Departament de Sistemes Informàtics i Computació

Universitat Politècnica de València

Narrowing and Unification in the Maude Programming Language

Master Thesis

Master's Degree in Software Engineering, Formal Methods and Information Systems

Author: Raúl López Rueda

Cotutors: Santiago Escobar Román
Julia Sapiña Sanchis

2020-2021

Abstract

Communications security protocols evolve every day seeking improvements and bug fixes, many of which can lead to vulnerabilities with fatal consequences. For this reason, it becomes vitally important to find algorithms and tools capable of finding these vulnerabilities in order to fix them as soon as possible. One of these algorithms is narrowing, implemented in Maude, a language used by various protocol analysis tools, which, despite being powerful and efficient in many cases, has some limitations, which can be resolved by using variants of the algorithm. A possible variant is canonical narrowing, which relies on irreducibility constraints to discard those sequences of the standard narrowing that can be ignored because they do not represent actual rewrite sequences.

Keywords: Maude, narrowing, unification, canonical narrowing, protocol.

Resumen

Los protocolos de seguridad en las comunicaciones evolucionan cada día buscando mejoras y corrección de errores, muchos de los cuales pueden llevar a vulnerabilidades con consecuencias fatales. Por ello, se vuelve de vital importancia el encontrar algoritmos y herramientas capaces de encontrar esas vulnerabilidades para poder solucionarlas cuanto antes. Uno de estos algoritmos es el *narrowing*, implementado en Maude, un lenguaje utilizado por varias herramientas de análisis de protocolos, que, a pesar de ser potente y eficiente en muchos casos, tiene algunas limitaciones, que pueden ser resueltas mediante la utilización de variantes del algoritmo. Una posible variante es el *canonical narrowing*, que se basa en restricciones de irreductibilidad para descartar aquellas secuencias del *narrowing* estándar que pueden obviarse porque no representan secuencias de reescritura reales.

Palabras clave: Maude, *narrowing*, unificación, *canonical narrowing*, protocolo.

Resum

Els protocols de seguretat en les comunicacions evolucionen cada dia buscant millores i correcció d'errors, molts dels quals poden portar a vulnerabilitats amb conseqüències fatals. Per això, es torna de vital importància el trobar algorismes i eines capaces de trobar aqueixes vulnerabilitats per a poder solucionar-les com més prompte millor. Un d'aquests algorismes és el *narrowing*, implementat en Maude, un llenguatge utilitzat per diverses eines d'anàlisis de protocols, que, malgrat ser potent i eficient en molts casos, té algunes limitacions, que poden ser resoltes mitjançant la utilització de variants de l'algorisme. Una possible variant és el *canonical narrowing*, que es basa en restriccions d'irreductibilitat per a descartar aquelles seqüències del *narrowing* estàndard que poden obviar-se perquè no representen seqüències de reescriptura reals.

Paraules clau: Maude, *narrowing*, unificació, *canonical narrowing*, protocol.

Contents

1. Introduction 9

1.1. Objectives 9

1.2. Motivation 10

1.3. Document structure 10

2. Preliminaries 13

2.1. Introduction to Maude 13

2.2. Rewrite theories 13

2.3. Sorts and subsorts 14

2.4. Variables and operators 14

2.5. Functional modules 15

2.6. System modules 16

2.7. Unification 17

2.8. Order-sorted unification modulo axioms *B* 17

2.9. Variant unification 19

2.10. Narrowing 21

2.11. Maude reflection 23

3. Generation of the reachability tree 27

3.1. Implementation methodology 27

3.2. Types, subtypes, and data structures 27

3.3. Tree unfolding 30

3.4. Variable renaming 36

3.5. Testing the tree generation 39

4. Narrowing using the reachability tree 45

4.1. Specification of the narrowing steps to implement 45

4.2. Types, subtypes, data structures and variables 45

4.3. The *getNarrowingSolutions* command and node generation 47

4.4. Unification with the target term 48

4.5. Calculation of solutions with the unifiers 50

4.6. Iteration and termination of the algorithm 53

4.7. Testing the narrowing algorithm implementation 54

5. Canonical narrowing 59

5.1. Limitations of standard narrowing 59

5.2. Introduction to canonical narrowing 60

5.3. Implementation in Maude 61

5.4. Comparison with the standard narrowing commands 63

6. Conclusions and future work 71

6.1. Conclusions 71

6.2. Future work 72

References 73

Table of figures

Figure 1: Example of types and subtypes definitions in Maude. 14
Figure 2: Example of type definition and the use of constructor symbols in Maude. 14
Figure 3: Example of variables definition in Maude. 15
Figure 4: Maude module that calculates the factorial of a number. 16
Figure 5: State diagram that models a simple rewrite system. 16
Figure 6: Basic example of a system module in Maude. 17
Figure 7: Maude Module that defines the concatenation of lists with associativity. 18
Figure 8: Unifiers returned by Maude as a solution to the unification problem seen above. 18
Figure 9: Output returned by Maude in response to an impossible unification problem. 19
Figure 10: EXCLUSIVE-OR module in Maude. 20
Figure 11: Variants of the exclusive-or symbol in Maude. 20
Figure 12: Example of a search graph in a vending machine module [4]. 21
Figure 13: NARROWING-VENDING-MACHINE module, from the Maude's official examples. 22
Figure 14: Example of using the vu-narrow command. 23
Figure 15: Definition of the metaNarrowingApply command and its related types in Maude's META-LEVEL module. 24
Figure 16: Definition of the metaVariantUnify command and its related types in Maude's META-LEVEL module. 25
Figure 17: Sorts, subsorts and operators in the GET-REACHABILITY-TREE module. 28
Figure 18: Variables declared in the GET-REACHABILITY-TREE module. 30
Figure 19: Equations that define the initial behavior of the getReachabilityTree command in the GET-REACHABILITY-TREE module. 30
Figure 20: Equation that defines the generation of the first level of the reachability tree in the GET-REACHABILITY-TREE module. 31
Figure 21: An example tree with 19 nodes labeled with letters. 32
Figure 22: Equation that defines the generation of levels higher than one of the reachability tree when the depth parameter takes unbounded as a value in the GET-REACHABILITY-TREE module. 33
Figure 23: Figure 20: Equation that defines the generation of levels higher than one of the reachability tree when the depth parameter takes a non-zero natural as a value in the GET-REACHABILITY-TREE module. 35
Figure 24: Equations to manage an empty list of nodes to process and nodes not found in the GET-REACHABILITY-TREE module. 35
Figure 25: Imports and variable declarations in the CONVERT-VARIABLES module. 36
Figure 26: The rename operator and its associated equations in the CONVERT-VARIABLES module. 36
Figure 27: The getVars operators and their associated equations in the CONVERT-VARIABLES module. 37
Figure 28: The getRangeVars operator and its associated equations in the CONVERT-VARIABLES module. 37
Figure 29: The getVarsNumber operator and its associated equations in the CONVERT-VARIABLES module. 38
Figure 30: The applySub operators and their associated equations in the CONVERT-VARIABLES module. 38
Figure 31: Example 1 of the generation of the reachability tree in the NARROWING-VENDING-MACHINE module. 40
Figure 32: Generation of the reachability tree using Narval and the NARROWING-VENDING-MACHINE module. 40
Figure 33: Example 2 of the generation of the reachability tree in the NARROWING-VENDING-MACHINE module. 41
Figure 34: The SUCC-PRED module in Maude. 42
Figure 35: Example 1 of the generation of the reachability tree in the SUCC-PRED module. 42
Figure 36: Generation of the reachability tree using Narval and the SUCC-PRED module. 43
Figure 37: Example 2 of the generation of the reachability tree in the SUCC-PRED module. 43
Figure 38: Sorts, subsorts and related operators in the GET-NARROWING-SOLUTIONS module. 46
Figure 39: Variables declared in the GET-NARROWING-SOLUTIONS module. 47
Figure 40: Equations that define the initial behavior of the getNarrowingSolutions command in the GET-NARROWING-SOLUTIONS module. 48

Figure 41: Start of the unification step when the search arrow is =>+ in the GET-NARROWING-SOLUTIONS module. 49
Figure 42: Start of the unification step when the search arrow is =>* or =>1 in the GET-NARROWING-SOLUTIONS module. 49
Figure 43: Definition of the iterativeMetaVariantUnify function in the GET-NARROWING-SOLUTIONS module. 50
Figure 44: Equation that ends the unification step in the GET-NARROWING-SOLUTIONS module. 50
Figure 45: Equations that begin the computation of the solutions associated with each unifier found in the GET-NARROWING-SOLUTIONS module. 51
Figure 46: Equation of the GET-NARROWING-SOLUTIONS module that computes the narrowing solutions when we are at the first level of the reachability tree. 52
Figure 47: Equations of the GET-NARROWING-SOLUTIONS module that compute the narrowing solutions for levels below the first level of the reachability tree. 52
Figure 48: Equation that removes unifiers from the queue if the maximum number of solutions has already been reached in the GET-NARROWING-SOLUTIONS module. 53
Figure 49: Specification of the reduceSub function in the GET-NARROWING-SOLUTIONS module. 53
Figure 50: Function that ends the unify step when the solutions for a node have been computed in the GET-NARROWING-SOLUTIONS module. 53
Figure 51: Equations that define the termination of the narrowing algorithm in the GET-NARROWING-SOLUTIONS module. 54
Figure 52: Example 1 comparing narrowing commands using the NARROWING-VENDING-MACHINE module. 55
Figure 53: Example 2 comparing narrowing commands using the NARROWING-VENDING-MACHINE module. 55
Figure 54: Example 3 comparing narrowing commands using the NARROWING-VENDING-MACHINE module. 56
Figure 55: Example of a comparison between narrowing commands using the SUCC-PRED module. 56
Figure 56: Example of using the maximum depth parameter in the getNarrowingSolutions command, using the SUCC-PRED module. 57
Figure 57: Variables, operators and equations defined in the GET-CANONICAL-NARROWING-SOLUTIONS module to obtain the irreducibility terms. 61
Figure 58: Example of using the getIrreducibilityTerm function in an equation from the GET-CANONICAL-NARROWING-SOLUTIONS module. 62
Figure 59: Comparison between the received parameters and the structure obtained from the new defined commands and the metaNarrowingSearch command. 63
Figure 60: Comparison of commands looking for a solution for the mentioned narrowing problem by using the SUCC-PRED module. 64
Figure 61: Comparison of commands looking for three solutions for the mentioned narrowing problem by using the SUCC-PRED module. 64
Figure 62: Module in Maude to make recursive calls to the metaNarrowingSearch command, allowing to specify the maximum number of required solutions. 65
Figure 63: Comparison of algorithms for the mentioned narrowing problem by using the SUCC-PRED module. 66
Figure 64: Comparison of algorithms for the mentioned narrowing problem by using the XOR-PROTOCOL module. 67
Figure 65: Module that specifies the properties of abelian groups in Maude. 68
Figure 66: Comparison of algorithms for the mentioned narrowing problem by using the XOR-PROTOCOL module. 68

1. Introduction

In this chapter we make an introduction to this master thesis, with the intention of giving the reader a more general vision of what to expect from it.

1.1. Objectives

In this master thesis, the aim is to reimplement the narrowing of the Maude programming language, using reflection and the unification mechanisms included in it. Although narrowing is already implemented at the native level in the language, a new implementation using the part of the meta-level that it includes will allow to have a base on which to make modifications, managing to implement other types of narrowing that are not yet included. One of them is canonical narrowing [7], which will also be implemented in this work, using the standard narrowing implementation as a basis.

To achieve all these, it is first necessary to understand how rewriting, unification, and standard narrowing work at a theoretical level, being aware of some of their limitations and the importance of canonical narrowing to solve them. After this, it is also necessary to understand the basic and some more advanced concepts of Maude, as well as the operation of the unification and narrowing commands already defined in the language itself, especially those belonging to its meta-level.

Once everything is clear at a theoretical level, it is necessary to first implement an algorithm that is capable of generating the reachability tree starting from a term and using a specific rewrite system (a module in Maude), also using various parameters that allow the user to modify the operation of the command that invokes the algorithm, customizing the expected output. Using the same algorithm as a base, a new one can be built that can use the reachability tree that is generated to solve narrowing problems, by adding a target term and a maximum of expected solutions as parameters. Finally, the implemented standard narrowing algorithm can be used as a basis for making modifications and, in this case, implementing canonical narrowing.

When the necessary algorithms have been implemented to allow the resolution of narrowing problems, comparisons will be made using different rewrite systems (some of them specifying security protocols). On the one hand, we compare the implemented new commands, and on the other hand, the Maude native command, with the aim of demonstrating that the canonical narrowing implemented using the language itself is capable of solving problems in a more efficient way than the standard narrowing used by the Maude native command, whose base is encoded in C++, considering their execution times and the size of the narrowing trees generated when using different rewrite systems, initial terms, target terms and variations of the rest of the command parameters.

In addition, it is intended to establish a knowledge and code base to be able to make subsequent modifications, improving the algorithms and allowing the implementation of other types of narrowing.

1.2. Motivation

Cybersecurity plays an increasingly important role in the field of computing due to the constant appearance of new technologies and improvements to the existing ones, which lead to the creation of cryptographic protocols and algorithms with their potential vulnerabilities. In addition, the existing protocols that we continue to use today may also contain vulnerabilities that have not yet been detected. Therefore, it becomes vitally important to find and use tools and algorithms capable of detecting these vulnerabilities, giving the opportunity to fix them as soon as possible, or if necessary, even rule out the use of protocols that have them.

Narrowing makes it possible to generate reachability trees from an initial term, and to find those nodes (states) in which a target term is reached. If we can specify the properties of a protocol in a rewrite system, as well as the initial and target states as terms, we can use narrowing to detect vulnerable states, checking if any of them can be reached from the initial state of the protocol. That is why narrowing is presented as a very powerful tool in the field of cybersecurity, and there are tools such as *Maude-NPA* [14] that use it exactly for the intended purpose [9]. Other tools do not directly use Maude for the narrowing, but they do use it for unification with variants, taking advantage of its efficiency in equational theories with axioms. Some examples are *Tamarin* [2] and *AKISS* [12], that require the prior installation of Maude to function properly.

The search for improvements in the standard narrowing algorithm and the implementation of these improvements in programming languages such as Maude, on which Maude-NPA is based, can offer a wide range of possibilities when analysing protocols through reachability, seeking an improvement in performance and the time needed to achieve it. This work aims to lay the foundations for an implementation of canonical narrowing, an alternative version to standard narrowing that solves some of its limitations, using the latest stable version available from Maude. With later improvements, it would be possible to introduce what has been implemented into Full Maude, an extension of Maude written in Maude itself, allowing users to use the defined commands.

1.3. Document structure

At the beginning of this document, we can find the table of contents and the table of figures, and at the end of this we can find a list of the references used. It contains six main chapters, the first of which corresponds precisely to this chapter, the introduction. The second chapter presents the necessary preliminaries to be able to understand the rest of the chapters correctly and completely, as well as containing several references to learn more about the topics covered. If the reader is familiar with unification, narrowing, and the Maude programming language, this chapter will be especially straightforward.

The third chapter details the procedure followed to implement the algorithm that generates the reachability tree from a term and a rewrite system, including parts of the code and comments on it to facilitate understanding. The fourth chapter contains the explanation about the development of the algorithm that, based on the one already defined in chapter three, uses the achievable tree that is being generated to solve narrowing problems, for which an objective term is needed. The fifth chapter presents canonical narrowing and its advantages over standard narrowing, as well as its encoding in an algorithm based on that seen in chapter four. Finally, in the sixth chapter, a series of conclusions that we reached because of this work are told, as well as future works that are presented from this.

Each of the main chapters also includes several sections to make it easier to find content and read. At the beginning of each chapter a short introduction is included in which the chapter itself is presented in a summarized form, including the structure of its sections.

2. Preliminaries

In this chapter we introduce the basic knowledge necessary to understand this master thesis. To put it as clearly and correctly as possible, and to obtain some interesting examples, the official Maude manual [4] has been used as documentation, as well as several articles in which people belonging to the development team of this system participate as authors or co-authors [5][6]. It is recommended to read these documents if you want to extend your knowledge of what is exposed below. In Section 2.1, we present Maude 3.1. In Section 2.2, we briefly describe rewrite theories. In Section 2.3, we explain the use of types and subtypes in Maude. In Section 2.4, we introduce the use of variables and operators in this same language. After that, we talk about the modules in Maude, specifically, the functional modules in Section 2.5, and the system modules in Section 2.6. In Section 2.7, we present the concept of *unification*, to later talk about *order-sorted unification modulo axioms* in Section 2.8 and about *variant unification* in Section 2.9. In Section 2.10 we introduce the concept of *narrowing* and its use in Maude. Finally, in Section 2.11, we present reflection in Maude, that is, the meta-programming capabilities.

2.1. Introduction to Maude

Maude is a system and a programming language based on rewriting logic, which allows defining a wide range of complex computational models, such as concurrent programming or object-oriented programming. It is a declarative language, in contrast to imperative languages like C or Java.

Maude programs are rewrite theories, defined by what we call modules, and the calculation with those modules are efficient deductions performed by rewriting. Therefore, considering their logical and formal basis, it can be said that modules in Maude can specify mathematical models with a very high precision.

Being extremely efficient and presenting great potential to be used in diverse areas, Maude is currently used in organizations around the world for teaching, research, formal modelling, and analysis of concurrent and distributed systems [10][11].

2.2. Rewrite theories

Considering an equational theory $(\sum, E)$, where $\sum$ is a set of function symbols and E is a set of equations, and a set of rewrite rules R specifying the local concurrent transitions that a system can perform, a rewrite theory is defined as the triple $\mathcal{R} = (\sum, E, R)$. As previously stated, programs in Maude are rewriting theories, and each of them is a system module, which is defined with the following syntax: `mod (∑, E, R) endm`. Furthermore, if no rewrite rule is defined, that is, $R = \emptyset$, then the module will become a functional module, giving rise to the Maude's functional sublanguage. A functional module can be defined in Maude by using the following syntax: `fmod (∑, E, R) endfm`.

2.3. Sorts and subsorts

Data types in Maude are called sorts. The Maude system itself contains predefined sorts, which can be found in the *prelude* file, located inside the Maude installation folder. Some examples are *Bool*, *Nat* or *Int*, which are sorts that define booleans, naturals, and integers, respectively. Predefined sorts can be imported using three different reserved words: *including*, *protecting*, or *extending*. Each one has its specific function, but the safest and simplest to use is *protecting*. The *Bool* module is always imported automatically into any other new module, unless otherwise specified. Type declarations in Maude have the form "sort T .", or "sorts T_1 ... T_n ." in case we want to define several types at the same time.

Maude also allows to define subtypes of data, called subsorts, used to create data types ordered or divided into hierarchies. As with sorts, there are subsorts already defined in the *prelude* file, for example, between natural numbers and integers. Subtype declarations in Maude have the form "subsort $T_1 < T_2$.", where T_2 is the type of higher order in the hierarchy, that is, T_1 is included in T_2. Several related subsorts can also be defined at the same time: "subsort $T_1 < ... < T_n$.".

```
sorts Nat Int Float Char String .
subsort Nat < Int < Float .
subsort Char < String .
```

Figure 1: Example of types and subtypes definitions in Maude.

Figure 1 shows an example in which the types natural, integer and real (related to numbers) are defined, in addition to the types character and string of characters. Likewise, the existing hierarchy between them is defined with subsorts.

2.4. Variables and operators

Once the types and subtypes have been defined, the constructor symbols that will create the data associated with these types must also be defined, using operators. The definition of operators in Maude has the form "op C : T_1 ... T_n -> T .", although the creation of several symbols of the same type at the same time is also supported: `op` C_1 `...` C_n `:` T_1 `... Tn -> T .` For example, to define the type "Bool" and two constants "true" and "false" of that type, in addition to the type that represents lists of natural numbers and an operator that allows a natural number to be concatenated with an existing list, we can use the code in Figure 2.

```
sort Bool .
ops true false : -> Bool .
sort NatList .
op nil : -> NatList .
op _:_ : Nat NatList -> NatList .
```

Figure 2: Example of type definition and the use of constructor symbols in Maude.

Furthermore, in Maude it is mandatory to define the variables to be used later, using the reserved words "var" (to declare a single variable) or "vars" (to declare two or more variables of the same type at the same time). Figure 3 shows how two variables of type natural number could be declared, in addition to a list of naturals variable, and a set of naturals variable. Obviously, the types must be previously defined.

```
vars N M : Nat .
var NL : NatList .
var NS : NatSet .
```

Figure 3: Example of variables definition in Maude.

It is also allowed, instead of declaring variables before using them, to specify their type directly when using them, such as “X:Nat + Y:Nat”, to perform the addition of two variables whose type is natural number. In any case, the type of the variables must be specified, either before use or on the fly.

2.5. Functional modules

As can be seen in the very definition of what a rewrite theory is, membership equational logic is a functional sublogic of rewriting logic. The difference between both logics is that rewriting logic is characterized by its non-determinism, while functional logic is deterministic. For this reason, an equational program is a functional program in which a functional expression, which we can refer to as a term, will be evaluated, using the equations defined in the program as rewriting rules, which must be confluent. This evaluation could be infinite, but if it terminates, it ensures that it will return a unique value modulo some structural axioms, which is commonly called the normal form of the initial term, simplifying it by means of the defined equations.

In Maude, *functional modules*, which specify functional programs, are conditional membership equational theories. These theories have the form ($\sum$, $M \cup E \cup B$), where:

- $\sum$ is a signature composed by sorts and subsorts.
- M is a set of memberships, which provide greater precision when evaluating the type of a term, since it allows lowering it if the membership condition is met.
- E is a set of equations, which will be used, as mentioned, as left-to-right rules to evaluate the terms. These equations may be conditional.
- B is a set of equational axioms (e.g., commutativity, associativity, identity). Since rewriting with the equations E is performed modulo B, these axioms are considered structural axioms.

Let *FOO* be an arbitrary name for a module, then to define a functional module with that name in Maude, defined by a theory ($\sum$, $M \cup E \cup B$), previously seen keywords are used: `fmod (∑, M ∪ E ∪ B) endfm`. In Figure 4, we can see a concrete example of a functional module in Maude, which, in this case, defines the function that allows us to calculate the factorial of a number. Notice that the equations specifying how to reduce a term to normal form are defined using the reserved word “eq”.

```
fmod FACT is
    protecting NAT .
    op _! : Nat -> Nat .
    var N : Nat .
    eq 0 ! = 1 .
    eq N ! = (sd(N,1))! * N [owise] .
endfm
```

Figure 4: Maude module that calculates the factorial of a number.

