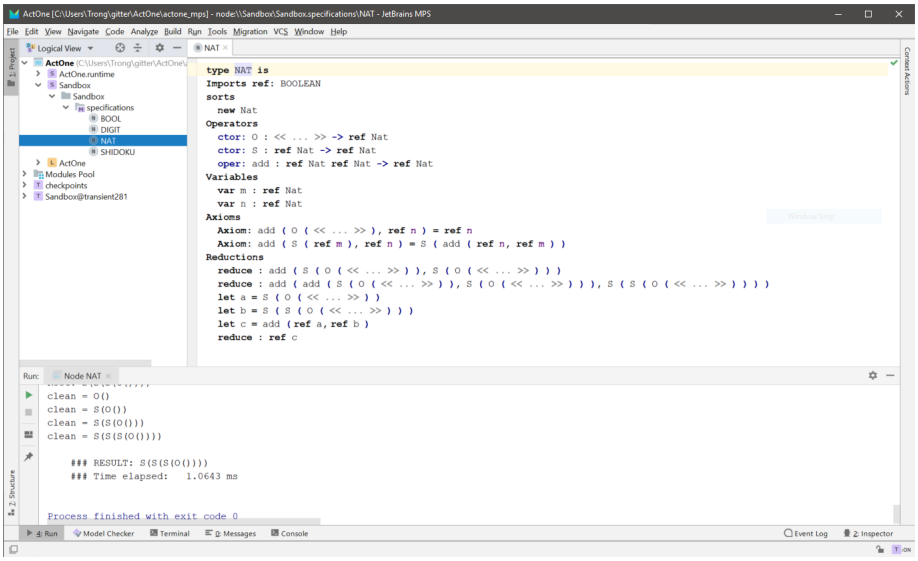


Figure 4.12: A view users of our implementation can expect to see. The bar to the left displays all specifications in current project, whereas the window to the right is the editor. The window at the bottom is the console which handles the output messages produced during an execution.

Declaration Template

A way the IDE assists users is by auto-generating declaration templates on command, like the one in Figure 4.13. This feature is available for all editable fields in the editor. For fields where multiple templates are available, the user must either input the desired template's name or choose it from the template menu, as shown in Figure 4.14.

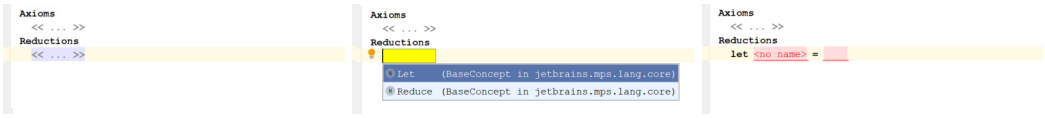
```
type ExampleSpecification is
Imports << ... >>
sorts
  << ... >>
Operators
  << ... >>
Variables
  << ... >>
Axioms
```

(a) Before

```
type ExampleSpecification is
Imports << ... >>
sorts
  << ... >>
Operators
  oper: <no name> : << ... >> -> ref <no reference>
Variables
  << ... >>
Axioms
```

(b) After

Figure 4.13: Pressing Enter when the caret is in state (a) will transition to state (b).

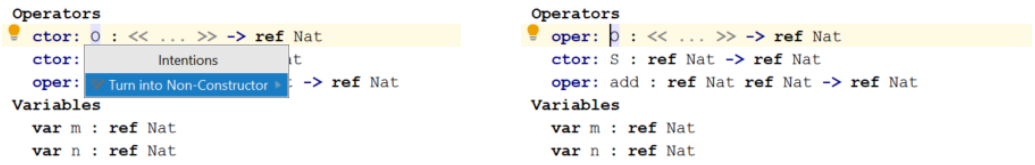


(a) Caret in position **(b)** Template menu **(c)** Selected template

Figure 4.14: Pressing Ctrl + [ ] when the caret is in state (a) opens the menu as shown in (b). Selecting one of the templates with Enter applies it into the editor, as shown in (c).

Intentions

Sometimes the IDE suggests a course of action, be it to optimise the specification or to fix an error. These suggestions are called *intentions* in MPS [LINK Volume 1]. The user is notified with a light-bulb symbol whenever a suggestion occur, as shown in Figure 4.15. The current version of the IDE has an intention for toggling an the constructor-property of an operation and another for toggling the mode of term reductions.



(a) The Intentions Menu **(b)** The result after accepting the suggestion

Figure 4.15: You can check available intentions either by clicking the light-bulb symbol or by pressing Alt + Enter , as seen in (a). Figure (b) shows the result after the selected intention was applied.

Static Error Checking

Like many IDEs, our IDE has a detection system for static semantic errors. Leaving required fields blank or using invalid names as identifiers are some errors the IDE can detect. When something invalid is detected, the user is warned about this by red highlighting, as seen in Figure 4.13b. If the user proceeds to rebuild the project without fixing these errors, a pop-up window will appear during the rebuilding process, as shown in Figure 4.16.