Maude also allows the use of conditional equations, so that the equations can be applied to reduce the terms only if the corresponding condition specified in each of them is met. For example, the equation “eq 0 ! = 1 .” seen in Figure 4 could be written as a conditional equation of the form “ceq N! = 1 if N == 0 .”, so the equation will only be used to reduce the “N!” term to 1 if “N” has the value 0. As mentioned above, the “Bool” module is automatically imported if not specified otherwise, so it will be used to solve the conditions in the conditional equations. Thanks to the importation of this module, the use of the “if_then_else_fi” operator is also allowed. Therefore, conditions could also be defined in the equations as follows: `eq N! = if N == 0 then 1 else (sd (N, 1))! * N fi .`

2.6. System modules

When we add rewriting rules to a module in Maude, it is no longer a functional module, and it becomes what is called a *system module*, which allows us to model concurrent systems as conditional rewrite theories, which have the following form: (∑, *M* ∪ *E* ∪ *B*, *R*, Φ), where (∑, *M* ∪ *E* ∪ *B)* is the same as seen before in functional modules, *R* is a set of rewrite rules specifying the system transitions behaviour, thus making it a concurrent system, and Φ is a frozenness map, that may or may not add rewriting restrictions when applying the rules. Note that the functional part of Maude specifies a static system, while rewriting rules add dynamism to the system.

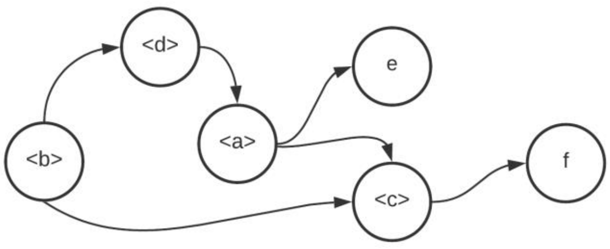


Figure 5: State diagram that models a simple rewrite system.

Let *FOO* be an arbitrary name for a module, then to define a system module with that name in Maude, defined by a theory (∑, *M* ∪ *E* ∪ *B, R,* Φ), previously seen keywords are used: `mod (∑, M ∪ E ∪ B) endm`. In Figure 6, we can see an example of a basic concurrent system defined using rules in Maude. Specifically, the behaviour defined in the state diagram of the Figure 5 is encoded.

```
mod SIMPLE-REWRITE-SYSTEM is
    sort Node .
    ops a b c d e f : -> Node .
    op <_> : Node -> Node .
    rl < a > => e .
    rl < a > => < c > .
    rl < b > => < d > .
    rl < b > => < c > .
    rl < c > => f .
    rl < d > => < a > .
endm
```

Figure 6: Basic example of a system module in Maude.

As with equations, the use of conditional rules is also allowed in system modules, that is, the definition of conditional transitions in the system is allowed. For example, let *N1* and *N2* be two natural numbers, then the conditional rule that rewrites the term *add (N1, N2)* to the result of the addition of the two variables only if *N1* is greater than *N2*, is the following: `crl add(N1, N2) => N1 + N2 if N1 > N2 .`

2.7. Unification

Unification is an algorithmic process of solving equations. It can take place in free algebras, taking the name of *syntactic unification*, or in relatively free algebras modulo a set of equations *E*, taking the name of *E-unification* (or *semantic unification* if E is not explicitly mentioned).

Each solution to a syntactic unification problem is a *substitution*, which when used to instantiate the variables that appear in the terms of the unification problem, will cause the terms to become equal. For example, a solution to $f(a, X) = f(Y, b)$ would be the substitution $\sigma = \{X \mapsto b, Y \mapsto a\}$, since when applied to the two terms, the equation will become of the form $f(a, b) = f(a, b)$, that is, both terms are equal now, which is exactly what is intended.

It is also interesting to clarify the concept of *E-unifier* for an equation $t = t'$, which is a substitution σ such that $t\sigma =_E t'\sigma$, that is, a substitution that equates the terms modulo *E*.

Maude supports unification algorithms that, as we will see in the next section, are capable of taking into account the axioms of associativity, commutativity, identity and their combinations, that we will refer to as *B*.

2.8. Order-sorted unification modulo axioms *B*

For any order-sorted theory (Σ, *B*), where *B* is any of the axioms (or their combinations) mentioned above, Maude provides an order-sorted *B*-unification algorithm, which can be invoked using the command that has the following form:

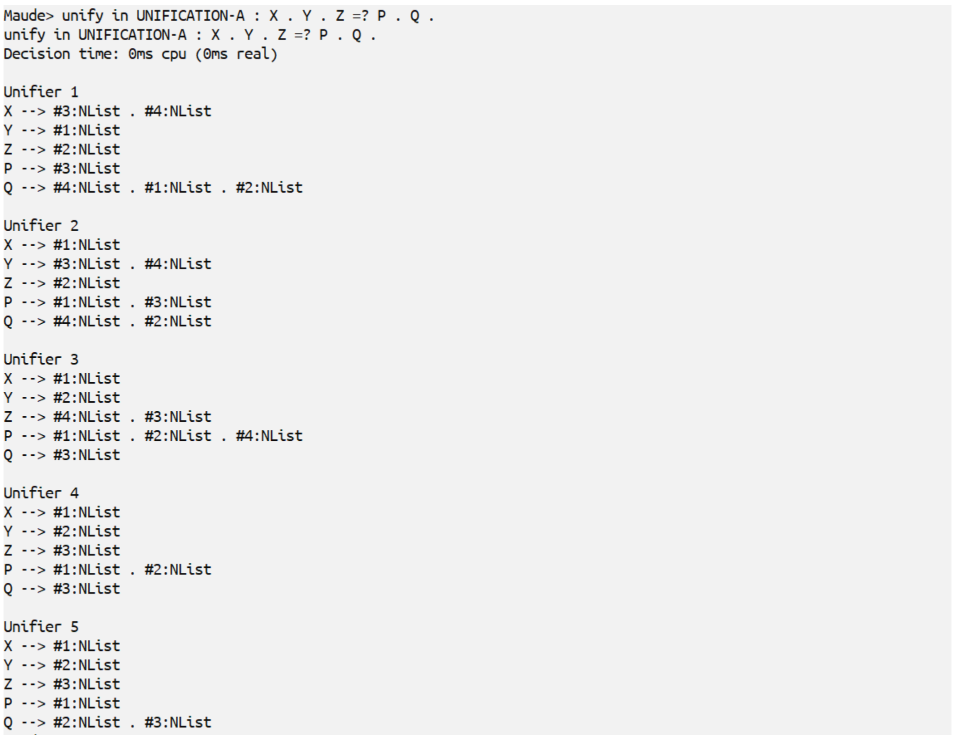
```
unify [n] in <ModId> :
    <Term-1> =? <Term'-1> /\ ... /\ <Term-k> =? <Term'-k> .
```

where *n* is a non-mandatory argument indicating the maximum number of requested unifiers, *k* is a natural number greater or equal to 1, and *ModId* is the name of the module where the command will be applied.

```
fmod UNIFICATION-A is
    protecting NAT .
    sort NList .
    subsort Nat < NList .
    op _._ : NList NList -> NList [assoc] .
    vars X Y Z P Q : NList .
endfm
```

Figure 7: Maude Module that defines the concatenation of lists with associativity.

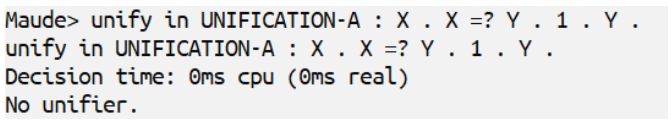
To see a practical example, consider the module defined in Figure 7. Associative unification is infinitary, that is, usually there is no finite set of unifiers for a unification problem with associativity, but there are some concrete realistic problems that are finitary. Maude manages to return five unifiers by asking it to solve the following unification problem: *X.Y.Z =? P.Q*, that is, by asking if there is a substitution for the variables used such that by concatenating the lists *X*, *Y* and *Z*, and the lists *P* and *Q*, an equal list is obtained in both cases. In Figure 8, we can see the command used for this, and the output (solution) returned by Maude, that is, the unifiers.



```
Maude> unify in UNIFICATION-A : X . Y . Z =? P . Q .
unify in UNIFICATION-A : X . Y . Z =? P . Q .
Decision time: 0ms cpu (0ms real)

Unifier 1
X --> #3:NList . #4:NList
Y --> #1:NList
Z --> #2:NList
P --> #3:NList
Q --> #4:NList . #1:NList . #2:NList

Unifier 2
X --> #1:NList
Y --> #3:NList . #4:NList
Z --> #2:NList
P --> #1:NList . #3:NList
Q --> #4:NList . #2:NList

Unifier 3
X --> #1:NList
Y --> #2:NList
Z --> #4:NList . #3:NList
P --> #1:NList . #2:NList . #4:NList
Q --> #3:NList

Unifier 4
X --> #1:NList
Y --> #2:NList
Z --> #3:NList
P --> #1:NList . #2:NList
Q --> #3:NList

Unifier 5
X --> #1:NList
Y --> #2:NList
Z --> #3:NList
P --> #1:NList
Q --> #2:NList . #3:NList
```

Figure 8: Unifiers returned by Maude as a solution to the unification problem seen above.

In this output we can see the way in which Maude generates and identifies fresh variables (by using the notation “#N:Sort”), since the solutions it returns in this case are made up of variables of the natural list type, which will be assigned to the input variables. In most bindings only one fresh variable is assigned to each input variable, being a simple renaming, but in others we can see how Maude assigns to one of the input variables the concatenation of two fresh variables (concatenation of two lists). By returning solutions composed only of variables, Maude is including many other implicit solutions, which will be obtained by instantiating those new variables.



```
Maude> unify in UNIFICATION-A : X . X =? Y . 1 . Y .
unify in UNIFICATION-A : X . X =? Y . 1 . Y .
Decision time: 0ms cpu (0ms real)
No unifier.
```

Figure 9: Output returned by Maude in response to an impossible unification problem.

In some specific cases, Maude will not be able to find unifiers in a finite time, nor will it be able to determine that there are no unifiers for the problem, but in many others, it is able to quickly detect that there is no unifier. We can see an example of this using the same module defined for the previous example, but this time we will ask Maude to find unifiers for $X.X =? Y.1.Y$. In the left part, we are concatenating a list with itself, and in the right part, we are concatenating a list with itself but adding a number between them (In this case, the number one). It is impossible to unify in this case, and Maude is able to quickly detect it, as can be seen in Figure 9.

There is also a variant of the command which after generating all the unifiers, is able to filter them to return only the *MGU* (Most general unifiers modulo axioms) set: the *irredundant unify* command.

2.9. Variant unification

Let t, u be terms and σ be a substitution. If t is in a convergent order-sorted equational theory $(\Sigma, E \cup B)$, where E are unconditional equations, then when applying σ to the term t, usually noted as $t\sigma$, we get u, a *variant of* t, which is a normal form of t computed by simplification with E modulo B. It may be useful to represent the variants as pairs of the form (u, σ) [3]. As in many cases a term can be reduced to several normal forms, it can have several variants, and some may be more general than others, because they will include some or all the others. Specifically, a variant (u, σ_1) is considered more general than another (v, σ_2) if and only if there is a substitution γ such that $u\gamma =_B v$, and for each variable z in t, $z\,\sigma_1\,\gamma =_B z\,\sigma_2$.

To better understand the above, let us see an example: consider the unsorted signature $\Sigma = \{0, s, \cdot, +\}$ of multiplication in the *Peano* natural numbers, with $B = \emptyset$ and $E = \{X \cdot 0 = 0, X \cdot s(Y) = X + (X \cdot Y)\}$. If we consider the substitution $\sigma_1 = \{Y \mapsto 0\}$, the normal form of the term $X \cdot Y$ will be 0, but if we consider the substitution $\sigma_2 = \{Y \mapsto s(Y')\}$, the normal form for the same term will be $X + (X \cdot Y)$, so that the pairs $(X + (X \cdot Y), \sigma_1)$ and $(0, \sigma_2)$ are variants of the term $X \cdot Y$.

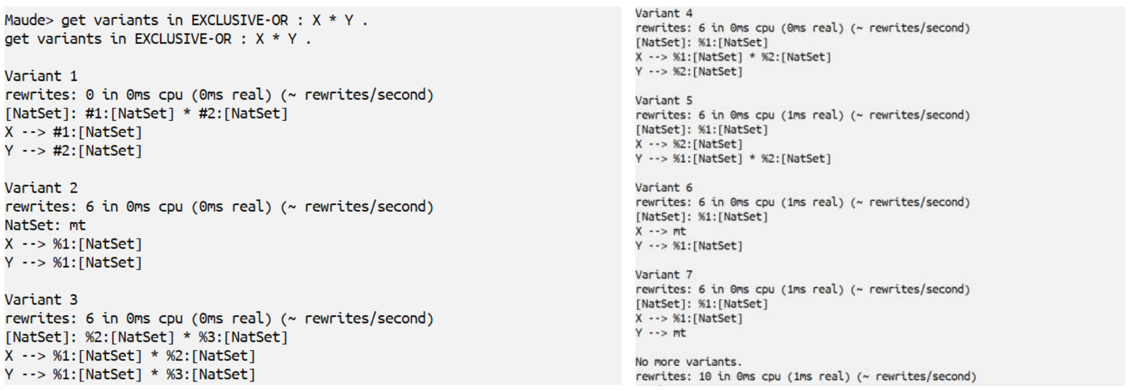
```
fmod EXCLUSIVE-OR is
    sorts Nat NatSet . subsort Nat < NatSet .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    op mt : -> NatSet [ctor] .
    op _*_ : NatSet NatSet -> NatSet [assoc comm] .
    vars X Y Z U V : [NatSet] .
    eq [idem] : X * X = mt [variant] .
    eq [idem-Coh] : X * X * Z = Z [variant] .
    eq [id] : X * mt = X [variant] .
endfm
```

Figure 10: EXCLUSIVE-OR module in Maude.

The finite variant property (FVP) occurs in a theory (∑, $E \cup B$), which is order-sorted, if and only if each term $t \in \sum$ has a finite set of most general variants. The *EXCLUSIVE-OR* module defined in Figure 10 fulfills this property, and to check it, Maude is useful, since it allows us to invoke a variant generator command that has the form:

```
get variants [n] in <ModId> : <Term> .
```

where *n* is a non-mandatory argument indicating the maximum number of requested variants (only used if the total of computed variants is greater than that number), and *ModId* is the name of the module where the command will be applied. When generating the variants with Maude for the exclusive-or symbol (represented by *), we can verify that, indeed, they meet the necessary conditions mentioned. We can see the command output in Figure 11, where we can also appreciate another identifier that Maude uses for fresh variables, some of them now being of the form "#N:Sort", and others of the form "%N:Sort". The first form is used for those fresh variables generated when performing the unification modulo axioms, and the second for the fresh variables generated in the substitutions returned by variant generation. Of course, each of the forms has its own counter.



```
Maude> get variants in EXCLUSIVE-OR : X * Y .
get variants in EXCLUSIVE-OR : X * Y .

Variant 1
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
[NatSet]: #1:[NatSet] * #2:[NatSet]
X --> #1:[NatSet]
Y --> #2:[NatSet]

Variant 2
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
NatSet: mt
X --> %1:[NatSet]
Y --> %1:[NatSet]

Variant 3
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
[NatSet]: %2:[NatSet] * %3:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %1:[NatSet] * %3:[NatSet]

Variant 4
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
[NatSet]: %1:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %2:[NatSet]

Variant 5
rewrites: 6 in 0ms cpu (1ms real) (~ rewrites/second)
[NatSet]: %1:[NatSet]
X --> %2:[NatSet]
Y --> %1:[NatSet] * %2:[NatSet]

Variant 6
rewrites: 6 in 0ms cpu (1ms real) (~ rewrites/second)
[NatSet]: %1:[NatSet]
X --> mt
Y --> %1:[NatSet]

Variant 7
rewrites: 6 in 0ms cpu (1ms real) (~ rewrites/second)
[NatSet]: %1:[NatSet]
X --> %1:[NatSet]
Y --> mt

No more variants.
rewrites: 10 in 0ms cpu (1ms real) (~ rewrites/second)
```

Figure 11: Variants of the exclusive-or symbol in Maude.

Once we check that the variants generated for the exclusive-or symbol meet the necessary conditions, we can also see that, since the rest of the symbols *f* (excluding the exclusive-or symbol) have the unique trivial variant *(f* $(x_1, \ldots x_n)$, *id)*, where *id* is the *identity substitution*, it can be easily proved that the module is FVP. This property is extremely useful for performing derived proofs on modules that meet it.

2.10. Narrowing

When we work with a rewrite theory ($\sum$, $E \cup B$, R, Φ), on many occasions, such as when we specify a security protocol, we will be interested in the problem of symbolic reachability, that is, given two terms t and t', we want to know if $\exists X\ t \rightsquigarrow^* t'$, where X is the set of variables appearing in both terms, which, in this case, is the result from a disjoint union of the variables that appear in t and of those that appear in t'.

We are therefore asking if there is any state (t and t' represent sets of state instances) from which we can reach t' in a finite number of rewriting steps, and to answer this question, it is necessary to search for a symbolic solution, trying to do it in a *complete* way, so that if such a solution exists, we are able to find it. To do this, we can perform *narrowing* in t, using the rewrite rules R that have been defined in a system module (as seen above), module equations $E \cup B$. If using these rules modulo equations we obtain a finite sequence of narrowing of the form $t \rightsquigarrow^*_{R,E\cup B} u$, by using the composition of unifiers that are computed to instantiate t in each step of the sequence, in addition to unifiers for the equations, which will be used to get $u = t'$.

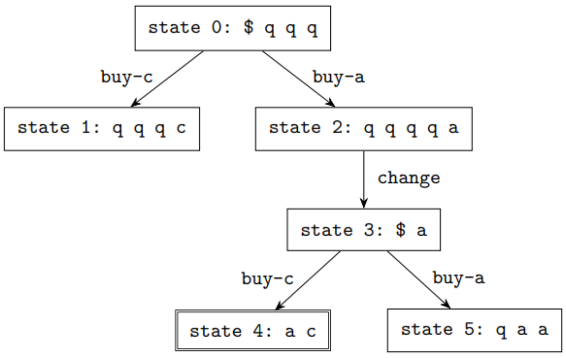


Figure 12: Example of a search graph in a vending machine module [4].

The usefulness of this is very wide. For example, if we take t as a set of initial terms describing states of a concurrent system, and t' as a possible set of terms that represent a state that violates certain invariants, using reachability in narrowing we would be doing logical model checking. It can also be used to, once certain security protocols have been specified as terms, to find out if they have any state of vulnerability at a theoretical level, and then to ask if these vulnerabilities could be exploited in practice.

The *search* command in Maude is slightly related to narrowing, since it tells us if in the space of possible program executions there is an execution that satisfies certain conditions, that is, it explores the reachable state space in different ways. This command has the form:

```
search [n, m] in <ModId> : <Term-1> <SearchArrow> <Term-2>
  such that <Condition> .
```

where n and m are optional arguments indicating the maximum solutions or depth desired, *ModId* is the module where the command is applied, *Term-1* and *Term-2* are the starting and target (to be reached) terms, *Condition* is an optional argument to specify a property that must be satisfied by the reached state, and *SearchArrow* is an arrow argument indicating the form of the rewriting proof, which can take the values:

- =>1, indicating exactly one step.
- =>+, indicating one or more steps.
- =>*, indicating any number of steps.
- =>!, indicating that only final steps will be considered solutions.

In Figure 12, we can see an example of the graphical representation (as a graph) of a possible output when executing the *search* command on a module that specifies a vending machine, similar to the one we can see in Figure 13.

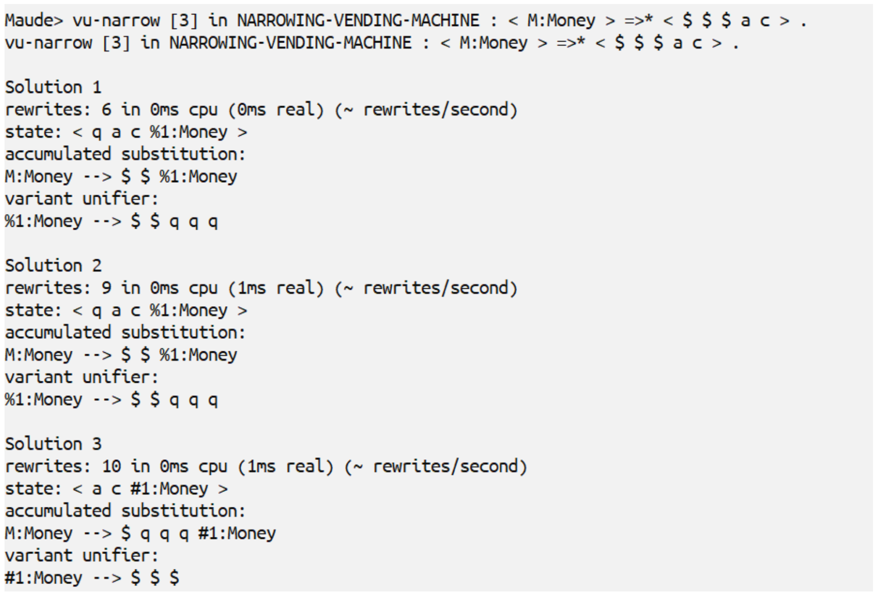
```
mod NARROWING-VENDING-MACHINE is
    sorts Coin Item Marking Money State .
    subsort Coin < Money .
    op empty : -> Money .
    op __ : Money Money -> Money [assoc comm id: empty] .
    subsort Money Item < Marking .
    op __ : Marking Marking -> Marking [assoc comm id: empty] .
    op <_> : Marking -> State .
    ops $ q : -> Coin .
    ops a c : -> Item .
    var M : Marking .
    rl [buy-c] : < M $ > => < M c > [narrowing] .
    rl [buy-a] : < M $ > => < M a q > [narrowing] .
    eq [change] : q q q q M = $ M [variant] .
endm
```

Figure 13: NARROWING-VENDING-MACHINE module, from the Maude's official examples.

The *vu-narrow* command has a form very similar to the *search* command, but in this case, we will be directly invoking the command that allows us to solve narrowing problems in Maude. It uses the variant-based unification at each step, and that is why the prefix of the command is *vu* (variant-based unification). The command is invoked as follows:

```
vu-narrow [n, m] in <ModId> :
                              <Term-1> <SearchArrow> <Term-2> .
```

where all the arguments have a meaning similar to that of the *search* command. Note that the term to be reached *Term-2* may share variables with the initial term *Term-1* (terms in the narrowing sequence will be unified with this target pattern, in contrast to the search command).



```
Maude> vu-narrow [3] in NARROWING-VENDING-MACHINE : < M:Money > =>* < $ $ $ a c > .
vu-narrow [3] in NARROWING-VENDING-MACHINE : < M:Money > =>* < $ $ $ a c > .

Solution 1
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
state: < q a c %1:Money >
accumulated substitution:
M:Money --> $ $ %1:Money
variant unifier:
%1:Money --> $ $ q q q

Solution 2
rewrites: 9 in 0ms cpu (1ms real) (~ rewrites/second)
state: < q a c %1:Money >
accumulated substitution:
M:Money --> $ $ %1:Money
variant unifier:
%1:Money --> $ $ q q q

Solution 3
rewrites: 10 in 0ms cpu (1ms real) (~ rewrites/second)
state: < a c #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> $ $ $
```

Figure 14: Example of using the vu-narrow command.