Executable Specifications

The IDE supports execution of specifications, as shown in Figure 4.17. This will initiate all declared actions of the executed specification and print out the results in the output console, as seen in Figure 4.18. Executing a specification is how we initiate term reductions.

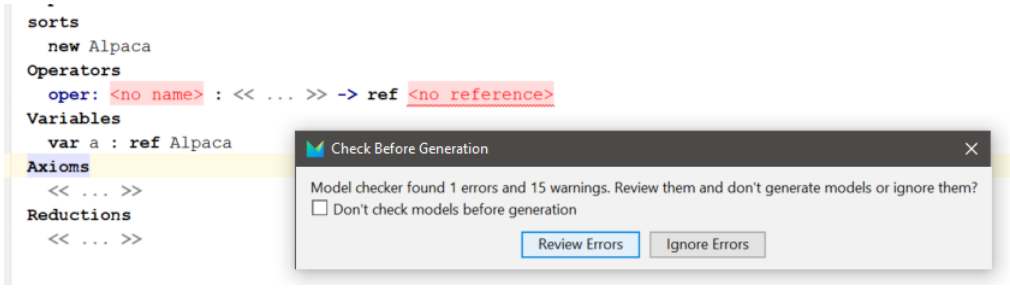


Figure 4.16: A pop-up window will appear during rebuilding if errors were detected.

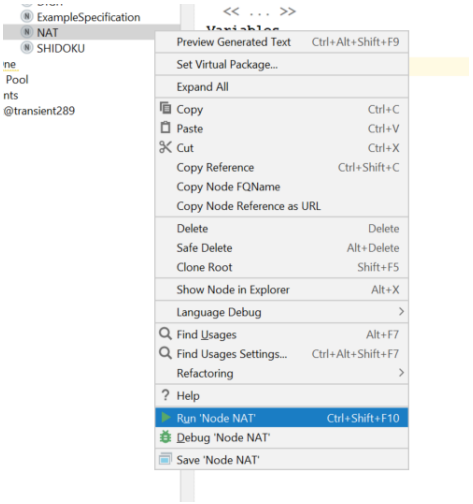


Figure 4.17: Right-clicking a specification in the *project window* prompts a menu where the user can choose to run the chosen specification.

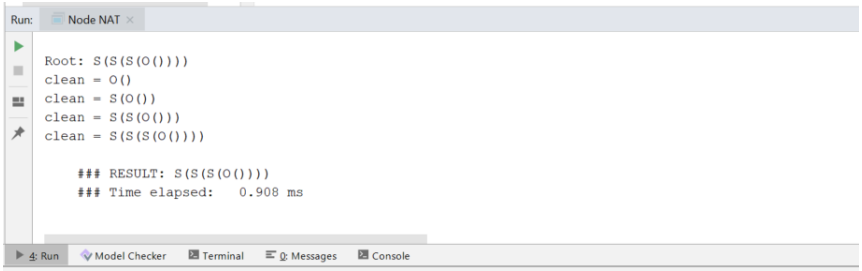


Figure 4.18: The output console after an execution.

4.2.3 Reduction System

The *Reduction System* is the component of our implementation that handles all term reductions. In reality, the reduction system is a set of rules that defines how the generator should generate Java code. The *reduction system* of our implementation is realised by the MPS generator. When a specification is successfully rebuilt, the generator generates the corresponding Java code for the specification, as illustrated in Figure 4.19.

Executing a specification simply means that its corresponding Java code that gets executed.