In Figure 14 we can see one of its possible invocations on the module defined in Figure 13, and the result obtained. Specifically, we have asked Maude for 3 solutions in which it is possible to arrive from a *Money* variable *M* to a state in which we have 3 dollars (*$ $ $*), an apple (*a*) and a cake (*t*).

It is important, due to the nature of this work, to point out that there are other types of narrowing as alternatives to the one we have seen in this section, which we can refer to as the *standard narrowing module an equational theory (∑, E ∪ B)*, such as *canonical narrowing module an equational theory (∑, E ∪ B)*, which is based on the calculation of irreducibility constraints that must be met in each step of the narrowing sequence. By using it, we will be able to discard some sequences that are not necessary, as we will see in more detail in Chapter 5.

2.11. Maude reflection

Maude makes the meta-theory of rewriting logic accessible to the user (rewrite logic is *reflective*), since it is something that is used a lot in the design and implementation of this language.

The reflection of the rewrite logic allows to have a finite rewrite theory U which we can use to represent any other finite rewrite theory R (including U itself) as a term R′, as well as any terms t_1, t_2 in R as new terms t'_1, t'_2 and any pair (R, *t*) as a term <R, *t'*>, and the following equivalency appears: $R \vdash t_1 \longrightarrow^* t_2 \Leftrightarrow U \vdash \langle R', t'_1\rangle \longrightarrow^* \langle R', t'_2\rangle$, that is t_1 rewrites into t_2 using the rewrite theory R if and only if the pair <R′, t'_1> rewrites into <R′, t'_2> using the rewrite theory U. Since U can also represent itself, there can be as many levels of reflection as we want, generating a *reflective tower*.

Maude contains an implemented *META-LEVEL* module that gives access to the U theory mentioned above, as well as *META-INTERPRETER* module which can even handle computational interactions with the external world. This allows the use of metaprogramming, and there is a part of Maude full of extensions that have been defined in Maude himself, using reflection, called *Full Maude*. Furthermore, the existing metaprogramming commands in Maude can be used to do, for example, modifications to algorithms already implemented in the language, achieving improvements, or even adding new functionalities of interest.

Reflection on Maude plays a very important role in this Master Thesis, since it has been used to implement all the modules discussed later. When we get to them, we will see reflective commands in a more detailed way, as we use various commands defined in Maude's *META-LEVEL* module to implement the new commands that we will discuss later, as well as types and subtypes also defined in that module. For example, some commands have been used such as *getType*, which has several versions, but in our version the one that receives a variable as an argument and returns its type in Maude has been used. Commands such as *string* and *qid* have also been used to construct variables, using numbers and identifiers.

```
op metaNarrowingApply :
    Module Term TermList Qid Nat -> NarrowingApplyResult?
    [special ...] .
sorts NarrowingApplyResult NarrowingApplyResult? .
subsort NarrowingApplyResult < NarrowingApplyResult? .
op {_,_,_,_,_,_,_} : Term Type Context Qid Substitution Substitution Qid
                        -> NarrowingApplyResult
                        [ctor format (d n++i d d d ni d ni d d d d d ni n--i d)] .
op failure : -> NarrowingApplyResult? [ctor] .
op failureIncomplete : -> NarrowingApplyResult? [ctor] .
```

Figure 15: Definition of the metaNarrowingApply command and its related types in Maude's META-LEVEL module.

There are some commands of Maude's meta-level that we consider necessary to highlight, since they have been of vital importance to achieve the objectives proposed in this master thesis. All of them have their base defined in the C++ language, so when defining them in the *META-LEVEL* module, special operators are used that refer to that code. One of them is the *metaNarrowingApply* command, whose definition is shown in Figure 15, and which is used to invoke a narrowing step. To do this, it needs to receive as arguments the module that defines equations and rules with which to rewrite, a term from which to rewrite, a list of irreducibility terms (which we will detail later), a *Qid* to indicate which identifier should not be used when generating fresh variables, and a natural number that indicates the narrowing step to take, since there can be more than one from the same term. In the same figure, the data type returned by the command is also defined, *NarrowingApplyResult?*, and its subtype *NarrowingApplyResult*. It is a data structure that returns the information of the result when performing the narrowing step, such as the new term obtained or the substitutions that have been used to reach it. The constants *failure* and *failureIncomplete* are returned if the narrowing step was not possible.

Likewise, there is a *metaNarrowingSearch* command, which receives similar arguments, but includes a target term, since it is used to perform the entire narrowing algorithm, instead of just one step. We will use it to perform the comparisons with our narrowing commands.

```
op metaVariantUnify :
    Module UnificationProblem TermList Nat Nat ~> UnificationPair?
    [special (...)] .
sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
            [ctor assoc comm prec 73] .
subsort UnificationPair < UnificationPair? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
```

Figure 16: Definition of the metaVariantUnify command and its related types in Maude's META-LEVEL module.

Another relevant command is *metaVariantUnify*, whose definition in Maude is shown in Figure 16. It is used to solve variant unification problems, and receives as arguments a module to work on, the unification problem, a list of irreducibility terms, a Natural number from which to number the identifiers of the generated fresh variables, and another natural number used to ask for the unifiers found (if any), since there may be more than one. This command is used when implementing narrowing, since it is necessary to try to unify the terms that are generated with the target term to find the solutions. Later we will see that the list of irreducible terms is very important to implement *canonical narrowing modulo an equational theory* ($\sum$, $E \cup B$), explained in Chapter 5.

3. Generation of the reachability tree

In this chapter we describe a possible implementation to generate the reachability tree using the Maude language, taking advantage of its reflective properties, that is, using some of the commands defined at the meta-level. In Section 3.1, we present the implementation process approach used to get through to final implementation. In Section 3.2, we present the types, subtypes, and operators that are initially used to generate the reachability tree. In Section 3.3, we analyse the equations necessary to generate the reachability tree. In Section 3.4, we detail the module used to rename the fresh variables generated by Maude. Finally, in Section 3.5, we show some examples of executing the command implemented to generate the reachability tree from a term.

3.1. Implementation methodology

Generating a reachability tree at the theoretical level is relatively simple, but when it comes to implementing it, several complications arise. We will deal with many of them in the subsequent sections.

The first thing we need to implement the generation of reachability trees in any language (Maude in this case) is to be clear about the necessary steps to achieve the complete algorithm, since several incremental iterations are necessary. The main phases of the implementation that we must go through are:

1. Define the types, subtypes and data structures that will be necessary throughout the process of generating the reachability tree.
2. Generate the reachability tree from an initial term, using a received module to generate the variants, and considering a parameter that indicates the maximum desired depth.
3. Rename the fresh variables generated by Maude, because if we use the original ones, then clashes between them will arise.

In the following sections, we detail each of the steps, as well as the code that has been generated in each of them, seeing how it has been transformed and grown, along with the necessary justifications. We also show some examples of executing the command using several different parameters to generate the reachability tree from an initial term.

3.2. Types, subtypes, and data structures

In this section we show the types, subtypes and data structures initially used to generate the reachability tree in the *GET-REACHABILITY-TREE* module. The code snipped that we detail in this section can be seen in Figure 17.

We want to generate a reachability tree from an initial term, which we can consider as *node 0* of the tree. We need to create a data structure that is able to represent that tree in Maude. The strategy chosen for this consists of using a list to which the nodes of the tree that are being generated are added. As later we will need to be clear about the position of each node in the tree to compute the final solution (which includes the substitutions made in each branch of the tree), each of the nodes must contain a reference to its parent node, so it is necessary to identify each of them in a unique way (for example, with a natural number that is self-increasing). This reference of each node to its parent node will not be useful only when computing the composition of the substitutions at the end, but it is also useful to always have control of the parameters passing from parents to children when performing the recursive calls used by the reachability tree generation.

```
sorts NarrowingApplyResultList NarrowingApplyResultStructure State .
subsort NarrowingApplyResultStructure < NarrowingApplyResultList < State .

op getReachabilityTree : Module Term TermList Qid Bound -> State .

op nil : -> NarrowingApplyResultList [ctor] .
op _;_ : NarrowingApplyResultList NarrowingApplyResultList
          -> NarrowingApplyResultList [assoc id: nil] .

op {_,_,_,_,_,_} : Nat NarrowingApplyResult Nat Nat Qid Nat
                    -> NarrowingApplyResultStructure .

op (_,_,_,_,_,_,_) : Module Term Nat NarrowingApplyResultList
                        NarrowingApplyResultList Bound Nat -> State .
```

Figure 17: Sorts, subsorts and operators in the GET-REACHABILITY-TREE module.

To generate each node in the tree, we call the *metaNarrowingApply* function from the *META-LEVEL* module. The nodes of our generated tree must therefore contain the outputs returned by that function (detailed in the Maude manual [4]). To do this, a type called *NarrowingApplyResultStructure* is defined, which represents each node and contains:

- A natural number that identifies the node. It is different from the identifiers of the other nodes.
- A *NarrowingApplyResult*, which corresponds to the output returned by the *metaNarrowingApply* command and contains the information of the term of that node and the substitutions made to obtain it.
- A natural number that identifies the sub-branch in which the node is located. To determine this, starting from a parent node, the child nodes are numbered from left to right, with the first child node generated on the left and the last one on the right.
- A natural number that refers to the identifier of the parent node. This number will be 0 if the node is obtained from the initial term.
- A *Qid* that indicates the identifier that has been used to name the generated fresh variables.
- A natural number that indicates the depth of the node within the tree.

This structure is created by using the defined operator "{_,_,_,_,_,_}", so that it takes a form similar to an object defined in *JSON* format.

A *nil* operator that symbolizes the empty node list and an operator "_;_" used to concatenate lists of nodes, so that they can grow, have been defined. Nodes are defined as subtypes of *NarrowingApplyResultList*, that is, of the lists that contain them. Thereby, nodes can be used as if they were lists of nodes, allowing the same operator to concatenate them and generate a larger list. Note that the concatenation of lists has been defined as associative, to facilitate subsequent operations.

A *State* type has been defined to save all the necessary data to generate the tree. Once the tree is generated, the irrelevant data is eliminated and only the list of tree nodes is kept. To make this easier, *NarrowingApplyResultList* is defined as a subtype of *State*.

There are two operators whose result is of type *State*. One of them is *getReachabilityTree,* which corresponds to the definition of the command that will be used to execute the generation of the tree, that is, the command that should normally be used as a user. This operator receives as parameters:

- The module where the command will take place.
- The term from which the tree is generated.
- A list of terms, which is not relevant now, and will be explained later.
- A *Qid* that indicates the symbol used in the initial term to identify the variables, so that Maude avoids using it to identify the fresh variables.
- A *Bound*, which is a predefined type that can be a natural number or *unbounded*. It is used to define the desired depth limit to generate the tree.

The second operator whose result is of type *State* is "(_,_,_,_,_,_,_)". It receives several parameters, since it is the main structure that the defined module will use throughout the tree generation process. These parameters are:

- The module that contains the rules, variants and axioms that will be used to generate the reachability tree.
- The initial term from which the tree will be generated.
- A natural number that works as a node counter. That is, it increases each time a node is generated, and it will be used to create the identifier of the next node.
- A *NarrowingApplyResultList* that will contain the nodes already processed, that is, the nodes whose children or next nodes have been generated.
- A *NarrowingApplyResultList* that will contain the nodes that must be processed, that is, the nodes from which the following children and nodes must still be generated.
- A *Bound*, which is a predefined type that can be a natural number or *unbounded*. It is used to define the desired depth limit to generate the tree.
- A natural number that works as a variable counter. That is, it increases each time a fresh variable is renamed. We detail this parameter later when we describe variable renaming in this module.

It is important to note that both the operators "{_,_,_,_,_,_}" and "(_,_,_,_,_,_,_)" work as auxiliary operators for internal computing, behaving in a different way than what we are used to with operators that are more similar to functions.

3.3. Tree unfolding

Once we have defined the types, subtypes, and data structures, together with the operators, we can proceed to specify the behavior using equations, in order to unfold the reachability tree. For this, it is necessary to use variables, which must be previously declared. In Figure 18 we show the variable declarations used later in equations.

```
var M : Module .   vars T1 T2 T3 T4 : Term .   var TList : TermList .
vars Q1 Q2 Q3 Q4 Q5 Q6 : Qid .   vars N1 N2 N3 N4 N5 N6 N7 N8 N9 : Nat .
vars Ty1 Ty2 : Type .   vars C1 C2 : Context .   vars S1 S2 S3 S4 : Substitution .
vars MaxDepth : Bound .   vars NARS1 NARS2 NARS3 : NarrowingApplyResultList .
```

Figure 18: Variables declared in the GET-REACHABILITY-TREE module.

As discussed above, the user is expected to use the *getReachabilityTree* command to call the algorithm that generates the reachability tree. Initially, two options should be considered, which in this case are that the parameter that specifies the maximum depth to be generated from the tree is 0, or that it is another of the allowed values. In the first case, the reachability tree cannot be generated, so the empty list must be returned. In the second case, it can be generated, so the initial data structures necessary for this must be prepared. Specifically, a *State* type structure will be created using the operator seen above. The module, the initial term, the *Qid* and the maximum depth will be obtained from the parameters specified by the user. The list of nodes processed will be initially empty, and the list of nodes to process will contain a call to the *metaNarrowingApply* command that generates the first node in the tree. The parameters for this call will also be obtained from those indicated by the user, except for the solution number, which will be 0 (since we want the first solution, which is indexed by that number in this command) and the list of terms, which will be *empty*. The identifier of the node that is generated will be 1, its parent node 0, its sub-branch 0, and its depth 1. We also set the variable counter to 1, the number from which the renaming will begin later. Figure 19 shows the two equations used for all this.

```
eq getReachabilityTree(M, T1, TList, Q1, 0) = nil .
eq getReachabilityTree (M, T1, TList, Q1, MaxDepth) =
   (M, T1, 2, nil, {1, metaNarrowingApply(M, T1, empty, Q1, 0),
    0, 0, Q1, 1}, MaxDepth, 1) [owise] .
```

Figure 19: Equations that define the initial behavior of the getReachabilityTree command in the GET-REACHABILITY-TREE module.

Assuming that we have generated the first child node of the initial node 0 (later we will detail what happens when a node is not generated), two other possibilities are now proposed, which depend on the value of the depth limit, which can be:

- A natural number, specifying the maximum level of the tree whose nodes will be generated. For example, if the user has indicated the value 4 for the parameter, 4 levels of nodes will be generated, ignoring the node corresponding to the initial term, which we consider level 0.

- The constant value *unbounded*, predefined in the *META-LEVEL* module, which indicates that the user does not want to specify a maximum depth for the reachability tree, but rather wants to generate it completely. This value for the parameter has the risk that it could generate an infinite computation in case the reachability tree for the given term is also infinite.

```
eq (M, T1, N1, NARS1, {N2, {T2, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, 1} ; NARS2,
    MaxDepth, N9) =
      (M, T1, N1 + 2, NARS1 ;
        {
          N2,
          {
            applySub(T2, rename((getVars(T2), getRangeVars(S1),
                                 getRangeVars(S2)), N9)),
            Ty1, C1, Q1,
            applySub(S1, rename((getVars(T2), getRangeVars(S1),
                                 getRangeVars(S2)), N9)),
            applySub(S2, rename((getVars(T2), getRangeVars(S1),
                                  getRangeVars(S2)), N9)),
            '$
          },
          N3, 0, Q3, 1
        },
        {
          N1,
          metaNarrowingApply(M, T1, empty, Q3,N3 + 1),
          N3 + 1, 0, Q3, 1
        } ; NARS2 ;
        {
          N1 + 1,
          metaNarrowingApply(M, applySub(T2, rename((getVars(T2),
                                getRangeVars(S1), getRangeVars(S2)), N9)),
                             empty, Q2, 0),
          0, N2, Q2, 2
        },
      MaxDepth,
      N9 + getVarsNumber((getVars(T2), getRangeVars(S1), getRangeVars(S2))) .
```

Figure 20: Equation that defines the generation of the first level of the reachability tree in the GET-REACHABILITY-TREE module.

In both cases, the behavior to generate the nodes of the first level of the reachability tree is defined by the same equation, since if the maximum depth parameter does not take the value 0, the first level will always be generated. Figure 20 shows the equation we use for this. Note that there are some calls to variable renaming functions, but we will explain that later.

We want the tree to be generated by levels, so for each node at this level, the next node in the same level must be generated. Furthermore, the generation of the first child node of the next level will be triggered but delayed for later, so that the nodes of that level will not be generated until all the nodes in the current level are generated. If we take a look at Figure 21, where an example tree is represented with the nodes labeled with letters from A to S, the order to generate the nodes that adapts to what we want is to follow the alphabetical order. To achieve this, we must pay special attention to the exact place where to put each of the calls to *metaNarrowingApply* in the list of nodes to be processed.

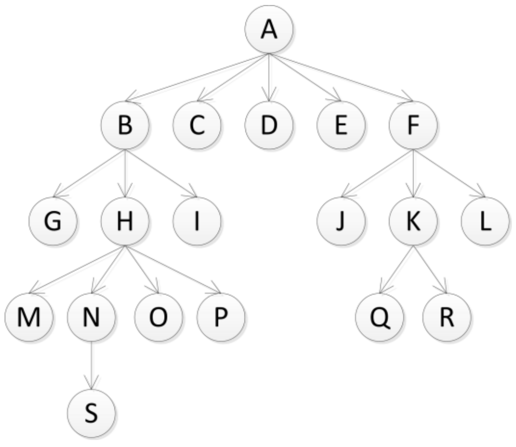


Figure 21: An example tree with 19 nodes labeled with letters.

At the code level, when in the main structure, of type *State*, a node in which it is indicated that the depth level is 1 and its parent node is 0 is at the beginning of the list of nodes to be processed, we move the node at the end of the list of nodes processed. At the same time, two calls to *metaNarrowingApply* are made:

- The first call occurs at the beginning of the list of nodes to be processed and is made to generate the next node that is at the same level that currently processed node, that is, in the next sub-branch of the current level. The arguments for the call will be the same as those used for the current node, only that we will set the list of terms to *empty* and add 1 to the requested solution (and therefore, to the parameter that indicates the sub-branch). The node depth will also be 1, and the parent node will be 0.
- The second call occurs at the end of the list of nodes to be processed and is made to generate the first child node of the node that is currently processed, and which will act as a trigger to generate the rest of the node's child nodes. In this case, the arguments for the call are the same module, the variables of the current node renamed (discussed later), an *empty* list of terms, the identifier used in the current node to generate the fresh variables, and the number of solutions 0, since it is the first node that we generate from the current one. The sub-branch will therefore be 0, the parent node will be the current node (its identifier), and the depth will be 2.

As the nodes of the same level are at the beginning of the list and all the child nodes behind them (at the end the list will contain all the nodes of the tree, ordered as they have been generated, by levels), we make sure that those of the current level are processed first, since the algorithm processes the nodes from beginning to end of the list.

```
eq (M, T1, N1, NARS1 ;
   {N2:NzNat, {T2, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5} ;
   NARS2,
   {N6, {T3, Ty2, C2, Q4, S3, S4, Q5}, N7, N2:NzNat, Q6, N8} ;
   NARS3, unbounded, N9) =
     (M, T1, N1 + 2, NARS1 ;
       {
         N2:NzNat,
         {T2, Ty1, C1, Q1, S1, S2, Q2},
         N3, N4, Q3, N5
       } ; NARS2 ;
       {
         N6,
         {
           applySub(T3,rename((getVars(T3), getRangeVars(S3),
                    getRangeVars(S4)), N9)),
           Ty2, C2, Q4,
           applySub(S3,rename((getVars(T3), getRangeVars(S3),
                                getRangeVars(S4)), N9)),
           applySub(S4,rename((getVars(T3), getRangeVars(S3),
                                getRangeVars(S4)), N9)),
           '$
         },
         N7, N2:NzNat, '$, N8
       },
       {
         N1,
         metaNarrowingApply(M, T2, empty, Q6, N7 + 1),
         N7 + 1, N2:NzNat, Q6, N8
       } ; NARS3 ;
       {
         N1 + 1,
         metaNarrowingApply(M, applySub(T3, rename((getVars(T3),
                             getRangeVars(S3),getRangeVars(S4)),N9)),
                             empty, Q5, 0),
         0, N6, Q5, N8 + 1
       },
     unbounded,
     N9 + getVarsNumber((getVars(T3),getRangeVars(S3),getRangeVars(S4)))) .
```

Figure 22: Equation that defines the generation of levels higher than one of the reachability tree when the depth parameter takes unbounded as a value in the GET-REACHABILITY-TREE module.

Figure 22 shows the code that defines the equation used when the depth parameter takes the value *unbounded* to generate levels higher than one. We look for a node that is at the beginning of the list of nodes to be processed, which we will refer to as the current node, and we also look for its parent node in the list of nodes already processed. Once we find both, the process is similar to the previous one, since we also want to generate the nodes following the current one at the same level, and the first child node of the current node. The biggest difference is that now, instead of taking the arguments of node 0 to generate the nodes of the same level, we will take the arguments of the parent node that we have looked for in the list of nodes already processed. At the code level, when we find the current node and its parent node in the lists, we move the current node to the end of the list of processed nodes, and in turn make two calls to *metaNarrowingApply*:

- The first call occurs at the beginning of the list of nodes to be processed and is used to generate the next node that is at the same level as the current one. The parameter passing is very similar to what we have seen in the previous equation, only instead of considering the arguments of node 0, the arguments of the parent node found are used.
- The second call occurs at the end of the list of nodes to be processed and is made to generate the first child node of the node that is currently processed just as it happened in the previous equation. In this case, the passing of parameters occurs in a similar way.

In Figure 23 we show the code to define the equation used to generate the rest of the levels when the maximum depth parameter takes a non-zero natural as value. It works in a very similar way to the equation seen for when the maximum depth parameter takes the value *unbounded*, only in this case, the *if_then_else_fi* operator is used. The condition for the conditional operator is that the current node must have a depth less than or equal to that indicated by the user as maximum depth, since if it is greater, we are not interested in the current node. Neither will the following nodes of the same level or their children be of interest, so it is not necessary to generate them.

```
eq (M, T1, N1, NARS1 ;
   {N2:NzNat, {T2, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5} ;
   NARS2,
   {N6, {T3, Ty2, C2, Q4, S3, S4, Q5}, N7, N2:NzNat, Q6, N8} ;
   NARS3, MaxDepth:NzNat, N9) =
   if N8 <= MaxDepth:NzNat then
     (M, T1, N1 + 2, NARS1 ;
       {
         N2:NzNat,
         {T2, Ty1, C1, Q1, S1, S2, Q2},
         N3, N4, Q3, N5
       } ; NARS2 ;
       {
         N6,
         {
           applySub(T3,rename((getVars(T3), getRangeVars(S3),
                    getRangeVars(S4)), N9)),
           Ty2, C2, Q4,
           applySub(S3,rename((getVars(T3), getRangeVars(S3),
                               getRangeVars(S4)), N9)),
           applySub(S4,rename((getVars(T3), getRangeVars(S3),
                               getRangeVars(S4)), N9)),
           '$
         },
         N7, N2:NzNat, '$, N8
       },
       {
         N1,
         metaNarrowingApply(M, T2, empty, Q6, N7 + 1),
         N7 + 1, N2:NzNat, Q6, N8
       } ; NARS3 ;
       {
         N1 + 1,
         metaNarrowingApply(M, applySub(T3, rename((getVars(T3),
                            getRangeVars(S3),getRangeVars(S4)),N9)),
                            empty, Q5, 0),
         0, N6, Q5, N8 + 1
       },
     MaxDepth:NzNat,
     N9 + getVarsNumber((getVars(T3),getRangeVars(S3),getRangeVars(S4))))
   else
     NARS1 ;
     {N2:NzNat, {T2, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5} ;
     NARS2
   fi .
```

Figure 23: Figure 20: Equation that defines the generation of levels higher than one of the reachability tree when the depth parameter takes a non-zero natural as a value in the GET-REACHABILITY-TREE module.

As before, we look for the first node in the list of nodes to be processed, and its parent node in the list of nodes to be processed. If the condition is met, the behavior is very similar to what we have already seen when the maximum depth parameter is *unbounded*, but if the condition is not met, what we do this time is to eliminate the current node and return the list of processed nodes, which will be the solution, since we will have reached the maximum depth desired by the user.

```
eq (M, T1, N1, NARS1, nil, MaxDepth, N9) = NARS1 .
eq (M, T1, N1, NARS1, {N2, failure, N3, N4, Q1, N5} ; NARS2, MaxDepth, N9)
    = (M, T1, N1, NARS1, NARS2, MaxDepth, N9) .
```

Figure 24: Equations to manage an empty list of nodes to process and nodes not found in the GET-REACHABILITY-TREE module.

At this point, two more equations are necessary, which are shown in Figure 24. In the first equation, we consider the possibility that the list of nodes to be processed remains empty (*nil*), for example, if all possible reachable nodes have already been found. In that case, we will not be able to generate more nodes of the tree, so the complete tree is defined with the list of nodes already processed, which will be returned to the user. The second equation considers those cases in which the call to the *metaNarrowingApply* command to generate a new node returns the constant *failure*. This constant indicates that a reachability solution could not be found for the given term and solution number, so from there it does not make sense to continue generating nodes that derive from the same parent node (same term), and the node containing that constant can be removed, leaving the rest of the node lists as they were.

3.4. Variable renaming

In some of the equations seen in the previous section, there are calls to functions such as *rename*, *applySub*, *getVars*, *getRangeVars*, etc. All of them are functions defined by equations in a module that we have called *CONVERT-VARIABLES*, and they are used to rename the fresh variables that Maude generates.

The main reason why it is necessary to rename the fresh variables generated by Maude is because when making the calls to *metaNarrowingApply*, it is only allowed to indicate an identifier that Maude will not use to name the fresh variables that will be generated in the solution, that is, a new node in the tree. This prevents a child node from repeating variable names that are already in the parent node but cannot prevent a grandchild node from doing so. When generating the reachability tree from a term, it is not a problem, but later, when we want to extend the algorithm to narrowing, that is, when we want to check in each node if the term unifies with an objective term and to compute the solution, it will be necessary to collect the variables and substitutions along the way from the solution node to the initial node, and there will be clashes between variables.

```
protecting META-LEVEL .
protecting CONVERSION .

var F : Qid .   var V : Variable .   var GT : GroundTerm .
var GNTL : NeGroundTermList .   var NTL : NeTermList .   var T : Term .
var N : Nat .   vars TL TL' TL'' : TermList .   vars SB SB' : Substitution .
```

Figure 25: Imports and variable declarations in the CONVERT-VARIABLES module.

First, it is necessary to declare the imports and declare the variables that will be used later, although as previously mentioned, variable declarations are something that is usually done on the fly. In Figure 25, we can see that the *CONVERT-VARIABLES* module imports two modules: the *META-LEVEL* module and the *CONVERSION* module. The *CONVERT-VARIABLES* module itself, although not explicitly stated, is imported by the *GET-REACHABILITY-TREE* module, and it is for this reason that it can also use functions defined in the *META-LEVEL* module. In that same figure, we show the declared variables. The name of its types is self-explanatory enough, although we will explain each variable as we go in the defined equations.

```
op rename : TermList Nat -> Substitution .
eq rename(empty,N) = none .
eq rename((TL,V,TL',V,TL''),N) = rename((TL,V,TL',TL''),N) .
eq rename((V,TL),N) = V <- qid("$" + string(N,10) + ":" +
string(getType(V))) ; rename(TL,N + 1) [owise] .
```

Figure 26: The rename operator and its associated equations in the CONVERT-VARIABLES module.

In Figure 26, we show the declaration of the *rename* operator, which receives a list of terms and a natural number and returns a substitution. The equations used to define its behavior are also shown. If the list of terms is empty, the substitution returns will be the empty substitution, represented by *none*. Otherwise, the list of terms received will be scanned looking for variables. When the first variable is found, it is renamed, replacing its original *Qid* with the symbol *$* followed by the natural number also received, and keeping its type, which is specified after the *Qid*, preceded by the *:* symbol . The natural number received is increased by one, and a recursive call is made that continues through the list of terms, following the same process until there is nothing left in the list of terms. Throughout this, a data structure is saved that follows the form of the substitutions in Maude, so that the substitution resulting from performing all the renaming of the variables in the list of terms can finally be returned. Of course, terms that are not variables are ignored, and if the list of terms does not contain variables, *none* is returned as well, as before. Note that in case there is a repeated variable in the list of terms received, before carrying out the whole process explained, the repetition of said variable is eliminated, leaving it only once in the list of terms.

```
op getVars : Term -> TermList .
eq getVars(GT) = empty .
eq getVars(V) = V .
eq getVars(F[NTL]) = getVars(NTL) [owise] .

op getVars : TermList -> TermList .
eq getVars(empty) = empty .
eq getVars((T,GNTL)) = getVars(T) .
eq getVars((T,NTL)) = (getVars(T),getVars(NTL)) [owise] .
```

Figure 27: The getVars operators and their associated equations in the CONVERT-VARIABLES module.

In Figure 27, we can find the code snippet used to define the *getVars* operators and their associated equations to define their behavior. These operators are responsible for returning a list of terms that contains all the variables found in the term or the list of terms received, respectively. In case of receiving a ground term (a constant) or an empty list of terms, the *empty* list is returned, but if the term received is a variable, the variable itself is returned. If the term received contains arguments, it will be necessary to search for variables within the arguments, discarding the root of the term. If a term accompanied by a non-empty list of ground terms is received, the list is ignored, and variables are searched recursively in the term. Finally, if a term accompanied by a non-empty list of terms is received, the variables must be searched recursively both in the term and in the list of terms.

```
op getRangeVars : Substitution -> TermList .
eq getRangeVars((none).Substitution) = empty .
eq getRangeVars(((V <- T) ; SB)) = (getVars(T),getRangeVars(SB)) .
```

Figure 28: The getRangeVars operator and its associated equations in the CONVERT-VARIABLES module.

We have also defined a *getRangeVars* operator, with its corresponding equations, as can be seen in Figure 28, and it is used to obtain the variables of a substitution, instead of a list of terms or a term, although it ends up calling the function that we had defined to find variables in a term, once it has separated the binding of the substitutions in variables and terms.

```
op getVarsNumber : Term -> Nat .
eq getVarsNumber(GT) = 0 .
eq getVarsNumber(V) = 1 .
eq getVarsNumber(F[NTL]) = get-
VarsNumber(NTL) [owise] .

op getVarsNumber : TermList -> Nat .
eq getVarsNumber((T,GNTL)) = getVarsNumber(T) .
eq getVarsNumber((T,NTL)) = get-
VarsNumber(T) + getVarsNumber(NTL) [owise] .
```

Figure 29: The getVarsNumber operator and its associated equations in the CONVERT-VARIABLES module.

In addition, a function *getVarsNumber* (operator plus equations) has been defined that is responsible for counting the number of variables in a list of terms. It is very similar to the *getVars* function, but instead of getting the variables, it just counts them. The code snippet used for this can be found in Figure 29.

```
op applySub : TermList Substitution -> TermList .
eq applySub(V,(V <- T) ; SB) = T .
eq applySub(F[TL], SB) = F[applySub(TL, SB)] .
eq applySub((T,NTL),SB) = (applySub(T,SB), applySub(NTL,SB)) .
eq applySub(T,SB) = T [owise] .

op applySub : Substitution Substitution -> Substitution .
eq applySub((none).Substitution,SB) = none .
eq applySub(V <- T ; SB,SB') =  V <- applySub(T,SB') ; applySub(SB,SB') .
```

Figure 30: The applySub operators and their associated equations in the CONVERT-VARIABLES module.

Finally, as can be seen in Figure 30, two *applySub* operators have been defined, along with their equations, to be able to apply a substitution to a list of terms, resulting in a new list of terms, or to apply a substitution to another substitution, also obtaining a new one as a result.

All the functions defined in this module are used in equations of the *GET-REACHABILITY-TREE* module, since the variables of each node will be renamed just before generating the nodes derived from it.

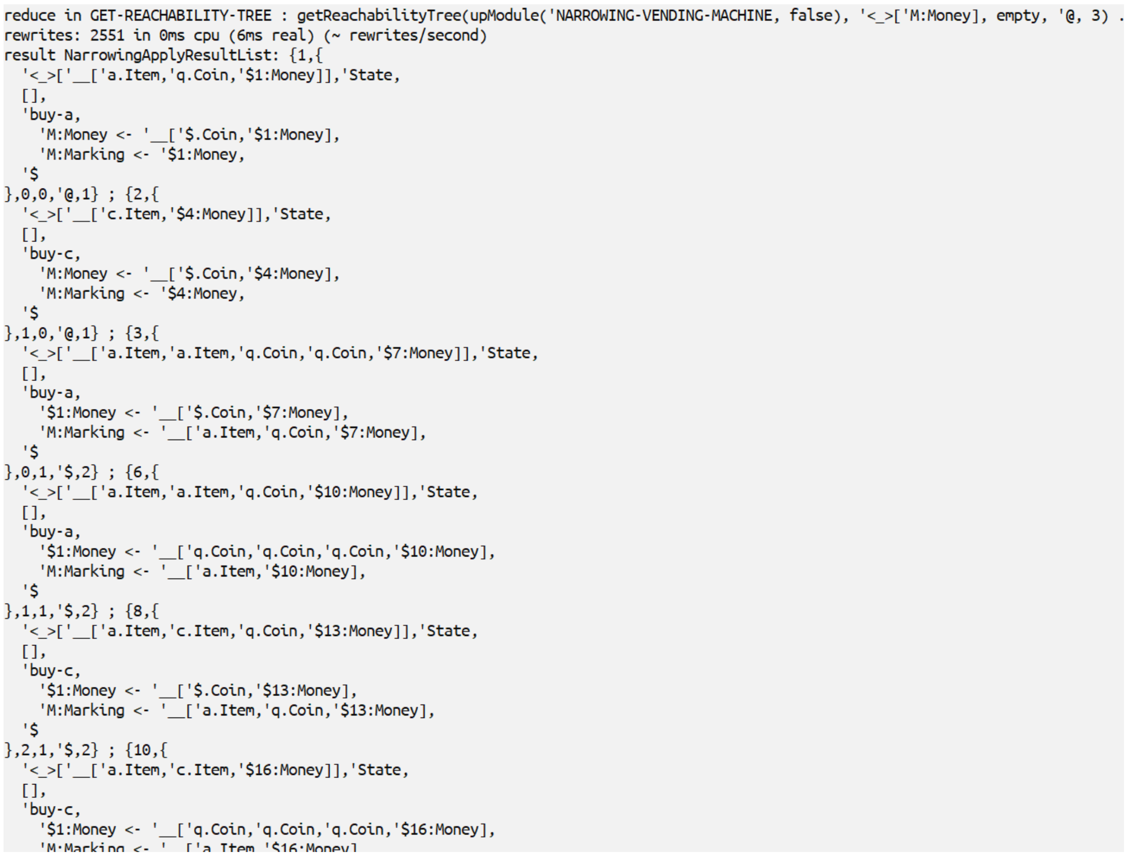
3.5. Testing the tree generation

Once the module to generate the reachability tree from a term has been implemented, we proceed to carry out some tests using different modules and terms, as well as variations of the rest of the parameters that the defined *getReachabilityTree* command admits. To check the goodness of the results obtained, we compare the nodes returned when executing the command with the nodes returned when using the *Narval* online tool [1], which allows a graphical view of the reachability tree generated from a term using a module to rewrite and normalize.

The first module to consider is the one that has been used to carry out small tests throughout the implementation: the *NARROWING-VENDING-MACHINE* module, taken directly from the official Maude examples [13]. Its definition is shown in Figure 13. This module defines a vending machine that handles dollars (*$*) and quarters of a dollar (*q*). One rule defines the sale of cakes (*c*) for one dollar, and another the sale of apples (a) for three-quarters of a dollar. An equation defines the equivalence between 4 quarters of a dollar and one dollar. First, we try to generate the reachability tree with depth two from the initial state < *M:Money* >, using the command defined as follows:

```
reduce in GET-REACHABILITY-TREE :
   getReachabilityTree(upModule('NARROWING-VENDING-MACHINE,
   false), '<_>['M:Money], empty, '@, 2) .
```

The result obtained is as expected. The nodes of the first three levels of the reachability tree are generated correctly, as partially shown in Figure 31. Note that a list of nodes is returned, in which the previously explained can be seen more clearly. Each node contains an identifier (natural number), the term reached together with its type, a context, the name of the rule used to perform the rewrite step, the necessary substitutions to perform the unification, the *Qid* used to name the fresh variables in the current node and its parent node (both will be $ because the variables have been renamed, except for the parent of the first level nodes, which will have the *Qid* used by the user in the command). Several natural numbers are also returned indicating the sub-branch and level at which this node is in the tree, as well as the identifier of its parent node.



```
reduce in GET-REACHABILITY-TREE : getReachabilityTree(upModule('NARROWING-VENDING-MACHINE, false), '<_>['M:Money], empty, '@, 3) .
rewrites: 2551 in 0ms cpu (6ms real) (~ rewrites/second)
result NarrowingApplyResultList: {1,{
  '<_>['__['a.Item,'q.Coin,'$1:Money]],'State,
  [],
  'buy-a,
    'M:Money <- '__['$.Coin,'$1:Money],
    'M:Marking <- '$1:Money,
  '$
},0,0,'@,1} ; {2,{
  '<_>['__['c.Item,'$4:Money]],'State,
  [],
  'buy-c,
    'M:Money <- '__['$.Coin,'$4:Money],
    'M:Marking <- '$4:Money,
  '$
},1,0,'@,1} ; {3,{
  '<_>['__['a.Item,'a.Item,'q.Coin,'q.Coin,'$7:Money]],'State,
  [],
  'buy-a,
    '$1:Money <- '__['$.Coin,'$7:Money],
    'M:Marking <- '__['a.Item,'q.Coin,'$7:Money],
  '$
},0,1,'$,2} ; {6,{
  '<_>['__['a.Item,'a.Item,'q.Coin,'$10:Money]],'State,
  [],
  'buy-a,
    '$1:Money <- '__['q.Coin,'q.Coin,'q.Coin,'$10:Money],
    'M:Marking <- '__['a.Item,'$10:Money],
  '$
},1,1,'$,2} ; {8,{
  '<_>['__['a.Item,'c.Item,'q.Coin,'$13:Money]],'State,
  [],
  'buy-c,
    '$1:Money <- '__['$.Coin,'$13:Money],
    'M:Marking <- '__['a.Item,'q.Coin,'$13:Money],
  '$
},2,1,'$,2} ; {10,{
  '<_>['__['a.Item,'c.Item,'$16:Money]],'State,
  [],
  'buy-c,
    '$1:Money <- '__['q.Coin,'q.Coin,'q.Coin,'$16:Money],
    'M:Marking <- '__['a.Item,'$16:Money],
```

Figure 31: Example 1 of the generation of the reachability tree in the NARROWING-VENDING-MACHINE module.

The execution is too long, so it cannot be seen completely in an image, so we show the tree that is generated in Narval using the same module and term, and limiting ourselves to the three levels indicated as maximum depth, in Figure 32. The nodes coincide with those generated in our execution, except for the name of the fresh variables, which in Narval have not been renamed. Furthermore, the order in which the nodes are shown in our execution coincides with the generation by levels of the same tree shown in Narval, as explained previously.

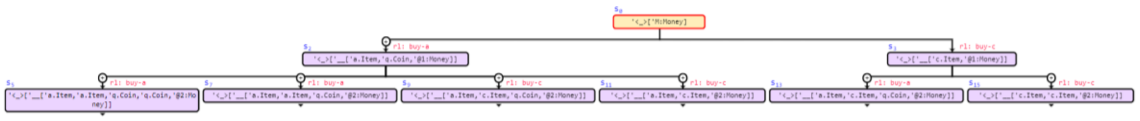
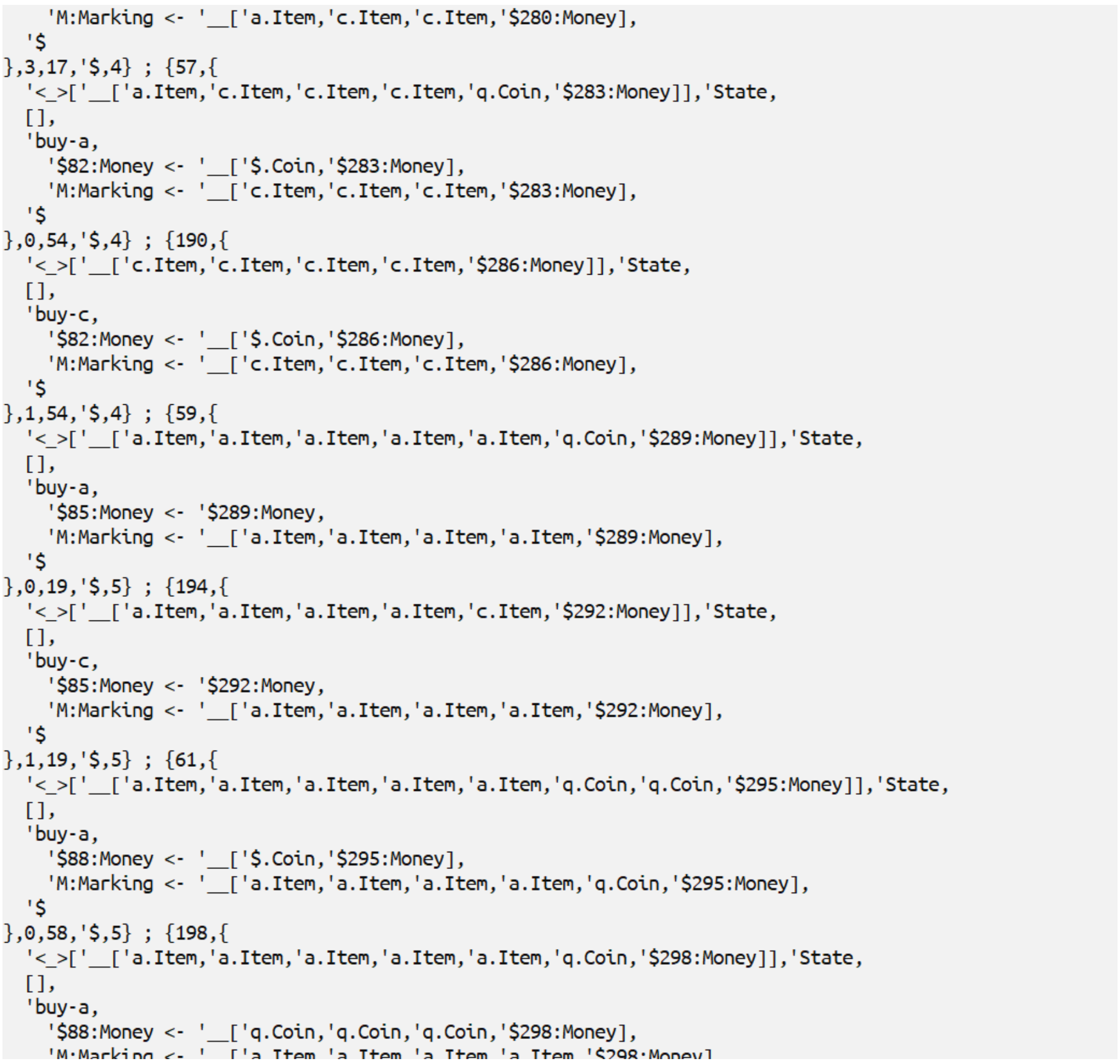

S0
'<_>['M:Money]
S1
rl: buy-a
'<_>['__['a.Item,'q.Coin,'@1:Money]]
S3
rl: buy-c
'<_>['__['c.Item,'@1:Money]]
S5
rl: buy-a
'<_>['__['a.Item,'a.Item,'q.Coin,'q.Coin,'@2:Money]]
S7
rl: buy-a
'<_>['__['a.Item,'a.Item,'q.Coin,'@2:Money]]
S9
rl: buy-c
'<_>['__['a.Item,'c.Item,'q.Coin,'@2:Money]]
S11
rl: buy-c
'<_>['__['a.Item,'c.Item,'@2:Money]]
S13
rl: buy-a
'<_>['__['a.Item,'c.Item,'q.Coin,'@2:Money]]
S15
rl: buy-c
'<_>['__['c.Item,'c.Item,'@2:Money]]

Figure 32: Generation of the reachability tree using Narval and the NARROWING-VENDING-MACHINE module.

If instead of generating only 2 levels we want to generate some more (for example, 5), we only have to change the parameter that indicates the maximum depth:

```
reduce in GET-REACHABILITY-TREE :
    getReachabilityTree(upModule('NARROWING-VENDING-MACHINE,
    false), '<_>['M:Money], empty, '@, 5) .
```

As we might expect, the result takes a little longer to return, although the execution time is still practically imperceptible (70ms). In Figure 33 we show a piece of the result obtained when executing the command. Specifically, some nodes at levels 4 and 5 of the generated tree can be observed.



```
    'M:Marking <- '__['a.Item,'c.Item,'c.Item,'$280:Money],
  '$
},3,17,'$,4} ; {57,{
  '<_>['__['a.Item,'c.Item,'c.Item,'c.Item,'q.Coin,'$283:Money]],'State,
  [],
  'buy-a,
    '$82:Money <- '__['$.Coin,'$283:Money],
    'M:Marking <- '__['c.Item,'c.Item,'c.Item,'$283:Money],
  '$
},0,54,'$,4} ; {190,{
  '<_>['__['c.Item,'c.Item,'c.Item,'c.Item,'$286:Money]],'State,
  [],
  'buy-c,
    '$82:Money <- '__['$.Coin,'$286:Money],
    'M:Marking <- '__['c.Item,'c.Item,'c.Item,'$286:Money],
  '$
},1,54,'$,4} ; {59,{
  '<_>['__['a.Item,'a.Item,'a.Item,'a.Item,'a.Item,'q.Coin,'$289:Money]],'State,
  [],
  'buy-a,
    '$85:Money <- '$289:Money,
    'M:Marking <- '__['a.Item,'a.Item,'a.Item,'a.Item,'$289:Money],
  '$
},0,19,'$,5} ; {194,{
  '<_>['__['a.Item,'a.Item,'a.Item,'a.Item,'c.Item,'$292:Money]],'State,
  [],
  'buy-c,
    '$85:Money <- '$292:Money,
    'M:Marking <- '__['a.Item,'a.Item,'a.Item,'a.Item,'$292:Money],
  '$
},1,19,'$,5} ; {61,{
  '<_>['__['a.Item,'a.Item,'a.Item,'a.Item,'a.Item,'q.Coin,'q.Coin,'$295:Money]],'State,
  [],
  'buy-a,
    '$88:Money <- '__['$.Coin,'$295:Money],
    'M:Marking <- '__['a.Item,'a.Item,'a.Item,'a.Item,'q.Coin,'$295:Money],
  '$
},0,58,'$,5} ; {198,{
  '<_>['__['a.Item,'a.Item,'a.Item,'a.Item,'a.Item,'q.Coin,'$298:Money]],'State,
  [],
  'buy-a,
    '$88:Money <- '__['q.Coin,'q.Coin,'q.Coin,'$298:Money],
    'M:Marking <- '__['a.Item,'a.Item,'a.Item,'a.Item,'$298:Money]
```

Figure 33: Example 2 of the generation of the reachability tree in the NARROWING-VENDING-MACHINE module.