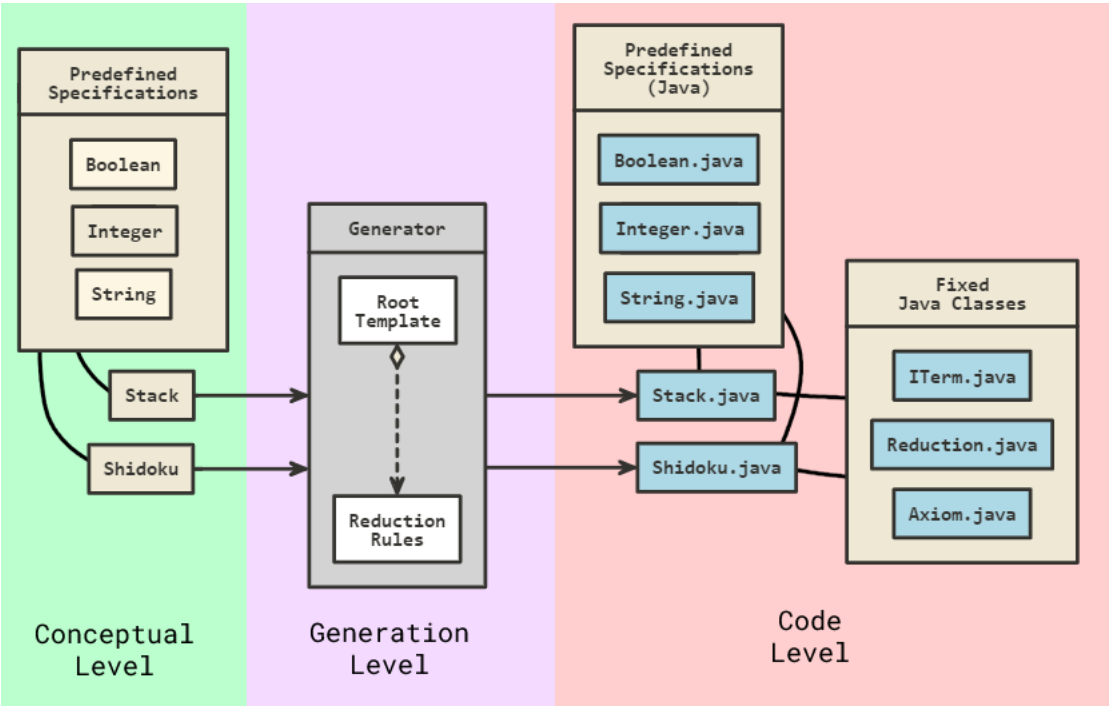


Figure 4.19: An example showing how user-made specifications (`Stack` and `Shidoku` in this example), undergo a generation process, whereas predefined specifications do not.

BFS and DFS

Our implementation supports two versions of term rewriting: *DFS* and *BFS*. These versions determines the order a term reduction will be applied, as elaborated in Section 2.5.3. In editor-level, users can toggle a term rewriting's mode via intentions, as seen in Figure 4.20. By default, a rewriting declaration is set to DFS. Depending on which mode that are selected, the generator will follow suit and generate the correct code.

```
reduce : ref c
reduce [ BFS ] : ref c
```

Figure 4.20: How different versions of term rewriting looks like in the editor.

Step-rewriting and Exhaustive-rewriting

Although these two exist in code, this feature is not available in the editor, making it impossible for users to use them.

4.3 Test Results

4.3.1 Unit Tests

The unit tests elaborated in previous chapter can be seen in Appendix C. These unit tests were made for the rewrite system before the code for it was copied over to MPS. During the time the unit tests were still relevant, that is, before it was moved over to MPS, the rewrite system passed all the unit tests.

4.3.2 Test specifications

To test the implementation when it was all in MPS, we created specifications as test cases, as seen in Appendix D. The `DIGIT`-specification from Figure D.1 did not have anything except declarations of `Sort` and `Operations`. This test case was made to check if the editor allowed simple specifications. The editor allowed this test case without any error, and thus passing the test case. `NAT`-specification from Figure D.2 introduced test cases for rewriting. This test case had some issues with rewriting whether it involved imported specifications. The last specification, `SHIDOKU` from Figure D.3, tried every declarations but in a bigger scale. This test case was left unfinished, and thus did not test anything in the end.

Chapter 5

Discussion

Like how the previous chapter explored the implementation as three, interconnected components, the discussions in this chapter will be put down in similar manner.

5.1 ACT ONE

5.1.1 Versus the original ACT ONE

This section will make the comparison between our implementation versus the original version.

Syntax

It is clear that the syntax differs between the two versions by just looking at examples from both. This difference is intentional. One of our gripes with the original ACT ONE is its verbose syntax for specifications, something we have tried to simplify without losing context. The reason why this is doable in our implementation and not in his, is due to mainly two reasons:

1. Specifications exist in the same scope in our implementation by default, contrasting the original ACT ONE's version where this must be explicitly done.
2. The original ACT ONE's version have more possibilities for specifications. One being the possibility to rename the contents of an imported specification, adding a new layer of verbosity.

Universal Sort

Our implementation includes the universal sort `Any`, a special sort that conforms with any sort. The original version does not include an equivalent concept to universal sort, which begs the question: is it a good addition for ACT ONE? An argument

against it, is that an universal sort is an exception when compared to other sorts; it is a sort that represents multiple sorts. Exceptions are, as we would put it, actual instances of inconsistency. Introducing inconsistency to ACT ONE may hurt its integrity. An argument for it, is that it adds a new layer of expressiveness that may allow alternative solutions for various problems.

Parameterised Specifications

The original ACT ONE introduces the concept of *Parameterised Specification*. A parameterised specification contains generic sorts or/and operations that can be instantiated (or actualised) with a sort. Another take on it, parameterised specifications behave in similar fashion as Generics in Java do. Our implementation does not support parameterised specifications, making all our specifications *unparameterised* by default. In practice, this makes it difficult, if possible at all, to define specifications such as `List[P]` or `Stack[P]` that contain `P`-sorted elements.

In our current implementation there exist two ways to make up for this absence, albeit with associated drawbacks:

1. If you wish to emulate, say, the specification `List[P]` of `P`-sorted elements, you can use the inbuilt sort `Any` as the placeholder for any sort, resulting the specification `List` of `Any`-sorted elements. The drawback with this method is that static type-checking will not be possible. In other words, adding a `Bool`-sorted element to a `List` of, supposedly, `Int`-sorted elements can only be caught during run-time. This is because both `Bool` and `Int` are `Any`-sorted.

2. You can define unique specifications with fixed sorts in place of parameterised sorts. In light of the previous scenario, you can define a specification `IntList` containing `Int`-sorted elements exclusively. A drawback here being the fact that you must define a specification for each permutation of fixed sorts.