The other module that we use for the examples in this section can be seen in Figure 34. In this case, a behavior is defined with successors and predecessors of numbers. The first equation [*E1*] defines the additive identity, the second [*E2*] the sum of a number with the successor of another, the third [*E3*] the sum of a number with the predecessor of another, and the fourth [*E4*] and fifth [*E5*] the simplification of the application of the predecessor function to a successor and vice versa. A rule that defines state transitions is also defined, in which if a state is found with the successor of a number *X* and another number *Y*, a transition is made to a new state with the predecessor of the successor of *X* and the successor from *Y*, that is, in a state with two natural numbers, one would be subtracted from the first number and one would be added to the second, using the operations successor and predecessor.

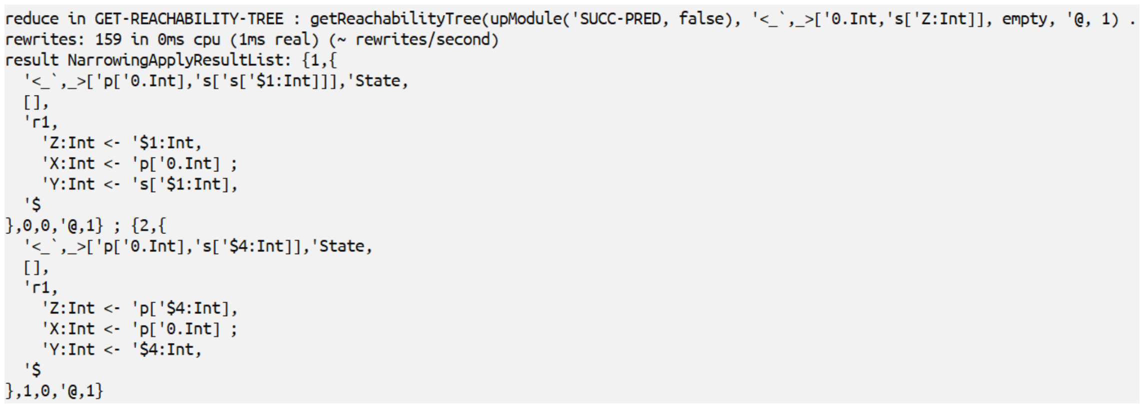
```
mod SUCC-PRED is
   sorts Int State .

   op 0 : -> Int [ctor] .
   ops s p : Int -> Int .
   op _+_ : Int Int -> Int [assoc comm prec 30] .
   op <_,_> : Int Int -> State [ctor] .

   vars X Y : Int .

   eq [E1] : X + 0 = X [variant] .
   eq [E2] : X + s(Y) = s(X + Y) [variant] .
   eq [E3] : X + p(Y) = p(X + Y) [variant] .
   eq [E4] : p(s(X))= X [variant] .
   eq [E5] : s(p(X)) = X [variant] .
   rl [r1] : < s(X),Y > => < p(s(X)),s(Y) > [narrowing] .
endm
```

Figure 34: The SUCC-PRED module in Maude.

On this occasion, we can start by generating only the first level of the reachability tree, starting from the term *<0, 0 + s (Z: Int)>*, which will be directly converted to its normal form using the defined equations, that is, *<0 , s (Z: Int)>*. We can use the following command for this (Note that the term is expressed in prefix form, since we are using the meta-level):

```
reduce in GET-REACHABILITY-TREE :

    getReachabilityTree(upModule('SUCC-PRED, false),

    '<_`,_>['0.Int,'s['Z:Int]], empty, '@, 1) .
```

Two nodes are successfully generated, as shown in Figure 35. In Figure 36, the generation of the first two levels of the reachability tree is shown, using the same module and term seen, as well as some nodes of level three. We can see that the nodes of the first level of the tree are the same that we have obtained in the execution with our implementation. The only exception, once again, is the names of the fresh variables.



```
reduce in GET-REACHABILITY-TREE : getReachabilityTree(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['Z:Int]], empty, '@, 1) .
rewrites: 159 in 0ms cpu (1ms real) (~ rewrites/second)
result NarrowingApplyResultList: {1,{
  '<_`,_>['p['0.Int],'s['s['$1:Int]]],'State,
  [],
  'r1,
    'Z:Int <- '$1:Int,
    'X:Int <- 'p['0.Int] ;
    'Y:Int <- 's['$1:Int],
  '$
},0,0,'@,1} ; {2,{
  '<_`,_>['p['0.Int],'s['$4:Int]],'State,
  [],
  'r1,
    'Z:Int <- 'p['$4:Int],
    'X:Int <- 'p['0.Int] ;
    'Y:Int <- '$4:Int,
  '$
},1,0,'@,1}
```

Figure 35: Example 1 of the generation of the reachability tree in the SUCC-PRED module.

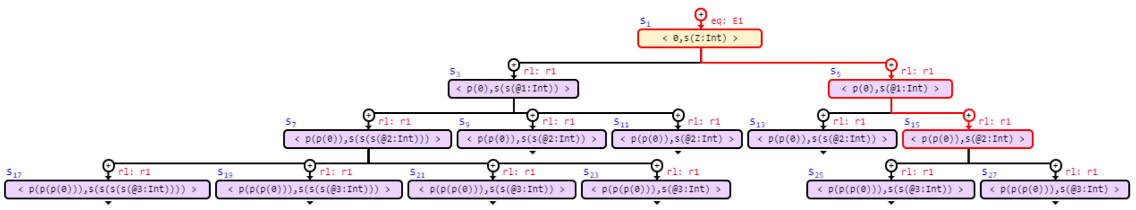


Figure 36: Generation of the reachability tree using Narval and the SUCC-PRED module.

Now we can increase the number of levels to generate, for example, to three, and the solution obtained should be like the one obtained with Narval, although we have not expanded some nodes of level 3. To do this, we simply changed the number that indicated the number of levels generated from 1 to 3, as we did previously with the commands used for the vending machine. In Figure 37 we show the result obtained when executing that. Comparing the complete execution with the result obtained in Narval, a coincidence is found again.

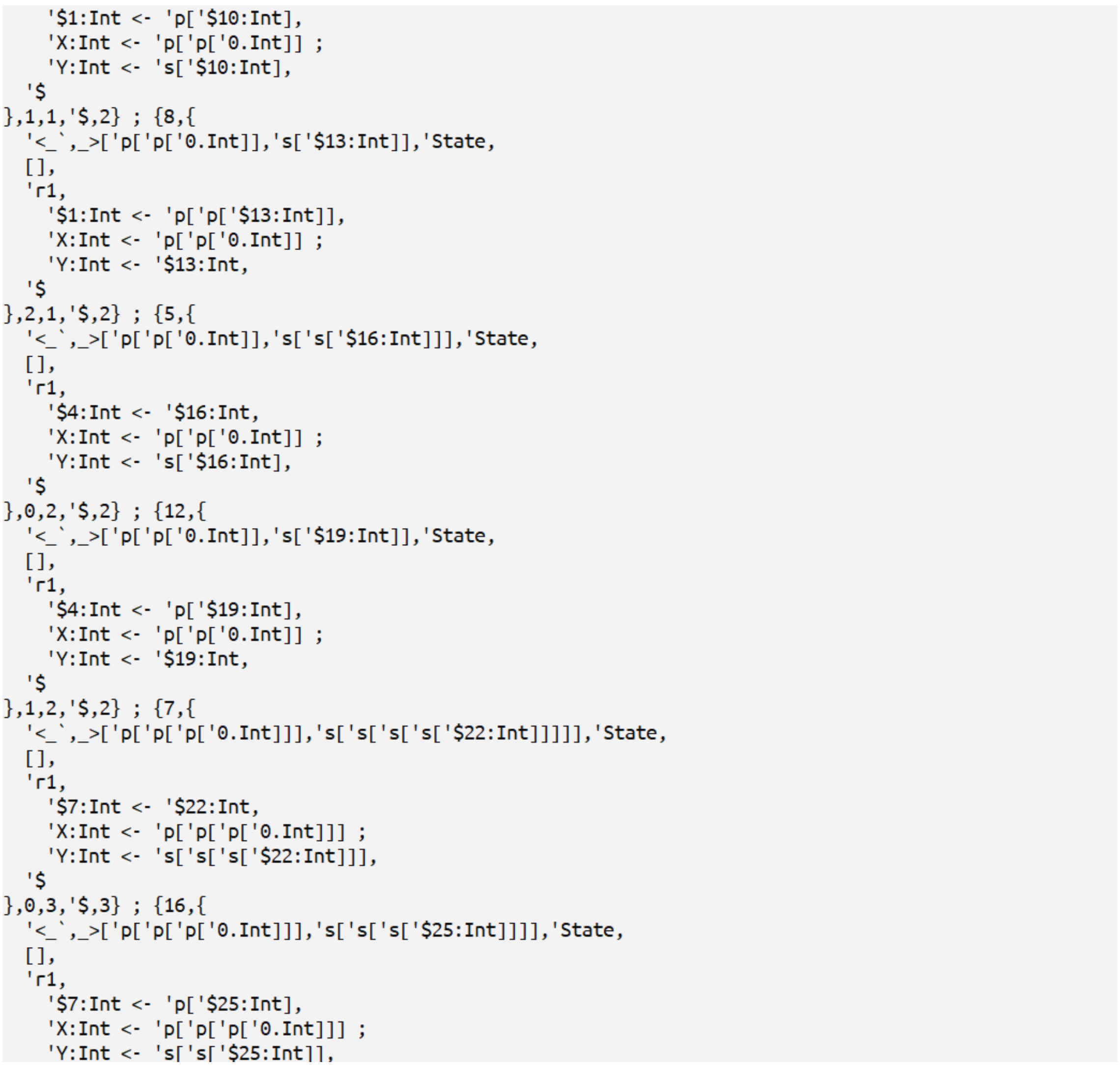

```
    '$1:Int <- 'p['$10:Int],
    'X:Int <- 'p['p['0.Int]] ;
    'Y:Int <- 's['$10:Int],
  '$
},1,1,'$,2} ; {8,{
  '<_`,_>['p['p['0.Int]],'s['$13:Int]],'State,
  [],
  'r1,
    '$1:Int <- 'p['p['$13:Int]],
    'X:Int <- 'p['p['0.Int]] ;
    'Y:Int <- '$13:Int,
  '$
},2,1,'$,2} ; {5,{
  '<_`,_>['p['p['0.Int]],'s['s['$16:Int]]],'State,
  [],
  'r1,
    '$4:Int <- '$16:Int,
    'X:Int <- 'p['p['0.Int]] ;
    'Y:Int <- 's['$16:Int],
  '$
},0,2,'$,2} ; {12,{
  '<_`,_>['p['p['0.Int]],'s['$19:Int]],'State,
  [],
  'r1,
    '$4:Int <- 'p['$19:Int],
    'X:Int <- 'p['p['0.Int]] ;
    'Y:Int <- '$19:Int,
  '$
},1,2,'$,2} ; {7,{
  '<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],'State,
  [],
  'r1,
    '$7:Int <- '$22:Int,
    'X:Int <- 'p['p['p['0.Int]]] ;
    'Y:Int <- 's['s['s['$22:Int]]],
  '$
},0,3,'$,3} ; {16,{
  '<_`,_>['p['p['p['0.Int]]],'s['s['s['$25:Int]]]],'State,
  [],
  'r1,
    '$7:Int <- 'p['$25:Int],
    'X:Int <- 'p['p['p['0.Int]]] ;
    'Y:Int <- 's['s['$25:Int]],
```

Figure 37: Example 2 of the generation of the reachability tree in the SUCC-PRED module.

4. Narrowing using the reachability tree

In this chapter we describe a possible implementation to perform *standard narrowing modulo an equational theory (∑, E ∪ B)*, using the previous code as a base to generate the reachability tree. In Section 4.1 we present the steps considered to implement narrowing in Maude. In Section 4.2, we detail the types, subtypes, data structures and variables needed to implement the algorithm. In Section 4.3, we present the *getNarrowingSolutions* command, and its initial behavior, including node generation. In Section 4.4, we analyze how the unification step is performed in the search for narrowing solutions at each node. In Section 4.5, we detail how the solutions are created from the unifiers that have been obtained. In Section 4.6, we describe the iterative process of the algorithm, as well as when it comes to an end. Finally, in Section 4.7, we show some examples of using the algorithm, using different modules and terms.

4.1. Specification of the narrowing steps to implement

In Section 2.10 we have explained the theoretical definition of narrowing, and now, in order to implement it, it is necessary to draw up a plan of steps to follow, based precisely on the theoretical algorithm, and adapting it to Maude programming, creating a new module named *GET-NARROWING-SOLUTIONS*. These steps are:

1. Define the required types, subtypes, data structures, and variables.
2. Define the command that the user will use and start the node generation.
3. Try to generate a new node, reusing the code detailed in the previous chapter.
4. Try to unify the target term indicated by the user with the term obtained in the generated node.
5. If it is possible to unify, go backwards through the branch of the tree in which the node is located to compose the substitutions, computing a new solution.
6. Reduce the computed substitutions to their normal form, using the equations defined in the module received as an argument.
7. Perform steps 3 - 6 iteratively, until a termination condition is met.

4.2. Types, subtypes, data structures and variables

To implement the algorithm, in addition to the types, subtypes, and data structures previously declared in the *GET-REACHABILITY-TREE* module, some new ones will be needed. Its declaration is shown in Figure 38.

First of all, we need a data structure that stores the global state of the algorithm. We have given this structure the name of *NarrowingState*, and it contains the following elements enclosed between the symbols "(" and ")", and separated by commas:

- The initial term indicated by the user, that is, the term from which everything will be generated.
- A *Qid* that represents the search arrow, which we will explain later.
- The target term indicated by the user, which will be used to determine whether each generated node is a solution.

- A *Bound* element that will mark the maximum number of desired solutions, also indicated by the user.
- A natural number used as a counter for the number of solutions, to number each new solution.
- A *Qid* that indicates the variable identifier used in the initial and target terms, to avoid clashes later.
- A list of the solutions that have been obtained.
- A “substate” generated by the code defined in the previous module, that is, the state of the reachability tree that is being generated.

```
sorts NarrowingResultStructure NarrowingResultList NarrowingState Step .
subsort UnificationPair < NarrowingResultStructure
        < NarrowingResultList < NarrowingState .

ops genNode unify genSolutions : -> Step .
op (_,_,_,_,_,_,_) : Term Qid Term Bound Nat Qid NarrowingResultList State ->
                   NarrowingState .
op getNarrowingSolutions : Module Term Qid Term TermList Qid Bound Bound ->
                           NarrowingResultList  .
op {_,_,_,_} : Nat Term Substitution Substitution -> NarrowingResultStructure .
op calculateCumulativeSub : Nat Term Substitution Substitution ->
                            NarrowingResultStructure .
op noSolution : -> NarrowingResultList [ctor] .
op _;_ : NarrowingResultList NarrowingResultList ->
         NarrowingResultList [assoc id: noSolution] .
op reduceSub : Module Substitution -> Substitution .
op iterativeMetaVariantUnify : Module UnificationProblem TermList Qid Nat ->
                               NarrowingResultList .
```

Figure 38: Sorts, subsorts and related operators in the GET-NARROWING-SOLUTIONS module.

The list of solutions computed by the narrowing algorithm is called *NarrowingResultList*, and it is a subtype of *NarrowingState*, since it is the only thing that we will be interested in returning to the user once the algorithm finishes. It is an ordered list, so it is associative and has an identity element (*noSolution*), but it is not commutative. The symbol “;” is the one chosen to concatenate elements of this list. Each of the computed solutions is also a data structure of the form “{_,_,_,_}”, called *NarrowingResultStructure*, and in which each of the underscores is:

- A natural number, identifying the solution number.
- The term associated with that solution, that is, the term found that unifies with the target term, which coincides with the term of the node in which the solution is found.
- The substitution that allows us to get from the initial term to the solution term.
- The substitution used to unify the solution term with the target term.

The type *NarrowingResultStructure* is a subtype of *NarrowingResultList*, since the list may contain only one solution, that is, a single structure of that type. Furthermore, the structure for each solution uses the *UnificationPair* type as a subtype. That type is the one returned by Maude when performing the unification, and it is already predefined in the *META-LEVEL* module. There are some other operators defined, but they will be discussed later, as they are used only in some specific steps of the algorithm.

```
var M : Module .   vars T1 T2 T3 T4 T5 T6 : Term .   var TList : TermList .
vars Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 SearchQid : Qid .  var UP : UnificationPair .
vars N1 N2 N3 N4 N5 N6 N7 N8 N9 N10 N11 : Nat .   vars Ty1 Ty2 : Type .
vars C1 C2 : Context .   vars S1 S2 S3 S4 S5 S6 : Substitution .
vars MaxDepth MaxSol : Bound .   vars NRS1 NRS2 : NarrowingResultList .
vars NARS1 NARS2 NARS3 NARS4 : NarrowingApplyResultList .
vars V1 V2 : Variable .
```

Figure 39: Variables declared in the GET-NARROWING-SOLUTIONS module.

In order to define the equations that will determine the behavior of each of the operators, it is necessary to previously define the variables to be used. Figure 39 shows the necessary variables, which are of types that we have already seen before and of the new types presented in this section.

4.3. The *getNarrowingSolutions* command and node generation

The command that the user is expected to use to invoke the algorithm that we define to perform narrowing is the *getNarrowingSolutions* command, and its needed input parameters are:

- The module that contains the rules, variants and axioms that will be used to generate the reachability tree, as well as to normalize the terms and to perform unification when necessary.
- The initial term, from which the reachability tree will be generated to search for possible solutions.
- A *Qid* that can take three different values. Using the value *'1* we ask the command to return only the solutions found in the first level of the tree (that is, those that only imply a rewrite step). If we use the value *'+*, we ask the command to return only the solutions that imply more than one rewrite step, and if we use the value *'**, we ask the command to return solutions that involve any number of rewrite steps. These three values emulate the narrowing search arrows $\rightsquigarrow^1$, $\rightsquigarrow^+$ and $\rightsquigarrow^*$ respectively.
- The target term, that is, the term we want to achieve by rewriting from the initial term using the indicated module. Each term reached by rewriting that unifies with the target term, will lead to one or more solutions, depending on whether there is one or more unifiers.
- A list of terms, which at the moment has no use, and whose value will be subsequently set to *empty*. It will be used later to extend the algorithm.
- A *Qid* that will be used to indicate the variable identifiers in the initial and target terms.

- The maximum depth of the reachability tree to generate, and the maximum of required narrowing solutions. Note that these parameters, together with the third parameter, allow the user to request different sets of solutions as desired.

```
eq getNarrowingSolutions(M, T1, SearchQid, T2, TList, Q1, 0, MaxSol) = noSolution .
eq getNarrowingSolutions(M, T1, SearchQid, T2, TList, Q1, MaxDepth, 0)
                                                                = noSolution .
eq getNarrowingSolutions(M, T1, SearchQid, T2, TList, Q1, MaxDepth, MaxSol) =
      if SearchQid == '1 then
         (T1, '1, T2, MaxSol, 1, noSolution,
          getReachabilityTree(M, T1, empty, Q1, 1))
      else
         (T1, SearchQid, T2, MaxSol, 1, noSolution,
          getReachabilityTree(M, T1, empty, Q1, MaxDepth))
      fi [owise] .
```

Figure 40: Equations that define the initial behavior of the getNarrowingSolutions command in the GET-NARROWING-SOLUTIONS module.

The declaration of the operator that defines the *getNarrowingSolutions* command is shown in Figure 38, while the equations used to define its initial behavior are shown in Figure 40. Two equations are needed to inform the user that no solutions are found in case the value 0 has been specified for the maximum depth of the reachability tree or for the maximum number of solutions required. If both parameters take a value different from 0, another equation is used that considers two cases: if the *SearchQid* variable, which represents the search arrow, takes the value *'1*, the command will create a new *NarrowingState* in which we it only needs to generate the first level of the tree, and in any other case, that is, if *SearchQid* takes the values *'+* or *'**, it will also create a new *NarrowingState*, but this time it needs to generate the tree depending on the maximum depth and maximum solutions indicated, which will be managed later. In both cases, the counter used to identify solutions is initialized to 1.

In the third case, the *getNarrowingSolutions* command calls the *getReachabilityTree* command, which we discussed in detail in the previous section. This command works exactly the same way as before, and the only difference is that the *State* data structure that it uses internally will contain a new *Step* parameter that can take three different constant values: *genNode*, *unify* or *genSolutions* (Its definition is shown in Figure 38). The value that this parameter takes will determine the narrowing step to be carried out, that is, this parameter is in charge of directing the execution, since now, every time we generate a new node, we have to try to unify with the target term, and if it is possible, we need to calculate the solutions before generating more nodes, to stop generating them when the maximum number of solutions indicated by the user is reached.

4.4. Unification with the target term

Each time we generate a new node, that is, a new term by rewriting, in the reachability tree, it is necessary to try to unify that new term obtained with the specified target term. In case the unification is possible, for each of the unifiers found we will have a solution to the problem. In our case, we must always consider that if the user has indicated a maximum number of solutions, it cannot be exceeded, so it will not make sense to try the unification once we reach that state, the resolution of which we will see later. It will also be necessary to distinguish the case in which the search arrow is =>+ and the case in which it is another of the two possible ones.

In the first case, we need an equation that evaluates whether we are in the first level of the reachability tree or at a deeper level, since when the search arrow is =>+, the first level does not interest us, and it will not be necessary to try to unify with the nodes belonging to that level. If the level we are at is deeper, we must try to unify the target term with the current node term. To do this, a call to the *iterativeMetaVariantUnify* function is used, which is detailed later in this section. This is specified by the equation defined in Figure 41.

```
eq  (T1, '+, T2, MaxSol, N11, Q7, NRS1, (M, T3, N1, NARS1 ;
    {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
    NARS2, MaxDepth, N9, genSolutions))
       = if N5 > 1 then
            (T1, '+, T2, MaxSol, N11, Q7, NRS1 ;
            iterativeMetaVariantUnify(M, T2 =? T4, empty, Q7, 0),
            (M, T3, N1, NARS1 ; {N2, {T4, Ty1, C1, Q1, S1, S2, Q2},
            N3, N4, Q3, N5}, NARS2, MaxDepth, N9, genSolutions))
         else
            (T1, '+, T2, MaxSol, N11, Q7, NRS1, (M, T3, N1, NARS1 ;
            {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
            NARS2, MaxDepth, N9, genNode))
          fi [owise] .
```

Figure 41: Start of the unification step when the search arrow is =>+ in the GET-NARROWING-SOLUTIONS module.

In case the search arrow takes the value =>* or =>*1*, the behaviour will be the same. If it is =>*1*, it is not necessary to check that the level we are in is deeper than the first, since the maximum depth limit will have been automatically set to one, and the lower levels cannot be generated. In these two cases, therefore, we simply must try to do the unification directly, also calling the *iterativeMetaVariantUnify* function, and without having to check the value of the depth level, as shown in Figure 42.

```
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1, (M, T3, N1, NARS1 ;
    {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
    NARS2, MaxDepth, N9, unify))
       = (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ;
         iterativeMetaVariantUnify(M, T2 =? T4, empty, Q7, 0),
         (M, T3, N1, NARS1 ; {N2, {T4, Ty1, C1, Q1, S1, S2, Q2},
         N3, N4, Q3, N5}, NARS2, MaxDepth, N9, genSolutions)) [owise] .
```

Figure 42: Start of the unification step when the search arrow is => or =>1 in the GET-NARROWING-SOLUTIONS module.*

The definition of the *iterativeMetaVariantUnify* function is shown in Figure 43, where four equations are specified. Calls are made to a predefined function in the *META-LEVEL* module called *metaVariantUnify* to find unifiers. Every time a new unifier is found, a recursive call to the *iterativeMetaVariantUnify* function itself is made, in order to continue finding even more unifiers. When no more unifiers are found, the *metaVariantUnify* function will return the constant value *noUnifier*. In that case, we will stop looking for unifiers for the current node, and we can proceed to the step that we will see in the next section.

```
eq iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, 0)
      = metaVariantUnify(M, T1 =? T2, TList, Q7, 0) ;
        iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, 1) .
eq {S1, Q8} ; iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, N2:NzNat)
      = {S1, Q8} ; metaVariantUnify(M, T1 =? T2, TList, Q7, N2:NzNat) ;
        iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, N2:NzNat + 1) .
eq NRS1 ; noUnifier ; iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, 1)
      = NRS1 ; noUnifier .
eq NRS1 ; noUnifier ; iterativeMetaVariantUnify(M, T1 =? T2, TList, Q7, N2:NzNat)
      = NRS1 [owise] .
```

Figure 43: Definition of the iterativeMetaVariantUnify function in the GET-NARROWING-SOLUTIONS module.

The arguments expected by the *iterativeMetaVariantUnify* function are the same as those taken by the *metaVariantUnify* function:

- The module in which the equations and rules to perform the unification are defined.
- An expression of the form “T1 =? T2” in which “T1” and “T2” are the two terms that we are trying to unify.
- A list of terms that is not used now, but that will be very important later.
- A *Qid* that indicates the identifier of variables to avoid when naming new fresh variables. Otherwise, there could be clashes with the variables of the initial and target terms.
- A natural number that allows iterating over the possible unifiers, so that if it takes the value “0” we will be asking for the first unifier, if it takes the value “1” for the second, etc.

4.5. Calculation of solutions with the unifiers

The *metaVariantUnify* returns either the constant *noUnifier* or a result of type *UnificationPair* following the form {*S*, *Q*}, where “S” is the computed substitution to solve the problem and “Q” is the Qid used to name the generated fresh variables.

```
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ; noUnifier,
    (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genSolutions))
      = (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1,
        (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genNode)) .
```

Figure 44: Equation that ends the unification step in the GET-NARROWING-SOLUTIONS module.

If we find the constant *noUnifier*, it means that no unifiers could be found, so we can directly end the unification step in this node, going back to the initial step (node generation). This process is carried out by the equation shown in Figure 44. If unifiers have been found, it means that the term of the current node represents one or more solutions of the proposed narrowing problem (depending on whether one or more unifiers have been found).

```
eq  (T1, SearchQid, T2, unbounded, N11, Q7, NRS1 ; {S5, Q8} ; NRS2,
     (M, T3, N1, NARS1 ;
     {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
     NARS2, MaxDepth, N9, genSolutions))
        = (T1, SearchQid, T2, unbounded, N11, Q7, NRS1 ;
          calculateCumulativeSub(N2, T4, S1, S5) ; NRS2,
          (M, T3, N1, NARS1 ;
          {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
          NARS2, MaxDepth, N9, genSolutions)) .
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ; {S5, Q8} ; NRS2,
    (M, T3, N1, NARS1 ;
    {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
    NARS2, MaxDepth, N9, genSolutions))
       = (T1, SearchQid, T2, sd(MaxSol, 1), N11, Q7, NRS1 ;
         calculateCumulativeSub(N2, T4, S1, S5) ; NRS2,
         (M, T3, N1, NARS1 ;
         {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4, Q3, N5},
         NARS2, MaxDepth, N9, genSolutions)) [owise] .
```

Figure 45: Equations that begin the computation of the solutions associated with each unifier found in the GET-NARROWING-SOLUTIONS module.

The equations shown in Figure 45 begin the generation of solutions, making a call to a defined function called *calculateCumulativeSub* with each of the unifiers found. One of those equations considers the case in which a maximum number of solutions has been defined, so each time the call to the function is made to calculate a new solution, it will subtract one from that parameter. The function receives the identifier of the node in which the unifier has been found, the term that characterizes that node, a substitution that will be processed to obtain the solution computed along the branch of the node (used to arrive at the term of the node), and another substitution that is the one used in the objective term to achieve unifying. The solution term will always be that of the current node, and the substitution used in the target term to unify will always be the one indicated by the *UnificationPair* obtained. The substitution used in the solution term must be calculated differently depending on the level of the reachability tree in which we are.

The simplest case occurs when the unifier has been found in a node of the first level, in which case it is not necessary to compute the first substitution along the tree branch. The computed substitution will be directly the substitution that we had saved in the node, which indicates precisely the substitution that has been used in the initial term to reach the term of the current node. Simply create the data structure that represents the *NarrowingResultStructure* solution with the data we already have, assigning a new identifier to said solution. For this, the solution counter that we keep in the state is used. All this is defined with the equation shown in Figure 46.

```
eq (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ;
   calculateCumulativeSub(N2, T6, S1, S6) ;
   NRS2, (M, T3, N1, NARS1 ;
   {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, N5} ;
   NARS2, NARS3, MaxDepth, N9, genSolutions))
      = (T1, SearchQid, T2, MaxSol, N11 + 1, Q7, NRS1 ;
        {N11, T6, reduceSub(M, S1), reduceSub(M, S6)} ;
        NRS2, (M, T3, N1, NARS1 ;
        {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, N5} ;
        NARS2, NARS3, MaxDepth, N9, genSolutions)) .
```

Figure 46: Equation of the GET-NARROWING-SOLUTIONS module that computes the narrowing solutions when we are at the first level of the reachability tree.

If we are in a deeper node in the tree, it is necessary to go backwards through the branch in which we are to combine the substitutions used in each step to generate the nodes, thus achieving a final substitution computed when reaching the initial node. For this, the equations defined in Figure 47 are used.

```
eq (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ;
   calculateCumulativeSub(N6, T6, S5, S6) ;
   NRS2, (M, T3, N1, NARS1 ;
   {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, N5} ; NARS2 ;
   {N6, {T5, Ty2, C2, Q4, S3, S4, Q5}, N7, N2, Q6, N4} ;
   NARS3, NARS4, MaxDepth, N9, genSolutions))
      = (T1, SearchQid, T2, MaxSol, N11 + 1, Q7, NRS1 ;
        {N11, T6, reduceSub(M, applySub(S1, S5)), reduceSub(M, S6)} ;
        NRS2, (M, T3, N1, NARS1 ;
        {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, N5} ; NARS2 ;
        {N6, {T5, Ty2, C2, Q4, S3, S4, Q5}, N7, N2, Q6, N4} ;
        NARS3, NARS4, MaxDepth, N9, genSolutions)) .
eq (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ;
    calculateCumulativeSub(N6, T6, S5, S6) ;
    NRS2, (M, T3, N1, NARS1 ;
    {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4:NzNat, Q3, N5} ; NARS2 ;
    {N6, {T5, Ty2, C2, Q4, S3, S4, Q5}, N7, N2, Q6, N8} ;
    NARS3, NARS4, MaxDepth, N9, genSolutions))
       = (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1 ;
         calculateCumulativeSub(N2, T6, applySub(S1, S5), S6) ;
         NRS2, (M, T3, N1, NARS1 ;
         {N2, {T4, Ty1, C1, Q1, S1, S2, Q2}, N3, N4:NzNat, Q3, N5} ;
         NARS2 ; {N6, {T5, Ty2, C2, Q4, S3, S4, Q5}, N7, N2, Q6, N8} ;
         NARS3, NARS4, MaxDepth, N9, genSolutions)) .
```

Figure 47: Equations of the GET-NARROWING-SOLUTIONS module that compute the narrowing solutions for levels below the first level of the reachability tree.

The second equation makes recursive calls to the function *calculateCumulativeSub* going backwards through the predecessors of the node in the branch, while the substitutions are being combined, until reaching the node where the predecessor is at level 1, that is, until we are at level 2 of the tree of reachability. At that moment, the last combination of substitutions is made, coming to compute the final substitution, which will be the one returned in the solution created.

We must bear in mind that, when considering each of the unifiers as a solution, the maximum number of solutions could be reached during this step, so it is necessary to define an equation capable of detecting this situation to eliminate the unifiers that remain queued by process. This equation is defined with the specification shown in Figure 48.

```
eq  (T1, SearchQid, T2, 0, N11, Q7, NRS1 ; {S5, Q8} ; NRS2,
    (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genSolutions))
        = (T1, SearchQid, T2, 0, N11, Q7, NRS1 ; NRS2,
          (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genSolutions)) .
```

Figure 48: Equation that removes unifiers from the queue if the maximum number of solutions has already been reached in the GET-NARROWING-SOLUTIONS module.

In some of the previous equations, specifically those that complete the substitution computation step, a call is made to the *reduceSub* function. This function normalizes the computed substitutions, using the equations defined in the module indicated by the user. Its specification can be found in Figure 49.

```
eq reduceSub(M, (none).Substitution) = (none).Substitution .
eq reduceSub(M, V1 <- T1 ; S1)
      = V1 <- getTerm(metaReduce(M, T1)) ; reduceSub(M, S1) .
```

Figure 49: Specification of the reduceSub function in the GET-NARROWING-SOLUTIONS module.

When all the solutions for the current node have been calculated, that is, when all the unifiers of the current term with the target term have been computed, the unification step is completed, using the equation shown in Figure 50.

```
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1,
    (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genSolutions))
       = (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1,
         (M, T3, N1, NARS1, NARS2, MaxDepth, N9, genNode)) [owise] .
```

Figure 50: Function that ends the unify step when the solutions for a node have been computed in the GET-NARROWING-SOLUTIONS module.

4.6. Iteration and termination of the algorithm

It should be noted that the algorithm implemented to perform the narrowing behaves iteratively. As we can see in the equations that we have defined, in all those in which the *NarrowingState* structure is used in a complete way, a parameter of type *Step* appears at the end of said structure, which marks the step of the algorithm in which we are, starting at *genNode* (generation of a new node), to later go to *unify* (search for unifiers of the term generated with the target term) and finally reaching *genSolutions* (creation of solutions from the unifiers found). Then, the parameter will return to the value *genNode*, starting this process again, until one of the following situations is reached:

a) The maximum depth specified by the user is reached. In that case, the reachability tree state data structure will be converted to just an ordered list of nodes.
b) The maximum solutions indicated by the user are reached. In that case, we will directly return the solutions computed so far, without continuing to generate nodes or compute solutions if we were in the unification step.
c) The entire tree is generated. This case can only occur if the reachability tree is finite. In that case, the *NarrowingState* structure will remain the same as in the first case.

```
eq  (T1, SearchQid, T2, 0, N11, Q7, NRS1,
    (M, T3, N1, NARS1, NARS2, MaxDepth, N9, unify))
      = (T1, SearchQid, T2, 0, N11, Q7, NRS1, NARS1) .
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, NRS1, NARS1)
      = NRS1 .
eq  (T1, SearchQid, T2, MaxSol, N11, Q7, noSolution, NARS1)
      = noSolution .
```

Figure 51: Equations that define the termination of the narrowing algorithm in the GET-NARROWING-SOLUTIONS module.

Figure 51 shows the equations that have been defined to manage the three cases raised, in addition to an equation that returns the constant *noSolution* if no solution has been found to the narrowing problem posed at the end of the algorithm in any of these ways.

4.7. Testing the narrowing algorithm implementation

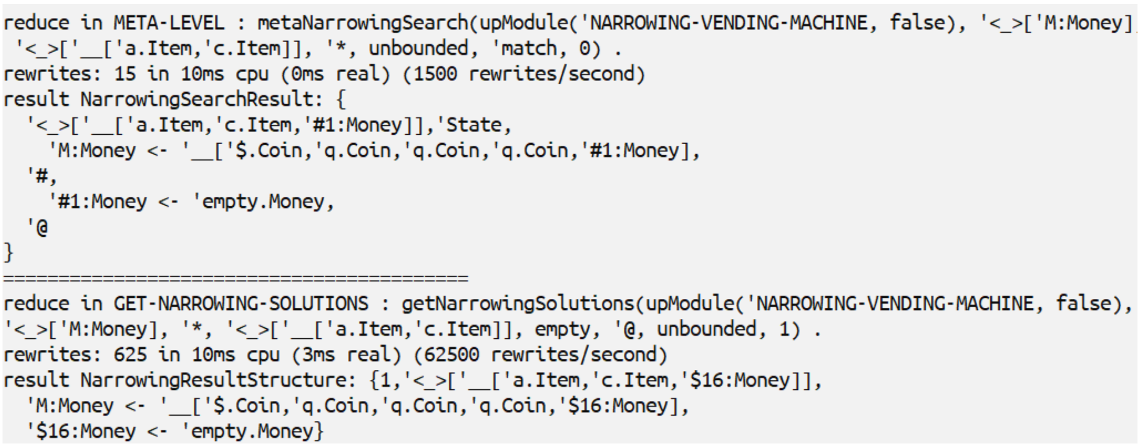
Once we have defined the narrowing algorithm in Maude, we can test it using different modules, terms and bounds as arguments when calling the command that invokes it. To begin with, we can use the modules previously discussed in Section 3.5. Their code level specification can be found in Figure 13 and Figure 34.

If we take the narrowing problem < *M:Money* > =>* < *a c* >, we can compare the result obtained when executing it using two different commands. On the one hand, the narrowing command already defined in the *META-LEVEL* module, and on the other hand, the new command we have defined. To do this, we use them as follows:

```
reduce in META-LEVEL :
metaNarrowingSearch(
        upModule('NARROWING-VENDING-MACHINE, false),
        '<_>['M:Money],
        '<_>['__['a.Item,'c.Item]],
        '*,
        unbounded, 'match, 0) .
```

```
reduce in GET-NARROWING-SOLUTIONS :
getNarrowingSolutions(
        upModule('NARROWING-VENDING-MACHINE, false),
        '<_>['M:Money],
        '*,
        '<_>['__['a.Item,'c.Item]],
        empty, '@, unbounded, 1) .
```

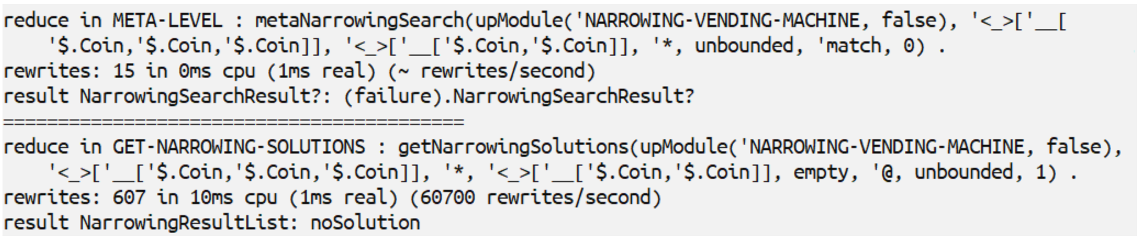
In Figure 52 we show the result obtained, which is very similar in both cases. The solution obtained is the same, although the data structure returned is slightly different, as well as the way of naming the variables. It can be seen how the command already defined in *META-LEVEL* is faster, since its implementation is written directly on C++, the language underlying Maude. This is something that we already expected, but the objective of this work is not to define a narrowing command faster than the existing one, but to define a command that allows a variation of parameters (maximum solutions instead of solution number), and that can be modified to implement other types of narrowing, as we will see later. In this specific example, the difference is minimal, since it is a relatively simple problem, but we will see that by complicating the problem, the difference in execution time will become more noticeable.



```
reduce in META-LEVEL : metaNarrowingSearch(upModule('NARROWING-VENDING-MACHINE, false), '<_>['M:Money]
 '<_>['__['a.Item,'c.Item]], '*, unbounded, 'match, 0) .
rewrites: 15 in 10ms cpu (0ms real) (1500 rewrites/second)
result NarrowingSearchResult: {
  '<_>['__['a.Item,'c.Item,'#1:Money]],'State,
    'M:Money <- '__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'#1:Money],
  '#,
    '#1:Money <- 'empty.Money,
  '@
}
==========================================
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('NARROWING-VENDING-MACHINE, false),
'<_>['M:Money], '*, '<_>['__['a.Item,'c.Item]], empty, '@, unbounded, 1) .
rewrites: 625 in 10ms cpu (3ms real) (62500 rewrites/second)
result NarrowingResultStructure: {1,'<_>['__['a.Item,'c.Item,'$16:Money]],
  'M:Money <- '__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'$16:Money],
  '$16:Money <- 'empty.Money}
```

Figure 52: Example 1 comparing narrowing commands using the NARROWING-VENDING-MACHINE module.

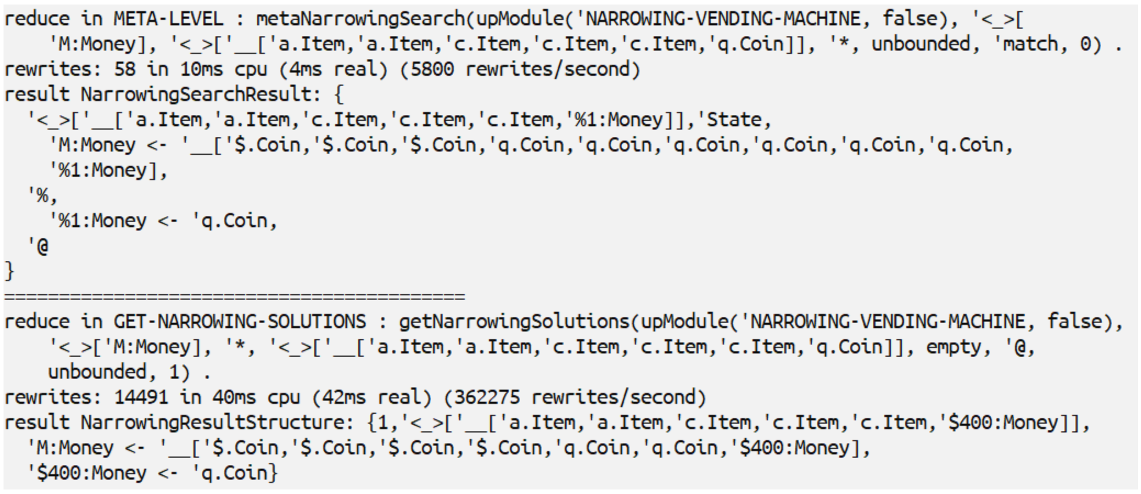
Using the commands in a similar way to what we have explained, we can define other narrowing problems to solve on the same module, and we will obtain similar results, ensuring that our narrowing algorithm fulfills its function correctly. We show some more examples in Figure 53 and Figure 54.



```
reduce in META-LEVEL : metaNarrowingSearch(upModule('NARROWING-VENDING-MACHINE, false), '<_>['__[
    '$.Coin,'$.Coin,'$.Coin]], '<_>['__['$.Coin,'$.Coin]], '*, unbounded, 'match, 0) .
rewrites: 15 in 0ms cpu (1ms real) (~ rewrites/second)
result NarrowingSearchResult?: (failure).NarrowingSearchResult?
==========================================
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('NARROWING-VENDING-MACHINE, false),
    '<_>['__['$.Coin,'$.Coin,'$.Coin]], '*, '<_>['__['$.Coin,'$.Coin]], empty, '@, unbounded, 1) .
rewrites: 607 in 10ms cpu (1ms real) (60700 rewrites/second)
result NarrowingResultList: noSolution
```

Figure 53: Example 2 comparing narrowing commands using the NARROWING-VENDING-MACHINE module.

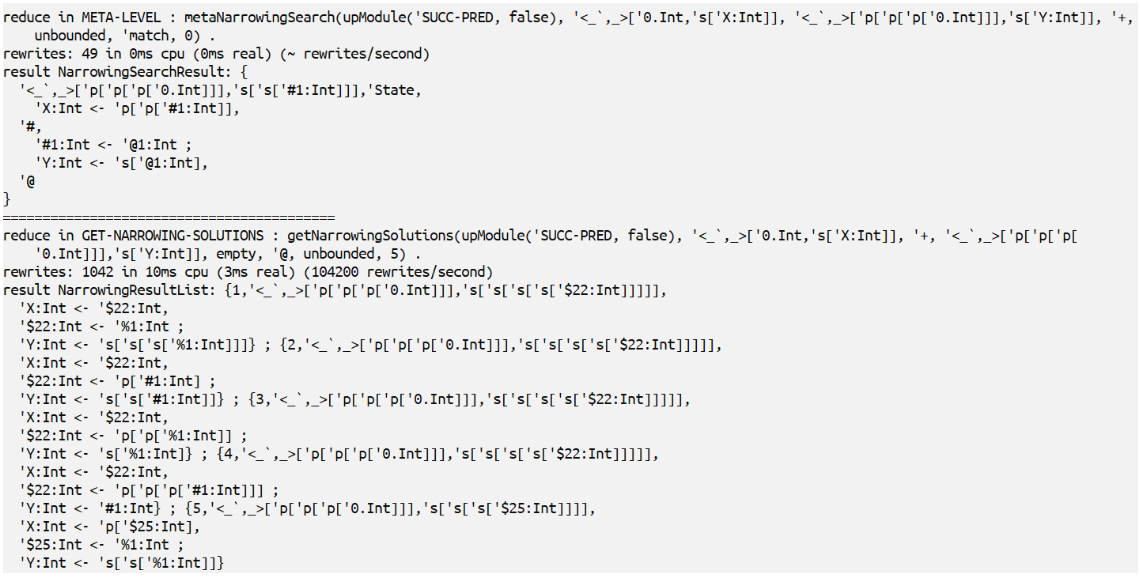
In the second one, we can appreciate a very important difference between the command already defined in Maude and the new command that we have defined. While the first one returns the solutions without reducing the substitutions to their normal form, we have taken this into account in our algorithm, always returning them in their normal form, which makes it easier for the user to read and understand.



```
reduce in META-LEVEL : metaNarrowingSearch(upModule('NARROWING-VENDING-MACHINE, false), '<_>[
    'M:Money], '<_>['__['a.Item,'a.Item,'c.Item,'c.Item,'q.Coin]], '*, unbounded, 'match, 0) .
rewrites: 58 in 10ms cpu (4ms real) (5800 rewrites/second)
result NarrowingSearchResult: {
  '<_>['__['a.Item,'a.Item,'c.Item,'c.Item,'c.Item,'%1:Money]],'State,
    'M:Money <- '__['$.Coin,'$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin,'q.Coin,'q.Coin,'q.Coin,
    '%1:Money],
  '%,
    '%1:Money <- 'q.Coin,
  '@
}
==========================================
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('NARROWING-VENDING-MACHINE, false),
    '<_>['M:Money], '*, '<_>['__['a.Item,'a.Item,'c.Item,'c.Item,'c.Item,'q.Coin]], empty, '@,
    unbounded, 1) .
rewrites: 14491 in 40ms cpu (42ms real) (362275 rewrites/second)
result NarrowingResultStructure: {1,'<_>['__['a.Item,'a.Item,'c.Item,'c.Item,'c.Item,'$400:Money]],
  'M:Money <- '__['$.Coin,'$.Coin,'$.Coin,'$.Coin,'q.Coin,'q.Coin,'$400:Money],
  '$400:Money <- 'q.Coin}
```

Figure 54: Example 3 comparing narrowing commands using the NARROWING-VENDING-MACHINE module.