The reason why parameterised specification is not implemented in our version, is due to the cost and the additional complexity it introduces. Another reason being that it is not needed in the course DAT233, making it just optional in our context.

Structuring Mechanisms

The original ACT ONE proposes four different mechanism of specification interoperability, coined *Structuring Mechanisms*. In simpler terms, these mechanisms are methods that promotes any form of composition between two specifications. These mechanism has in common that they all must be declared within a specification. Summarised, the usage for these mechanisms entails the following:

`Extension:` Extends an existing specification with the current specification.

`Union:` Merging two existing specifications with each-other.

`Actualisation:` Instantiate a parameterised specification with sort(s) from another specification.

`Renaming:` Rename the contents of an existing specification.

Our `import` is an implementation of the original ACT ONE's *Extension*-mechanism. Although implementing all four mechanisms would allow maximum expressiveness, having all would increase the complexity of the implementation. Higher complexity may damage the guarantee of correctness our implementation has, a risk we are not willing to make.

Sort Hierarchy

In an earlier iteration of our implementation, the concept of *Subsort* existed, as seen in Figure 5.1. This concept introduced the possibility for users to define the relation between two sorts as a parent-child-relation, effectively creating a hierarchy for sorts. Sort hierarchy is not elaborated in Ehrig's ACT ONE.

Like some features, the addition of sub-sorts would increase the level of expressiveness. However, it functioned as a catalyst for future problems, as everything involving sorts had to be adjusted to accommodate it. This being a risk to the correctness resulted its removal in a later iteration.

```
type NAT is
Imports ref: BOOLEAN
sorts
  new Nat
  new Odd
  new Even
subsorts
  Subsort Nat > Odd
  Subsort Nat > Even
Operators
  ctor: 0 : << ... >> -> ref Nat
```

Figure 5.1: The concept of sub-sort existed in an earlier iteration of our implementation.

Even though our ACT ONE does not support new subsorts in specifications, the hierarchy of sorts still exists. Recall that our implementation has `Any`, the universal sort. In practice, `Any` is parent of all other sorts, which results a hierarchy. Since this is a fixed relation, contrasting the aforementioned concept of subsort, we deemed it secure enough to remain in our current iteration.

Infix Notation

Due to time restrictions, we did not implement support for Infix Notation. This is another feature the original ACT ONE has that our version does not.

Comments

Our implementation has no support for *in-line comments* within specifications, a concept the original ACT ONE has. This is yet another feature that is optional and has been abandoned due to time constraints.

5.2 IDE

5.2.1 Usability

One of the advantages with using MPS is its integration with IntelliJ Platform. Our implementation can achieve similar features that other IDEs built on the IntelliJ Platform have. In previous chapter we elaborated about features like declaration templates and intentions, two features that make the editor more user-friendly. Having IntelliJ Platform as our stepping stone makes such features possible.

As elaborated earlier, the editor works with models rather than text. In our case, this presents itself as a double-edged sword. It paves the road for the aforementioned features enabling a better user experience with the editor. In the same time, it may make the editor feel restrictive to use. The projectional editor stands in contrast to plain text editors, where one can change the formatting of the text. It is possible for the projectional editor to mimic the feel of plain text editors to some degree, but due to how the editor in MPS is tightly linked to models, it will likely never achieve such feel.

5.3 Reduction System

5.3.1 Test cases

Given the results of the test cases elaborated in the previous chapter, we can determine that rewriting does not always work, depending on if it involves imported specifications or not. This is a breach on correctness, which, given that it is one of our quality attribute requirement, is not good. That the specification `DIGIT` from Figure D.1 worked without any errors detected is a comforting fact, as just having a working IDE to specify non-executable specifications is a good feature as well.

5.4 Alternatives

Implementation of the language

Alternatives to MPS can be considered. For the implementation of the language, an interpreter for it can be implemented from scratch instead. An advantage with this approach is that it bypasses the abstraction level language workbenches like MPS bring, and thus allow more optimal interpretation. However, lower abstractions will likely increase the difficulty of the implementation, and in result affect the time budget.

Implementation of the IDE

If implementing an IDE for ACT ONE is the main objective, there are two approaches that springs to mind. Firstly being to create an IDE from ground-up, and secondly being to extend an existing software in lieu of starting from scratch. The first approach allows a great range of freedom but at the expense of time. This cost alone is why the first approach was not chosen for this project. The second approach, however, is less costly. One can extend an existing text editor or an IDE into an more suited IDE for ACT ONE. An example here being Microsoft's *Visual Studio Code*, a text editor which can be modified through the inclusion of extensions [12]. An extension that brings features like auto-completion and syntax checker for ACT ONE would in effect result an editor akin to an IDE for ACT ONE. However, both alternatives still require you to implement an interpreter for ACT ONE.