Let's now consider another module that we already know, the *SUCC-PRED* module (see Figure 34). In it we can see one of the great differences between the existing command and the new defined command: while in the one already defined it is necessary to carry out executions one by one to obtain the different solutions, the new command allows specifying a maximum number of solutions, obtaining as many as we want (if they exist) at once.

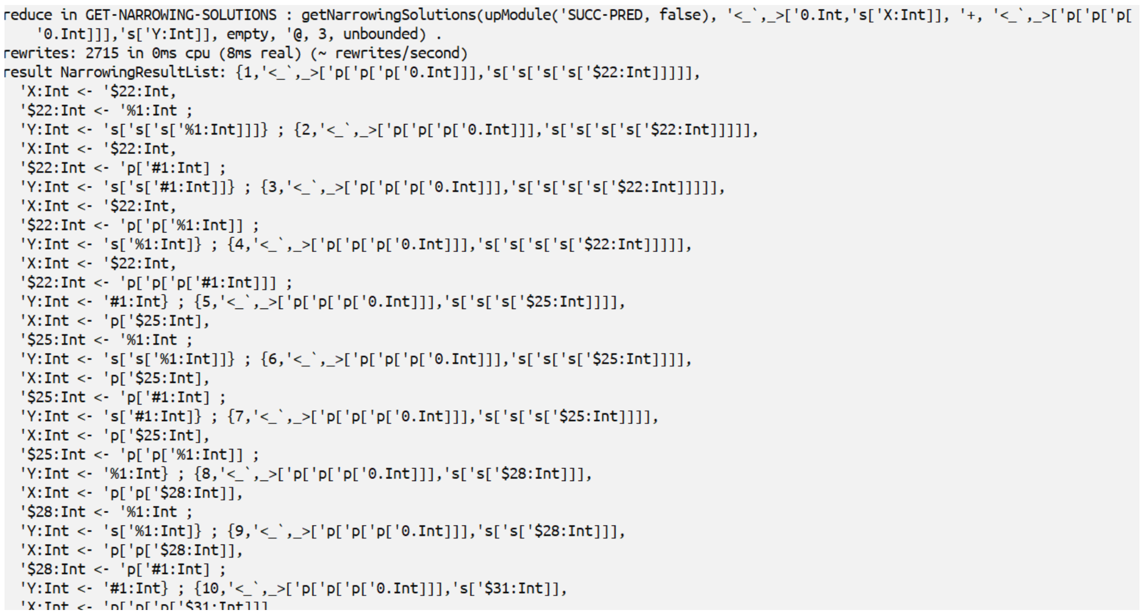


```
reduce in META-LEVEL : metaNarrowingSearch(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '<_`,_>['p['p['p['0.Int]]],'s['Y:Int]], '+,
    unbounded, 'match, 0) .
rewrites: 49 in 0ms cpu (0ms real) (~ rewrites/second)
result NarrowingSearchResult: {
  '<_`,_>['p['p['p['0.Int]]],'s['s['#1:Int]]],'State,
    'X:Int <- 'p['p['#1:Int]],
  '#,
    '#1:Int <- '@1:Int ;
    'Y:Int <- 's['@1:Int],
  '@
}
==========================================
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '+, '<_`,_>['p['p['p[
    '0.Int]]],'s['Y:Int]], empty, '@, unbounded, 5) .
rewrites: 1042 in 10ms cpu (3ms real) (104200 rewrites/second)
result NarrowingResultList: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]} ; {2,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['#1:Int] ;
  'Y:Int <- 's['s['#1:Int]]} ; {3,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['p['%1:Int]] ;
  'Y:Int <- 's['%1:Int]} ; {4,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['p['p['#1:Int]]] ;
  'Y:Int <- '#1:Int} ; {5,'<_`,_>['p['p['p['0.Int]]],'s['s['s['$25:Int]]]],
  'X:Int <- 'p['$25:Int],
  '$25:Int <- '%1:Int ;
  'Y:Int <- 's['s['%1:Int]]}
```

Figure 55: Example of a comparison between narrowing commands using the SUCC-PRED module.

In Figure 55 we show the invocation of the commands considering the narrowing problem *< 0, X:Int > =>+ < p(p(p(0))), s(Y:Int) >* , but using the argument of the new command defined to return several solutions at the same time, instead of just one.

We could also use the other argument of the new defined command (maximum depth). Now, instead of asking for a specific number of solutions, we can ask, for example, for the solutions that can be found by expanding to level 3 of the tree. In this case, using the same narrowing problem and the same module, 28 solutions are obtained. Some of them can be seen in Figure 56.



```
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '+, '<_`,_>['p['p['p[
    '0.Int]]],'s['Y:Int]], empty, '@, 3, unbounded) .
rewrites: 2715 in 0ms cpu (8ms real) (~ rewrites/second)
result NarrowingResultList: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]} ; {2,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['#1:Int] ;
  'Y:Int <- 's['s['#1:Int]]} ; {3,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['p['%1:Int]] ;
  'Y:Int <- 's['%1:Int]} ; {4,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['p['p['#1:Int]]] ;
  'Y:Int <- '#1:Int} ; {5,'<_`,_>['p['p['p['0.Int]]],'s['s['s['$25:Int]]]],
  'X:Int <- 'p['$25:Int],
  '$25:Int <- '%1:Int ;
  'Y:Int <- 's['s['%1:Int]]} ; {6,'<_`,_>['p['p['p['0.Int]]],'s['s['s['$25:Int]]]],
  'X:Int <- 'p['$25:Int],
  '$25:Int <- 'p['#1:Int] ;
  'Y:Int <- 's['#1:Int]} ; {7,'<_`,_>['p['p['p['0.Int]]],'s['s['s['$25:Int]]]],
  'X:Int <- 'p['$25:Int],
  '$25:Int <- 'p['p['%1:Int]] ;
  'Y:Int <- '%1:Int} ; {8,'<_`,_>['p['p['p['0.Int]]],'s['s['$28:Int]]],
  'X:Int <- 'p['p['$28:Int]],
  '$28:Int <- '%1:Int ;
  'Y:Int <- 's['%1:Int]} ; {9,'<_`,_>['p['p['p['0.Int]]],'s['s['$28:Int]]],
  'X:Int <- 'p['p['$28:Int]],
  '$28:Int <- 'p['#1:Int] ;
  'Y:Int <- '#1:Int} ; {10,'<_`,_>['p['p['p['0.Int]]],'s['$31:Int]],
  'X:Int <- 'p['p['p['$31:Int]]]
```

Figure 56: Example of using the maximum depth parameter in the getNarrowingSolutions command, using the SUCC-PRED module.

The new narrowing command defined is, therefore, slower than the one already defined in Maude's *META-LEVEL*, but it brings other types of possibilities to the user, combining the search arrow parameter with the maximum solutions and the maximum depth parameters. For example, with this new command, we could ask for the first 20 solutions to a narrowing problem that are found in levels 2, 3 and 4 of the reachability tree, ignoring the possible solutions in level 1 and levels deeper than 4. To do this, simply use =>+ as the search arrow, indicating a maximum of 20 solutions and the maximum depth of 4. Furthermore, as we have previously commented, the definition using the Maude language itself of a narrowing algorithm like this one, will allow us to make modifications to implement other types of narrowing not included in Maude native commands. One of those types of narrowing that can be implemented is *canonical narrowing modulo an equational theory (∑, E ∪ B)*, which we will see next.

5. Canonical narrowing

In this chapter we introduce and implement *canonical narrowing modulo an equational theory ($\sum$, $E \cup B$),* which solves some of the limitations of standard narrowing, seen earlier. In Section 5.1, we explain some of the limitations that we can find in the standard narrowing, presenting a motivation to look for an alternative. In Section 5.2, we introduce canonical narrowing modulo an equational theory, which is the one chosen as an alternative to standard narrowing to solve the limitations seen. In Section 5.3, we show the implementation process followed to code the canonical narrowing modulo an equational theory algorithm in Maude, taking as a starting point the code seen above. Finally, in Section 5.4, we present a comparison between the new canonical narrowing modulo an equational theory command with the standard narrowing command included in Maude and the standard narrowing command that we have defined ourselves.

To carry out this part of the work, the article *Canonical Narrowing with Irreducibility Constraints as a Symbolic Protocol Analysis Method* [7], by Santiago Escobar (Cotutor of this master thesis) and José Meseguer has been used as a reference.

5.1. Limitations of standard narrowing

Standard narrowing modulo an equational theory ($\sum$, $E \cup B$) is a powerful and efficient tool for solving many problems and identifying, for example, vulnerabilities in protocols, but in some cases its efficiency is not the best.

Let us consider that the standard narrowing computes a two-step sequence with its two corresponding substitutions, as follows:

$$(t_1 \downarrow_{E,B}) \leadsto^{\sigma_1}_{l_1 \to r_1} (t_2 \downarrow_{E,B}) \leadsto^{\sigma_2}_{l_2 \to r_2} t_3$$

where there is first a narrowing step in which the normalized initial term t_1 is rewritten to the t_2 term using the rule $l_1 \to r_1$ and the substitution σ_1, to later normalized the term t_2 and rewrite it to t_3 using the rule $l_2 \to r_2$ and the substitution σ_2.

We can see that although the terms are normalized before performing each narrowing step, the substitutions σ_1 and σ_2 are computed and used without reducing them to their normal form, being the final computed substitution the composition of both of them: $\sigma_1\sigma_2$. Under normal conditions, this is not a problem, but if we have the case where the composition is reducible, that is, $\sigma_1\sigma_2 \neq_B (\sigma_1\sigma_2 \downarrow_{E,B})$, then the computed narrowing trace will not have actual representation with a rewrite trace on the system used, since in the rewriting traces the substitutions are normalized in each of the steps.

Therefore, on some occasions, several traces computed by the standard narrowing algorithm could be ignored, and by not doing so there is a consumption of time and resources greater than it could be, and depending on the problem, the growth of this consumption is exponential, since from traces that could be ignored, more and more traces will be generated, creating a narrowing tree in which most of the branches are useless for a realistic analysis. In addition to this temporary and resource inefficiency, once the solutions to the problem that we specify have been obtained, we need to filter by hand the solutions that are really of interest to us, since, as we have already commented, many of them will not have a real correspondence with rewrite sequences. This situation motivates us to look for a narrowing that can find those narrowing steps that do not have the real correspondence with rewriting steps in the rewrite system specified each time, ignoring, on the one hand, the corresponding step, and on the other, all the narrowing steps that could be taken later from that one. Consequently, a much smaller narrowing tree is generated, saving a large part of the time and resources we were talking about. The narrowing capable of doing all of this is canonical narrowing.

5.2. Introduction to canonical narrowing

As we have seen in the previous section, canonical narrowing relation is a narrowing capable of detecting those narrowing steps that do not have a real correspondence with rewriting steps in the rewrite system used. To do this, it relies on conditions that we refer to as irreducibility constraints: when a new term is reached through the generation in the narrowing tree, an irreducibility constraint associated with it is calculated and saved to be used later along the entire narrowing sequence from that new term.

If we position ourselves on a generated term, the irreducibility constraint associated with it is calculated with the normalized left-hand side of the rule used in the transition that allows reaching that term from the previous one. If we do this for each of the generated terms, we will build a list of irreducibility restrictions that must be fulfilled all the time in the sequence in which we find ourselves. This allows us to discard sequences that do not meet them, giving rise to a narrowing tree which, on multiple occasions, will be much smaller than the one that could be generated without those restrictions, which translates into greater computational efficiency when carrying it out in practice.

For example, we can consider a new narrowing sequence starting from the one specified as an example in the previous section, as follows:

$$(t_1 \downarrow_{E,B}) \rightsquigarrow^{\sigma'_1}_{l'_1 \to r'_1} (t'_2 \downarrow_{E,B}) \rightsquigarrow^{\sigma'_2}_{l'_2 \to r'_2} t'_3$$

that uses irreducibility constraints, so that if any of the two narrowing steps do not meet any of the restrictions generated, the sequence would stop being generated, thus avoiding the previous situation, always obtaining that $\sigma'_1\sigma'_2 =_B (\sigma'_1\sigma'_2 \downarrow_{E,B})$. Specifically, when performing the first step in the narrowing sequence, the irreducibility constraint *irr(*$l'_1\sigma'_1 \downarrow_{E,B}$*)* would be generated, creating a list of irreducibility conditions to which the constraint *irr(*$l'_2\sigma'_2 \downarrow_{E,B}$*)*, which is calculated using the first irreducibility constraint, would be added when performing the second step. Thanks to the completeness of the narrowing, and considering the substitutions that were computed in the previous section, we know that $\sigma'_1\sigma'_2 < (\sigma_1\sigma_2 \downarrow_{E,B})$, that is, the composition of the new computed substitutions is more general than the composition of the computed substitutions without irreducibility restrictions, so using this method we can obtain an equivalent but more efficient narrowing tree starting from the same initial term and taking into account the same term objective.

We will be able to appreciate the differences between standard narrowing and canonical narrowing in Section 5.4, where some practical examples are shown.

5.3. Implementation in Maude

To implement the canonical narrowing in Maude, we can use as a starting point the code of the standard narrowing explained and shown throughout Chapter 4, and only some modifications will be necessary, since in the design of the previous code we had in mind that it would be necessary to implement this new algorithm. This is where the *TermList* parameter that we have been ignoring until now (always using the constant empty, regardless of the value the user indicated) comes into play.

To implement canonical narrowing in Maude, we need to have a list of irreducible terms at each node in the reachability tree, containing the irreducibility constraints computed so far on that branch. We have defined some auxiliary functions that will be useful to carry out this process. Specifically, the variables, operators and equations shown in Figure 57 have been defined.

```
var Rls : RuleSet .   var Rl : Rule .   vars RHS LHS : Term .
var Atts : AttrSet .

op getIrreducibilityTerm : Module Qid Substitution -> Term .
eq getIrreducibilityTerm(M, Q1, S1)
      = getTerm(metaReduce(M,applySub(getLhs(getRls(M), Q1),S1))) .

op getLhs : RuleSet Qid -> Term .
eq getLhs(Rls (rl LHS => RHS [Atts label(Q1)] .), Q1) = LHS .
```

Figure 57: Variables, operators and equations defined in the GET-CANONICAL-NARROWING-SOLUTIONS module to obtain the irreducibility terms.

If we look at the variables, one of the *RuleSet* type, one of the *Rule* type, two of the *Term* type (*RHS* and *LHS*, acronyms referring to the right-hand side and the left-hand side of a rule respectively), and one of the *AttrSet* type are necessary.

The *getLhs* operator defined is responsible for obtaining the left-hand side of a rule found in a received rule set, searching for said rule from an identifier (*Qid*) also indicated. The identifier corresponds to the label of the searched rule. To define the behavior of the operator, an equation has been defined that takes advantage of the commutative property of the received rule set, and of the rule's attribute set, to quickly find the rule whose label corresponds to the one indicated as the identifier. Once it finds the rule, it just returns its left part, ignoring everything else.

```
eq (M, T1, N1, NARS1,
   {N2, {T2, Ty1, C1, Q1, S1, S2, Q2}, N3, 0, Q3, 1, TList1} ; NARS2,
   MaxDepth, N9, genNode)
      = (M, T1, N1 + 2, NARS1 ;
        {N2,
           {applySub(T2,
               rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9)),
            Ty1, C1, Q1,
            applySub(S1,
               rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9)),
            applySub(S2,
               rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9)),
            '$},
         N3, 0, Q3, 1, TList1},
        {N1,
           metaNarrowingApply(M, T1, TList1, Q3, N3 + 1),
         N3 + 1, 0, Q3, 1, TList1} ; NARS2 ;
        {N1 + 1,
           metaNarrowingApply(M,
           applySub(T2,
              rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9)),
        (TList1, getIrreducibilityTerm(M, Q1,
                  applySub(S2,
                  rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9)))),
         Q2, 0), 0, N2, Q2, 2,
        (TList1, getIrreducibilityTerm(M, Q1,
                  applySub(S2,
                  rename((getVars(T2),getRangeVars(S1),getRangeVars(S2)),N9))))},
        MaxDepth,
        N9 + getVarsNumber((getVars(T2),getRangeVars(S1),getRangeVars(S2))),
        unify) .
```

Figure 58: Example of using the getIrreducibilityTerm function in an equation from the GET-CANONICAL-NARROWING-SOLUTIONS module.

Once we have the left-hand side of the rule of interest (which will be a term), we can normalize it to obtain the irreducibility term. To do this, a *getIrreducibilityTerm* operator has been defined with an associated equation that defines its behavior. The operator receives as parameters the module to be used to normalize, the identifier of the rule to be searched (which will be used in the call to the *getLhs* view function), and a substitution, the origin of which we will see later. A term is returned that corresponds to the desired irreducibility term. The equation associated with the operator uses several predefined functions in Maude's *META-LEVEL* module, such as the *getRls* function, which gets the set of rules from a module, the *metaReduce* function, which reduces a term to its normal form using the received module, and *getTerm*, to obtain the term from the structure of type *ResultPair* returned by *metaReduce*. In addition, a call is made to the *getLhs* function defined previously, to obtain the left part of the indicated rule and to be able to normalize it, and a call to the *applySub* function, also defined previously, applying the substitution received to the left part of the rule before to reduce it to normal form.

The calls to the *getIrreducibilityTerm* equation will be made from the equations that generate the new nodes of the reachability tree, adding the irreducibility terms obtained to the irreducibility terms lists on the fly. Figure 58 shows one of the modified equations in which a call to the new function is made. Specifically, it is the equation that generates the nodes in the first level of the reachability tree, but for the equations that define the generation of deeper nodes, the modification is similar. We can see how when making a new call to *metaNarrowingApply* to try to generate a new node, the list of irreducibility terms is passed as a parameter, to which the irreducibility term calculated with the parent node of the node to be generated is added on the fly, from the rule that was used to generate it. Furthermore, the irreducibility term is added to the list of irreducibility terms of the node, thus ensuring that when generating the possible child nodes, the irreducibility constraints computed so far in that branch are considered. Note that to do all this, it is necessary that all the rules of the module must be labeled.

5.4. Comparison with the standard narrowing commands

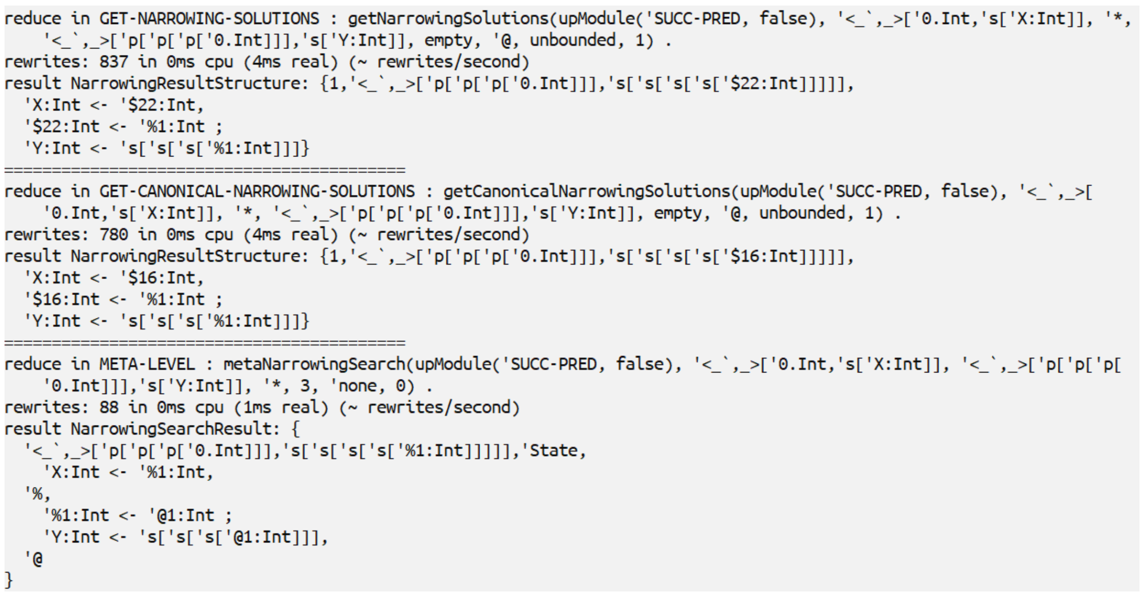
Once we have implemented both the *getNarrowingSolutions* command, which encodes the standard narrowing algorithm in Maude, and the *getCanonicalNarrowingSolutions* command, which encodes the canonical narrowing algorithm in Maude, we can proceed to the comparison of results using different practical examples. In addition, we can compare both commands with the *metaNarrowingSearch* command, which also performs the standard narrowing and is natively included in Maude, with its base encoded in C++.

```
op metaNarrowingSearch : Module Term Term Qid Bound
                         Qid Nat -> NarrowingSearchResult? [special ...] .
op getNarrowingSolutions : Module Term Qid Term TermList Qid
                           Bound Bound -> NarrowingResultList  .
op getCanonicalNarrowingSolutions : Module Term Qid Term TermList Qid
                                    Bound Bound -> NarrowingResultList  .
```

Figure 59: Comparison between the received parameters and the structure obtained from the new defined commands and the metaNarrowingSearch command.