Chapter 6

Conclusion

The aim of this project was to use MPS to implement a system to work with the algebraic specification language ACT ONE, tailored towards the course DAT233 held at the University of Agder. The final product is an IDE that caters to a modified ACT ONE and its rewrite system. The implemented ACT ONE differs from the original ACT ONE in multiple aspects: For starters, our version has fewer features due to various reasons. Many concepts described in the original ACT ONE were deemed optional in the requirement analysis. As our project was concerned about the correctness of the overall product, less features in some cases were preferred, as this lowered the complexity, and thus making it easier to assure correctness.

As for the IDE, more time could have been spent improving it. In its current version, it provides just enough for the other components to work, and therefore lacks some additional features for itself. As most of the requirements involving the IDE were optional, these were put aside for more prioritised requirements, like the ones involving the rewrite system.

Implementing the rewrite system was allocated more time than the other two components as it required a more intricate implementation, let alone being dependent on the other two components. Although the current rewrite system supports multiple working rewriting options, some of these options remain unreachable from the user as the current version of IDE cannot present these options in the editor.

For future work, improving the IDE would be a priority. Regardless of how well defined the ACT ONE language and rewrite system are, a sub-par IDE will overshadow them due to it being the face of our system. The rewrite system needs some improvements too: Adding tests to assure its correctness would be a good way to increase the overall trust between the user and the system. As for the language ACT ONE, more predefined functionalities would be preferred, as this would make it more user-friendly. Some predefined functionalities that springs to mind here are the complete support for `Integer` and `List`, to make it more similar to existing programming languages.

Other than that, the final product works as intended. It allows users to specify specifications and perform term rewriting with them. It is accessible in the sense that the all components are coupled together under a single MPS project. For these reasons, the final product, although flawed, provides sufficient features to warrant its usage in future lectures in DAT233.

Bibliography

[1] *Abstract data types*. 2019. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/ADT.html.

[2] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Addison Wesley, 2007. ISBN: 0321486811.

[3] *Axiom*. 2019. URL: https://en.oxforddictionaries.com/definition/axiom.

[4] *BaseLanguage*. 2019. URL: https://www.jetbrains.com/help/mps/base-language.html.

[5] L. Bass, P. Clemens, and R. Kazman. *Software Architecture in Practice: Third Edition*. Pearson Education, 2013. ISBN: 978-0-321-81573-6.

[6] S. Bechtold et al. *JUnit 5 User Guide*. 2019. URL: https://junit.org/junit5/docs/current/user-guide/.

[7] K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN: 9780321146533.

[8] F. Campagne. *The MPS Language Workbench, Vol. 1*. CreateSpace Independent Publishing Platform, 2014.

[9] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development: The ACT Approach*. World Scientific Publishing, 1993. ISBN: 981-02-1227-5.

[10] J. De Meer, R. Roth, and S. Vuong. "Introduction to algebraic specifications based on the language ACT ONE". In: *Computer networks and ISDN systems* 23.5 (1992), pp. 363–392.

[11] H. Ehrig and B. Mahr. "Fundamentals of Algebraic Specification 1". In: Springer-Verlag, 1985.

[12] *Extension Marketplace*. 2019. URL: https://code.visualstudio.com/docs/editor/extension-gallery.

[13] M. Gramner. "A Study in Formal Specifications". Bachelors Thesis. Umeå University, 2005.

[14] R. M. Hierons et al. "Using Formal Specifications to Support Testing". In: *ACM Comput. Surv.* 41.2 (Feb. 2009), 9:1–9:76. ISSN: 0360-0300. DOI: 10.1145/1459352.1459354. URL: http://doi.acm.org/10.1145/1459352.1459354.

[15] J. W. Klop and J. Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[16] *MPS Concepts*. 2019. URL: https://www.jetbrains.com/mps/concepts/.

[17] *MPS Homepage*. 2019. URL: https://www.jetbrains.com/mps/.

[18] *MPS project structure*. 2019. URL: https://www.jetbrains.com/help/mps/mps-project-structure.html.

[19] E. A. Santos et al. "Syntax and sensibility: Using language models to detect and correct syntax errors". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 311–322.

[20] R. W. Sebesta. "Concepts of Programming Languages, Global Edition". In: 11th ed. Pearson Education, 2016. ISBN: 9781292100555.

[21] B. Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25.

[22] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201656973.

[23] *Specification*. 2019. URL: https://en.oxforddictionaries.com/definition/specification.

[24] J. V. Tucker and J. I. Zucker. "Abstract Computability and Algebraic Specification". In: *ACM Trans. Comput. Logic* 3.2 (Apr. 2002), pp. 279–333. ISSN: 1529-3785. DOI: 10.1145/505372.505375. URL: http://doi.acm.org/10.1145/505372.505375.

[25] J. Turner and M. T.L. *The construction of formal specifications: an introduction to the model-based and algebraic approaches*. 1992.

[26] S. Zilles. *Algebraic Specification of Data Types, Proj. MAC Report 11*. 1974.

Appendices

Appendix A

Preliminary Report

Refer to the separate file.

Appendix B

UML Diagram

For our implementation of ACT ONE, Figure B.1 shows the concepts of our implementation.

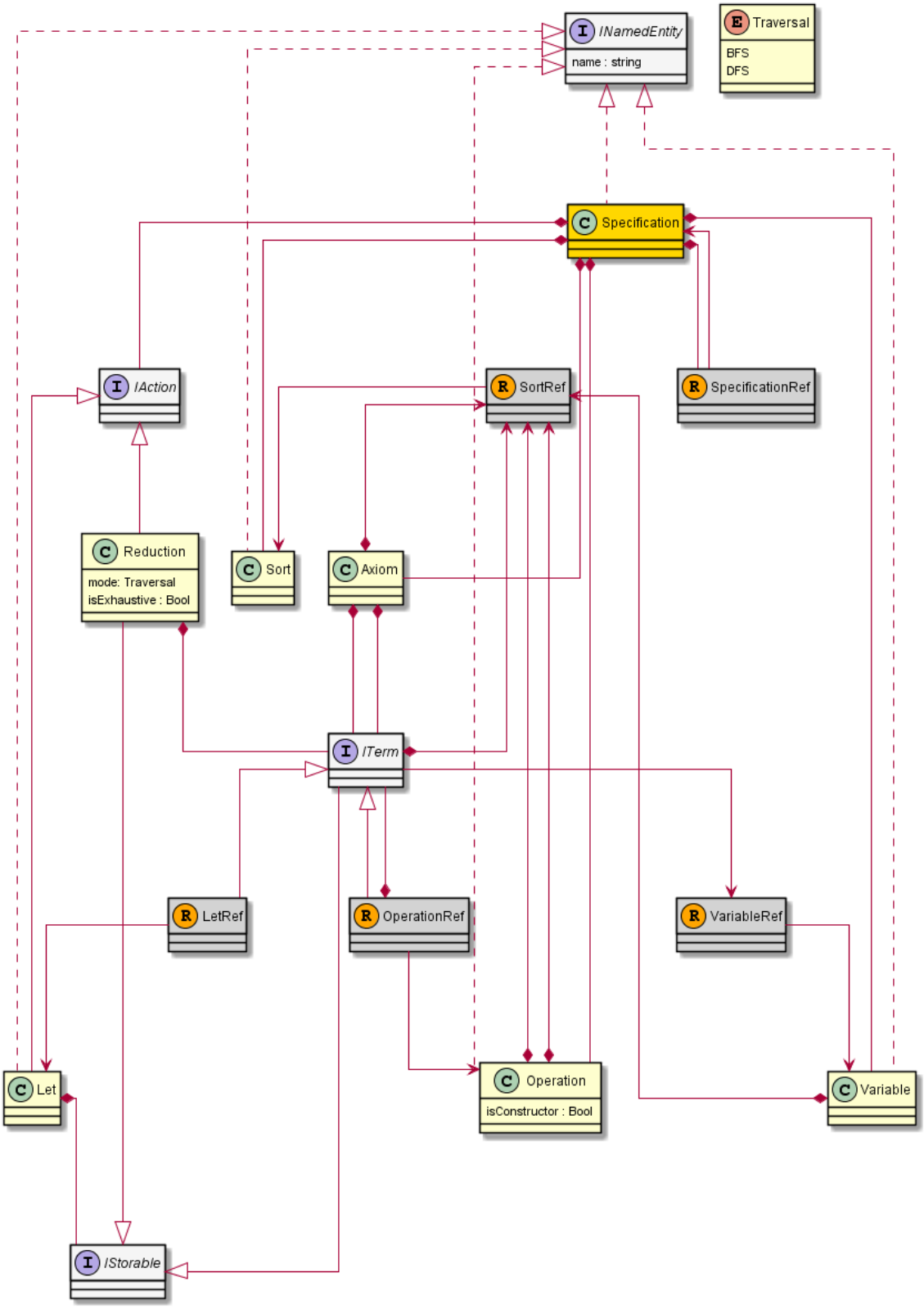


Figure B.1: The UML diagram of the structure of our ACT ONE implementation.