The first significant differences that we can find are in the parameters received by the commands, as shown in Figure 59. While the *metaNarrowingSearch* command does not allow the use of a list of terms, the *getNarrowingSolutions* and *getCanonicalNarrowing-Solutions* commands do, although only the second command will use it to save and later use the irreducibility constraints. Note that by receiving the list as an argument from the beginning, we can manually indicate irreducibility constraints, which will be used when generating the narrowing sequences, and to which new ones that are calculated on the fly will be added. In addition, we can see that the predefined command in Maude receives at the end a parameter (natural number) that is used to choose a solution number, and that is the reason why the command returns a structure of type *NarrowingSearchResult?* which returns a single solution (if it exists). The commands that we have defined in this work do not use that parameter, since they have been prepared to return sets of solutions, attending to an extra parameter of type Bound that allow the user to handle the maximum number of solutions to calculate. The other parameter of type *Bound* is used to indicate the maximum depth to generate from the reachability tree, and it also appears in the *metaNarrowingSearch* command.

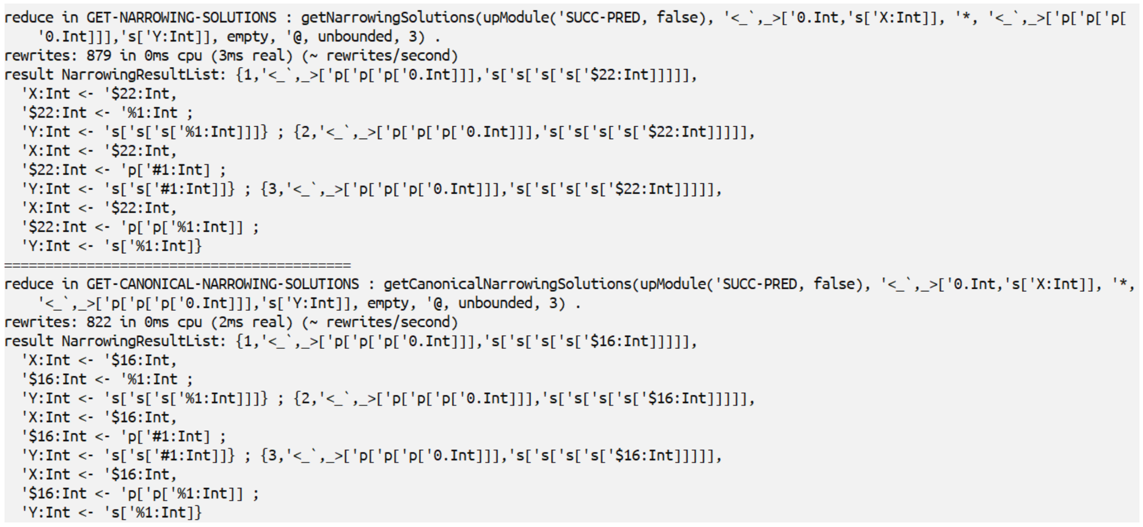
Let's consider the *SUCC-PRED* module defined in Figure 34, and the narrowing problem $< 0, \mathrm{s(X)} > \rightsquigarrow^{*}_{R,E\cup B} < \mathrm{p(p(p(0)))}, \mathrm{s(Y)} >$. We can search for a solution by invoking the *metaNarrowingSearch* command with the solution number 0, and also by invoking the *getNarrowingSolutions* and *getCanonicalNarrowingSolutions* commands indicating a maximum of one solution. In Figure 60 we can see the results obtained, noticing that when searching for a single solution, the Maude native command has better performance, followed by our canonical narrowing command, which can perform fewer rewrites than our standard narrowing command because it ignores sequences that are not useful.



```
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '*,
    '<_`,_>['p['p['p['0.Int]]],'s['Y:Int]], empty, '@, unbounded, 1) .
rewrites: 837 in 0ms cpu (4ms real) (~ rewrites/second)
result NarrowingResultStructure: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]}
==========================================
reduce in GET-CANONICAL-NARROWING-SOLUTIONS : getCanonicalNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>[
    '0.Int,'s['X:Int]], '*, '<_`,_>['p['p['p['0.Int]]],'s['Y:Int]], empty, '@, unbounded, 1) .
rewrites: 780 in 0ms cpu (4ms real) (~ rewrites/second)
result NarrowingResultStructure: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$16:Int]]]]],
  'X:Int <- '$16:Int,
  '$16:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]}
==========================================
reduce in META-LEVEL : metaNarrowingSearch(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '<_`,_>['p['p['p[
    '0.Int]]],'s['Y:Int]], '*, 3, 'none, 0) .
rewrites: 88 in 0ms cpu (1ms real) (~ rewrites/second)
result NarrowingSearchResult: {
  '<_`,_>['p['p['p['0.Int]]],'s['s['s['s['%1:Int]]]]],'State,
    'X:Int <- '%1:Int,
  '%,
    '%1:Int <- '@1:Int ;
    'Y:Int <- 's['s['s['@1:Int]]],
  '@
}
```

Figure 60: Comparison of commands looking for a solution for the mentioned narrowing problem by using the SUCC-PRED module.

As we have commented, our commands allow to search for more than one solution in a single execution. For example, if we ask them to return three solutions instead of one to the previous problem, they will achieve it without practically increasing the necessary rewrites and execution time, since they use the same reachability tree that was already being generated, as shown in the Figure 61. Using the *metaNarrowingSearch* command, it would be necessary to invoke it several times with different solution numbers.



```
reduce in GET-NARROWING-SOLUTIONS : getNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '*, '<_`,_>['p['p['p[
    '0.Int]]],'s['Y:Int]], empty, '@, unbounded, 3) .
rewrites: 879 in 0ms cpu (3ms real) (~ rewrites/second)
result NarrowingResultList: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]} ; {2,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['#1:Int] ;
  'Y:Int <- 's['s['#1:Int]]} ; {3,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$22:Int]]]]],
  'X:Int <- '$22:Int,
  '$22:Int <- 'p['p['%1:Int]] ;
  'Y:Int <- 's['%1:Int]}
==========================================
reduce in GET-CANONICAL-NARROWING-SOLUTIONS : getCanonicalNarrowingSolutions(upModule('SUCC-PRED, false), '<_`,_>['0.Int,'s['X:Int]], '*,
    '<_`,_>['p['p['p['0.Int]]],'s['Y:Int]], empty, '@, unbounded, 3) .
rewrites: 822 in 0ms cpu (2ms real) (~ rewrites/second)
result NarrowingResultList: {1,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$16:Int]]]]],
  'X:Int <- '$16:Int,
  '$16:Int <- '%1:Int ;
  'Y:Int <- 's['s['s['%1:Int]]]} ; {2,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$16:Int]]]]],
  'X:Int <- '$16:Int,
  '$16:Int <- 'p['#1:Int] ;
  'Y:Int <- 's['s['#1:Int]]} ; {3,'<_`,_>['p['p['p['0.Int]]],'s['s['s['s['$16:Int]]]]],
  'X:Int <- '$16:Int,
  '$16:Int <- 'p['p['%1:Int]] ;
  'Y:Int <- 's['%1:Int]}
```

Figure 61: Comparison of commands looking for three solutions for the mentioned narrowing problem by using the SUCC-PRED module.

To make comparisons of our commands with the *metaNarrowingSearch* command to find more than one solution, we can write a module that iteratively calls the command using different solution numbers according to an added parameter that specifies the desired number of solutions, as shown in Figure 62.

```
fmod META-NARROWING-SEARCH-SET is
  protecting META-LEVEL .

  sorts NarrowingSearchResultList NarrowingSearchResultListAux .
  subsort NarrowingSearchResult < NarrowingSearchResultList
          < NarrowingSearchResultListAux .

  op empty : -> NarrowingSearchResultList [ctor] .
  op metaNarrowingSearchSet : Module Term Term Qid Bound Qid Bound
                             -> NarrowingSearchResultList .
  op metaNarrowingSearchSet : Module Term Term Qid Bound Qid Bound Nat
                             -> NarrowingSearchResultListAux .
  op _;_ : NarrowingSearchResultList NarrowingSearchResultList
                             -> NarrowingSearchResultList [assoc id: empty] .

  var M : Module .  vars T1 T2 : Term .  vars Q1 Q2 : Qid .  var N : Nat .
  vars MaxDepth MaxSol : Bound .  var NSRL : NarrowingSearchResultList .
  var NSR : NarrowingSearchResult .

  eq metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, 0) = failure .
  eq metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol)
     = metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol, 0) [owise] .

  eq metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol, 0)
     = metaNarrowingSearch(M, T1, T2, Q1, MaxDepth, Q2, 0) ;
       metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol, 1) .
  eq metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, unbounded, 0)
     = metaNarrowingSearch(M, T1, T2, Q1, MaxDepth, Q2, 0) ;
       metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, unbounded, 1) .
  eq NSRL ; failure ;
     metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol, N) = NSRL .

  eq NSR ; metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, 0, N) = NSR .
  eq NSR ; metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, unbounded, N)
     = NSR ; metaNarrowingSearch(M, T1, T2, Q1, MaxDepth, Q2, N) ;
       metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, unbounded, N + 1) .
  eq NSR ; metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, s(MaxSol), N)
     = NSR ; metaNarrowingSearch(M, T1, T2, Q1, MaxDepth, Q2, N) ;
       metaNarrowingSearchSet(M, T1, T2, Q1, MaxDepth, Q2, MaxSol, N + 1) .
endfm
```

Figure 62: Module in Maude to make recursive calls to the metaNarrowingSearch command, allowing to specify the maximum number of required solutions.

Note that it has been necessary to define a new type to save the list of solutions obtained, as well as an auxiliary type to facilitate recursive calls. Additionally, a new subtype hierarchy had to be defined in which the least general type is the data structure returned by the *metaNarrowingSearch* command as a solution.

Once we have defined the module, we can compare our two commands with Maude's meta-level command using combinations of two parameters: the maximum depth of the reachability tree and the maximum number of solutions. For example, if we consider the same narrowing problem used so far in this section, and the same module (*SUCC-PRED*), performing several executions with different values for the mentioned parameters, we obtain the results shown in Figure 63.

Algorithm	Max. depth	Max. solutions	Time	Rewrites	Solutions
Standard Narrowing	*unbounded*	8	4 ms	1198	8
Standard Narrowing (Native command)	*unbounded*	8	2 ms	158	8
Canonical Narrowing	*unbounded*	8	3 ms	1182	8
Standard Narrowing	6	40	870 ms	208462	28
Standard Narrowing (Native command)	6	40	94 ms	7758	28
Canonical Narrowing	6	40	18 ms	4499	19
Standard Narrowing	9	unbounded	47630 ms	2511280	28
Standard Narrowing (Native command)	9	unbounded	4617 ms	304012	28
Canonical Narrowing	9	unbounded	39 ms	7146	19

Figure 63: Comparison of algorithms for the mentioned narrowing problem by using the SUCC-PRED module.

As expected, the command that defines the standard narrowing included natively in Maude is always faster than our command that encodes the same algorithm, since having its base encoded in C++, it needs to perform fewer rewriting steps. But our standard narrowing command was intended to be a base that would allow modifications to, for example, codify the canonical narrowing command, which does result in a significant improvement in execution time and rewriting steps, especially when we generate very large reachability trees. For example, in the last case presented in the previous figure, we can see how an execution that takes almost 5 seconds in the standard narrowing command in Maude becomes an execution of just around 40 milliseconds with the new canonical narrowing command, drastically reducing the number rewrite steps required. In addition, it manages to return fewer solutions, since as we have explained previously, it is capable of discarding those narrowing sequences that do not have correspondence with any real rewriting sequence.

Algorithm	Max. depth	Max. solutions	Time	Rewrites	Solutions
Standard Narrowing	*unbounded*	*unbounded*	11974 ms	239202	84
Standard Narrowing (Native command)	*unbounded*	*unbounded*	1272 ms	30562	84
Canonical Narrowing	*unbounded*	*unbounded*	1180 ms	27553	21

Figure 64: Comparison of algorithms for the mentioned narrowing problem by using the XOR-PROTOCOL module.

We can also consider the module that specifies the properties of the XOR encryption in Maude. Considering what is explained in the article used as a reference, we can use the following narrowing problem to find out if there is a sequence within the protocol that can reach the final state from the initial state (note that a reverse search is performed, just like in Maude-NPA):

$$< [+(pk(a, n(b, r1))), -(pk(b, Y)), +(Y * n(b, r1)) \mid nil]\ [-(pk(a, X)),+(pk(b, n(a, r2))), -(X * n(a, r2)) \mid nil] \mid inI(X * n(a, r2)), inI(pk(a, X)),inI(pk(b, Y)) >$$

$$\leadsto^{*}_{R,E\cup B}$$

$$< [nil \mid +(pk(a, n(b, r1))), -(pk(b, Y)), +(Y * n(b, r1))]\ [nil \mid -(pk(a, X)), +(pk(b, n(a, r2))), -(X * n(a, r2))] \mid nI(X * n(a, r2)), nI(pk(a, X)), nI(pk(b, Y)) >$$

The results obtained when executing the problem using the three commands without any kind of restriction in the boundaries (that is, looking for all possible solutions) are shown in Figure 64. On this occasion, once again and within expectations, our standard narrowing command is slower than the Maude's one, but the command with which we perform the canonical narrowing is still faster, although in this case the difference in execution time is not as noticeable as before. However, now we find a very notable difference: while the standard narrowing finds 84 solutions, the canonical narrowing finds only 21, which means that the standard narrowing considers 63 solutions whose sequence does not have correspondence with rewriting sequences, that is, 63 unreal sequences.

Finally, we can define a module that specifies the properties of the abelian groups and using a transition rule similar to the one shown in the *SUCC-PRED* module in Figure 34. This rewrite theory is much more complex when it comes to doing narrowing, since by containing a greater number of variants, the first level of the reachability tree is already very large in width, exponentially increasing the computation if we try to generate the second level. The module specification to encode it is shown in figure 12.

```
mod ABELIAN-GROUP is
   sorts Int State .

   ops 0 1 : -> Int [ctor] .
   op _+_ : Int Int -> Int [assoc comm prec 30] .
   op <_,_> : Int Int -> State [ctor] .
   op -_ : Int -> Int .

   vars X Y Z : Int .
```

```
  eq X + 0 = X [variant] .
  eq X + (- X) = 0 [variant] .
  eq X + (- X) + Y = Y [variant] .
  eq - (- X) = X [variant] .
  eq - 0 = 0 [variant] .
  eq (- X) + (- Y) = -(X + Y) [variant] .
  eq -(X + Y) + Y = - X [variant] .
  eq -(- X + Y) = X + (- Y) [variant] .
  eq (- X) + (- Y) + Z = -(X + Y) + Z [variant] .
  eq - (X + Y) + Y + Z = (- X) + Z [variant] .

  rl [r1] : < (X + 1),Y > => < ((X + 1) + (- 1)),(Y + 1) > [narrowing] .
endm
```

Figure 65: Module that specifies the properties of abelian groups in Maude.

Therefore, if we use the narrowing problem $< 0, (1 + X)> \leadsto^{*}_{R,E\cup B} < (\text{-}1), Y >$ to invoke the three commands, by generating only one level of the reachability tree, we will find 184 solutions, being the Maude native command slightly faster than our two commands. But if we take a look at Figure 66, which shows the previous comparison of the commands together with the comparison when trying to generate the second level of the reachability tree, we can see how the two standard narrowing commands (even the Maude native one) fail to terminate, while the canonical narrowing command, considering a smaller number of branches, it does, and in a relatively short time.

Algorithm	Max. depth	Max. solutions	Time	Rewrites	Solutions
Standard Narrowing	*1*	*unbounded*	349 ms	5359	184
Standard Narrowing (Native command)	*1*	*unbounded*	302 ms	1420	184
Canonical Narrowing	*1*	*unbounded*	304 ms	5141	184
Standard Narrowing	*2*	*unbounded*	∞	∞	-
Standard Narrowing (Native command)	*2*	*unbounded*	∞	∞	-
Canonical Narrowing	*2*	*unbounded*	1180 ms	8091	184

Figure 66: Comparison of algorithms for the mentioned narrowing problem by using the XOR-PROTOCOL module.

In this specific example, the number of solutions is the same in the first level of the tree as in the second, but in others, the possibility of generating one more level of the tree could show us new solutions, giving us a clear advantage in this type of problem. to the canonical narrowing command over the other two.

6. Conclusions and future work

In this chapter we present the conclusions obtained from the work carried out and the possible future work that can be based on it. Specifically, in Section 6.1 we will discuss the conclusions, and in Section 6.2, the potential future work.

6.1. Conclusions

As a result of the completion of this work, we can confirm that Maude is presented as a system with great potential for the definition of rewrite systems and their associated problems. For example, a very important characteristic of this system for this work is its relative ease in working with unification with axioms, something that other systems are not capable of doing. Thanks to this, we have been able to write the modules in which we implement standard narrowing and canonical narrowing.

Regarding the algorithms, we can say that the implementation of standard narrowing using Maude's reflexive level does not make much sense on its own, since Maude already includes a standard narrowing command that, having the base encoded in C++, is more efficient and quicker. However, the implementation of this makes sense if the intention is to create a basis for making modifications, improving the algorithm in different ways or even defining another type of narrowing. In our case, we have managed to start from that base to write a module that defines canonical narrowing in Maude, a narrowing that uses irreducibility constraints with the intention of improving many of the computations of the standard narrowing, through the early removal of irrelevant branches from the reachability tree.

The canonical narrowing implemented turns out to be, in most cases, more efficient and faster than the standard narrowing, even if we compare it with the standard narrowing already included in Maude itself. Only if we are going to generate the first level of the reachability tree is it usually more useful to use the standard narrowing algorithm, since the same result is obtained by performing fewer rewriting steps. Once we pass that level, thanks to the irreducibility conditions, the canonical narrowing will normally have to perform fewer rewriting steps. Thanks to this, the wider and deeper the reachability tree to be generated, the greater the benefit of using canonical narrowing compared to standard narrowing.

In the field of security protocols, the consideration of canonical narrowing can be very useful, not only to improve performance when solving reachability problems, but also to solve problems that the standard narrowing is not able to compute in a reasonable time. This usually occurs when the standard algorithm generates reachability trees with very large levels in amplitude, as we could see in the last example of the previous chapter.

For all this, we consider that the work carried out for this master thesis entails a notable improvement over what is already implemented in Maude and may lead to improvements in the language itself and the tools that use it, many of them, precisely for the analysis of vulnerabilities in protocols.

6.2. Future work

The work carried out in this master thesis leaves a foundation that can be extended to achieve even more sophisticated algorithms. The first task before us is to review and optimize some critical points of the implementation, managing to further maximize the performance of the canonical narrowing command in Maude. Once the algorithm has been reviewed to achieve the maximum possible performance, it must be borne in mind that the canonical narrowing, despite being better than the standard narrowing and other variants, is not perfect, and leaves room for some improvements that we discuss below.

Currently the algorithm performs a search of all possible unifiers to perform each narrowing step, but there will be many cases in which some of the unifiers are contained in others that may be more general. This is where the concept of the *most general unifiers* or *MGUs* [8] comes into play. If we make the algorithm calculate the most general unifier at each step, the unifiers that it subsumes can be ignored, and although computing time is wasted in that process, it will be profitable later because fewer nodes will be generated at the next level of the tree. Also, the solutions returned by the algorithm will be much more accurate. Maude already includes this mechanism for standard narrowing, although not in the meta-level. There is a variant of the *vu-narrow* command, called *filtered vu-narrow*, which computes only the MGUs instead of all the unifiers, and the comparison between both commands is a clear proof of the increase in performance that its use implies. If this functionality were implemented using canonical narrowing, the performance and efficiency improvement of the algorithm would be even better.

Finally, another interesting improvement for the future may be the union of the unification with axioms and variants with *satisfiability modulo theories (SMT) solving*, that is, in our case, canonical narrowing with SMT solving. Currently there are several definitions of conditional narrowing in which the conditions are precisely SMT solving rules, but none of them have been put into practice, that is, there is no implementation. Being able to unify the two areas at a practical level, using canonical narrowing would represent a notable improvement in this field.

If we put all the above together, we could achieve a canonical narrowing algorithm (which is an important improvement over standard narrowing) that takes into account only the MGUs, further improving the performance and precision of the solutions, and that also allows the use of SMT solving rules as conditions that serve as a guide for execution.

References

[1] Alpuente, M., Ballis, D., Escobar, S., & Sapiña, J. (2019). Retrieved July 24, 2021. Narval. Symbolic Analysis of Maude Theories with Narval. http://safe-tools.dsic.upv.es/narval/

[2] Basin, D., Cremers, C., Dreier, J., Meier, S., Sasse, R., & Schmidt, B. (n.d.). Tamarin Prover. Tamarin Prover. Retrieved September 9, 2021, from http://tamarin-prover.github.io/

[3] Cholewa, A., Meseguer, J., & Escobar, S. (2007, February 7). Variants of variants and the finite variant property. The Grainger College of Engineering. https://courses.engr.illinois.edu/cs576/sp2017/readings/07-feb-07/cholewa-meseguer-escobar-variants.pdf

[4] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., & Talcott, C. (2020, October). Maude Manual (Version 3.1). The Maude System. http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html

[5] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., & Talcott, C. (2007). All About Maude - A High-Performance Logical Framework. Springer Publishing. Google Scholar

[6] Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., & Talcott, C. (2020). Programming and symbolic computation in Maude. Journal of Logical and Algebraic Methods in Programming, 110, 100497. https://doi.org/10.1016/j.jlamp.2019.100497

[7] Escobar, S., & Meseguer, J. (2019). Canonical Narrowing with Irreducibility Constraints as a Symbolic Protocol Analysis Method. Foundations of Security, Protocols, and Equational Reasoning, 15–38. https://doi.org/10.1007/978-3-030-19052-1_4

[8] Escobar, S., & Sapiña, J. (2019). Most General Variant Unifiers. Electronic Proceedings in Theoretical Computer Science, 306, 154–167. https://doi.org/10.4204/eptcs.306.21

[9] Maude-NPA Protocols. (n.d.). Maude-NPA: Repository of Protocol Specifications in Maude-NPA. Retrieved September 9, 2021, from http://personales.upv.es/sanesro/Maude-NPA_Protocols/index.html

[10] Meseguer, J. (2012). Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming, 81(7–8), 721–781. https://doi.org/10.1016/j.jlap.2012.06.003

[11] Meseguer, J. (2021). Symbolic Computation in Maude: Some Tapas. Logic-Based Program Synthesis and Transformation, 3–36. https://doi.org/10.1007/978-3-030-68446-4_1

[12] The AKiSs Team. (n.d.). AKiSs. AKiSs. Retrieved September 9, 2021, from http://akiss.gforge.inria.fr/

[13] The Maude Team. (n.d.). Maude 3.1 Manual Book Examples. Retrieved July 26, 2021, from http://maude.cs.illinois.edu/w/images/4/4f/Maude-3.1-manual-book-examples.zip

[14] The Maude Team. (n.d.-b). Maude Tools: Maude-NPA - The Maude System. Maude. Retrieved September 9, 2021, from http://maude.cs.illinois.edu/w/index.php/Maude_Tools:_Maude-NPA