Appendix C

Unit Tests

Unit tests that were used to test the logic of term reduction and term equivalence.

```
1  package specifications;
2
3  import act.two.*;
4
5  import org.junit.jupiter.api.Test;
6  import static org.junit.Assert.*;
7  import static org.junit.jupiter.api.Assertions.assertEquals;
8
9  import java.util.ArrayList;
10 import java.util.Objects;
11
12 import static act.two.Reductions.*;
13 import static act.two.Reductions.Traversal.*;
14
15 import static specifications.NATURAL.Nat.*;
16 import static specifications.NATURAL.Bool.*;
17
18 class Tests{
19
20     /**
21      * Load the specification NATURAL
22      */
23     Tests() {
24         NATURAL.Initialize();
25     }
26
```

```
27      /**
28       * Test various term reductions, both BFS- and DFS Exhaustive-Reduction.
29       */
30      @Test
31      void testReductions() {
32
33          ITerm a = add(S(O()), O());
34          assertEquals(initReduction(BFS, a), S(O()));
35          assertEquals(initReduction(DFS, a), S(O()));
36
37          ITerm b = add(S(O()), S(O()));
38          assertEquals(initReduction(BFS, b), S(S(O())));
39          assertEquals(initReduction(DFS, b), S(S(O())));
40
41          ITerm c = S(S(add(add(S(O()), S(S(S(O())))), add(S(O()), O()))));
42          assertEquals(initReduction(BFS, c), S(S(S(S(S(S(O()))))))));
43          assertEquals(initReduction(DFS, c), S(S(S(S(S(S(O()))))))));
44
45          ITerm d = Equal(add(S(O()),S(O())), S(S(O())));
46          assertEquals(initReduction(BFS, d), True());
47          assertEquals(initReduction(DFS, d), True());
48
49          ITerm e = Equal(S(O()), add(S(O()),S(O())));
50          assertEquals(initReduction(BFS, e), False());
51          assertEquals(initReduction(DFS, e), False());
52
53          ITerm f = add(add(S(O()), S(S(O()))), O());
54          assertEquals(initReduction(BFS,f),S(S(S(O()))));
55          assertEquals(initReduction(DFS,f),S(S(S(O()))));
56
57          ITerm g = And(Or(True(),And(False(),True())),Or(True(),False()));
58          assertEquals(initReduction(BFS,g),True());
59          assertEquals(initReduction(DFS,g),True());
60
61      }
62
63      /**
64       * Test Breadth-First-Search Step-Reduction.
65       */
66      @Test
```

```
67      void testReduceInIterationsBFS() {
68
69          ITerm a = add(add(S(O()), S(S(O()))), O());
70          ITerm b = add(S(O()), S(S(O())));
71          ITerm c = S(add(O(), S(S(O()))));
72          ITerm d = S(S(S(O())));
73
74          assertEquals(a, initReduction(BFS, 0, a));
75          assertEquals(b, initReduction(BFS, 1, a));
76          assertEquals(c, initReduction(BFS, 2, a));
77          assertEquals(d, initReduction(BFS, 3, a));
78          assertEquals(d, initReduction(BFS, 4, a));
79          assertEquals(d, initReduction(BFS, 5, a));
80          assertEquals(d, initReduction(BFS, 6, a));
81
82      }
83
84      /**
85       * Test Depth-First-Search Step-Reduction.
86       */
87      @Test
88      void testReduceInIterationsDFS() {
89
90          ITerm a = add(add(S(O()), S(S(O()))), O());
91          ITerm b = add(S(add(O(),S(S(O())))),O());
92          ITerm c = add(S(S(S(O()))),O());
93          ITerm d = S(S(S(O())));
94
95          assertEquals(a, initReduction(DFS, 0, a));
96          assertEquals(b, initReduction(DFS, 1, a));
97          assertEquals(c, initReduction(DFS, 2, a));
98          assertEquals(d, initReduction(DFS, 3, a));
99          assertEquals(d, initReduction(DFS, 4, a));
100         assertEquals(d, initReduction(DFS, 5, a));
101         assertEquals(d, initReduction(DFS, 6, a));
102
103     }
104
105     /**
106      * If no further reductions are found, the reductions should
```

```
107         * return the original term, and not a copy of it.
108         */
109        @Test
110        void testExhaustedReductions(){
111
112            ITerm term = S(S(S(O())));
113
114            assertSame(term, initReduction(BFS, term));
115            assertSame(term, initReduction(BFS, 10, term));
116            assertSame(term, initReduction(DFS, term));
117            assertSame(term, initReduction(DFS, 10, term));
118
119        }
120
121        /**
122         * Test equivalence of terms, and not the equality.
123         */
124        @Test
125        void testEquivalencyOfEquals(){
126
127            assertEquals(n, n);
128            assertEquals(S(O()), S(O()));
129            assertEquals(n, S(O()));
130            assertEquals(S(O()), n);
131
132            assertNotEquals(n, m);
133            assertNotEquals(S(O()), O());
134            assertNotEquals(O(), S(O()));
135
136        }
137
138        /**
139         * Test if copying of term is correct, as this
140         * is needed in this implementation of term reduction.
141         */
142        @Test
143        void testCopy() {
144
145            ArrayList<ITerm> cases = new ArrayList<>();
146            cases.add(O());
```

```
147             cases.add(n);
148             cases.add(add(S(0()), m));
149 
150             for (ITerm x : cases) {
151 
152                 ITerm y = x.clone();
153                 ITerm z = y.clone();
154 
155                 // Reflexive Test
156                 assertEquals(x,x);
157                 assertEquals(y,y);
158                 assertEquals(z,z);
159 
160                 // Symmetry Test
161                 assertEquals(x, y);
162                 assertEquals(y, x);
163 
164                 // Transitive Test
165                 assertEquals(x, y);
166                 assertEquals(y, z);
167                 assertEquals(x, z);
168 
169                 // Null test
170                 assertNotEquals(null, x);
171                 assertNotEquals(null, y);
172                 assertNotEquals(null, z);
173 
174                 // Equivalence check
175                 assertTrue(correctCloned(x, y));
176                 assertTrue(correctCloned(y, z));
177                 assertTrue(correctCloned(x, z));
178 
179                 // HashCode Check
180                 assertEquals(x.hashCode(), y.hashCode());
181                 assertEquals(y.hashCode(), z.hashCode());
182                 assertEquals(x.hashCode(), z.hashCode());
183             }
184         }
185 
186         /**
```

```
187         * Determines if given terms do not refer to the
188         * same object, but are equivalent.
189         *
190         * @return true if both terms are equivalent, false if not or
191         * refer to same object.
192         */
193        boolean correctCloned(ITerm a, ITerm b) {
194
195            if (a.kind != b.kind) return false;
196            if (!Objects.equals(a.name, b.name)) return false;
197
198            if (a.terms != null && b.terms != null) {
199                // Both terms are OPERATIONS
200                for (int i = 0; i < a.terms.size(); i++){
201                    if (!correctCloned(a.terms.get(i), b.terms.get(i)))
202                        return false;
203                }
204                return a != b;
205            }
206            else{
207                // Both terms are VARIABLES
208                return a == b;
209            }
210        }
211    }
```

Appendix D

Test Cases

```
type DIGIT is
Imports << ... >>
sorts
  new NonZeroDigit
  new Digit
Operators
  ctor: d0 : << ... >> -> ref Digit
  ctor: d1 : << ... >> -> ref NonZeroDigit
  ctor: d2 : << ... >> -> ref NonZeroDigit
  ctor: d3 : << ... >> -> ref NonZeroDigit
  ctor: d4 : << ... >> -> ref NonZeroDigit
  ctor: d5 : << ... >> -> ref NonZeroDigit
  ctor: d6 : << ... >> -> ref NonZeroDigit
  ctor: d7 : << ... >> -> ref NonZeroDigit
  ctor: d8 : << ... >> -> ref NonZeroDigit
  ctor: d9 : << ... >> -> ref NonZeroDigit
  oper: ToDigit : ref NonZeroDigit -> ref Digit
Variables
  << ... >>
Axioms
  << ... >>
Reductions
  << ... >>
```

Figure D.1: `DIGIT`-specification used for testing purposes.

```
type NAT is
Imports ref: BOOLEAN
sorts
  new Nat
Operators
  ctor: O : << ... >> -> ref Nat
  ctor: S : ref Nat -> ref Nat
  oper: add : ref Nat ref Nat -> ref Nat
Variables
  var m : ref Nat
  var n : ref Nat
Axioms
  Axiom: add ( O ( << ... >> ), ref n ) = ref n
  Axiom: add ( S ( ref m ), ref n ) = S ( add ( ref n, ref m ) )
Reductions
  reduce : add ( S ( O ( << ... >> ) ), S ( O ( << ... >> ) ) )
  reduce [ BFS ] : add ( add ( S ( O ( << ... >> ) ), S ( O ( << ... >> ) ) ), S ( S ( O ( << ... >> ) ) ) )
  let a = S ( O ( << ... >> ) )
  let b = S ( S ( O ( << ... >> ) ) )
  let c = add (ref a, ref b )
  reduce : ref c
  reduce : ref c
  reduce [ BFS ] : ref a
  reduce : ref c
```

Figure D.2: NAT-specification used for testing purposes.

```
type SHIDOKU is
Imports ref: BOOLEAN
sorts
  new shidoku
  new line
  new element
Operators
  ctor: v_1 : << ... >> -> ref element
  ctor: v_2 : << ... >> -> ref element
  ctor: v_3 : << ... >> -> ref element
  ctor: v_4 : << ... >> -> ref element
  ctor: v_x : << ... >> -> ref element
  ctor: mkline : ref element ref element ref element ref element -> ref line
  ctor: mkpuzzle : ref line ref line ref line ref line -> ref shidoku
  oper: columns : ref shidoku -> ref shidoku
  oper: blocks : ref shidoku -> ref shidoku
  oper: test_elem : ref element ref element -> ref Bool
  oper: test_line : ref line -> ref Bool
  oper: test_puzzleLines : ref shidoku -> ref Bool
  oper: test_puzzle : ref shidoku -> ref Bool
Variables
  var p : ref shidoku
  var l1 : ref line
  var l2 : ref line
  var l3 : ref line
  var l4 : ref line
  var e1 : ref element
  var e2 : ref element
  var e3 : ref element
  var e4 : ref element
  var e11 : ref element
  var e12 : ref element
  var e13 : ref element
  var e14 : ref element
  var e21 : ref element
  var e22 : ref element
  var e23 : ref element
  var e24 : ref element
  var e31 : ref element
  var e32 : ref element
  var e33 : ref element
  var e34 : ref element
  var e41 : ref element
  var e42 : ref element
  var e43 : ref element
  var e44 : ref element
Axioms
  Axiom:
      columns ( mkpuzzle ( mkline ( ref e11, ref e12, ref e13, ref e14 ), mkline ( ref e21,
      =
      mkpuzzle ( mkline ( ref e11, ref e21, ref e31, ref e41 ), mkline ( ref e12, ref e22,
```

Figure D.3: SHIDOKU-specification used for testing purposes.