A Rodin Plug-in for Probabilistic Event-B based on a Rewriting Logic Approach

Daniel F. Osorio-Valencia

May, 2022

Abstract

Acknowledgments

Contents

Chapter 1

Introduction

Chapter 2

Literature Review

The main purpose of this chapter is to present the needed theoretical and technical background to understand how the plugin works. Therefore, a short but sufficient definition of various concepts and tools will be given. In terms of theoretical context, this chapter will discuss topics like Event-B, Maude and the rewriting logic approach to Event-B. Furthermore, this chapter will also provide some insight into plugin development in Rodin. To guide the reader through the different sections of this chapter and facilitate its reading, the following section dependency is given:

2.1 Event-B

2.2 Probabilistic Event-B

2.3 Maude

Maude is a high performance declarative language, that allows the specification of programs or systems, and their formal verification [1, 2, 3]. Maude programs are represented as *functional modules* declared with syntax:

```
fmod MODULENAME is
    BODY
endfm
```

where *MODULENAME* is the name of the functional module, and *BODY* is a set of declarations that specify the program. The body of the module contains *sorts* (written in Maude as `sorts`), where each sort correspond to an specific data type of the program. It also contains a set of function symbols or function declarations called *operators* (abbreviated as `op` in Maude), that specify the

constructors of the different sorts, along with the syntax of the program functions. Finally, a set of *equations* (abbreviated as `eq` in Maude) is used to define the behavior of the functions. These equations use *variables* (abbreviated as `var` in Maude) to describe how each function works.

To illustrate how a Maude program is contructed, the following code corresponds to a program that defines the natural numbers and the addition operation, borrowed from [3]:

```
fmod NAT-ADD is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

  *** Recursive Definition for addition
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

The sort `Nat` is a data type that represents the natural numbers. This sort has two constructors (represented in the code with the key word `ctor`): 0 which is a constant and the operator `s`, which takes one argument of type `Nat` and represents the successor function in the natural numbers. With these two operators, it is possible to define arithmetic functions in the natural numbers, like addition or multiplication. In this module, both functions are defined inductively using equations. Using this module, it is possible to compute the value for addition or multiplication for two natural numbers using the command `red`. For example, if the command `red s(s(s(0))) + s(s(0))` is used, that represents the operation 3 + 2, the answer 5 is obtained represented as `s(s(s(s(s(0)))))`:

```
********** equation
eq N + s(M) = s(N + M) .
N --> s(s(s(0)))
M --> s(0)
s(s(s(0))) + s(s(0))
-->
s(s(s(s(0))) + s(0))
********** equation
eq N + s(M) = s(N + M) .
N --> s(s(s(0)))
M --> 0
s(s(s(0))) + s(0)
```

```
-->
s(s(s(0))) + 0)
********** equation
eq N + 0 = N .
N --> s(s(s(0)))
s(s(s(0))) + 0
-->
s(s(s(0)))
result Nat: s(s(s(s(0)))))
```

Maude computes using equations from left to right. Therefore computation steps like the first one, the expression `s(s(s(0))) + s(s(0))` is matched with the left side of the equation `N + s(M) = s(N + M)` and matching substitution $\{\texttt{N} \mapsto \texttt{s(s(s(0)))}, \texttt{M} \mapsto \texttt{s(0)}\}$. The resulting expression `s(s(s(0)))` `+ s(s(0))`, will be simplified again with the same equation, until it reduces to a simplified expression that can be matched with the equation `N + 0 = N` (as seen in the last step).

Semantically, functional modules in Maude are represented as *equational theories* [2, 3], that are represented as a pair (Σ, E) where:

- the *signature* Σ describes the syntax of the theory, which is the data types and operators symbols (sorts and operators).
- E is the set of equations between expressions written in the syntax of Σ.

As mentioned before, computations in Maude are done by using the equations over expressions constructed with operators. This method is called *term rewriting* [2, 3] and behaves in the following way:

- With the equations E of (Σ, E), *term rewriting rules* are defined as $\vec{E} = \{u \to v \mid (u = v) \in E\}$.
- A term t, which are expressions formed using the syntax in Σ, is rewritten to t' in one step $t \to_{\vec{E}} t'$ if and only if, the following conditions are suffice:
  - there is a subterm w in t, expressed as $t[w]$.
  - there is a rule $(u \to v) \in \vec{E}$ and a substituon θ s.t. : $w = u\theta$, $w' = v\theta$, $t' = t[w'] = t[v\theta]$.

for example, in the previous computation `red s(s(s(0))) + s(s(0))` the term rewriting process in the second step, is the following:

- $E =$ `N+s(M)=s(N + M)`
- $t =$ `s(s(s(s(0))) + s(0))`
- $\theta = \{\texttt{N} \mapsto \texttt{s(s(s(0)))}, \texttt{M} \mapsto \texttt{0}\}$
- $w =$ `N+s(M)` $\theta = s(s(s(0))) + s(0)$

- $w' = \texttt{s (N+M)}\ \theta = s(s(s(0))) + 0)$
- $t' = s([w']) = s(s(s(s(0))) + 0))$

the resulting term rewriting is $t \rightarrow_{\vec{E}} t'$. Aside from building programs in Maude using functional modules, it is also possible to model concurrent systems. This is done with *system modules*, which permits the construction of system states and transitions. Semantically, a system module is a *rewrite theory* $\mathscr{R} = (\Sigma, E, L, R)$ where:

- (Σ, E) is an equational theory.
- L is a set of labels.
- R is a set of unconditional labeled rewrite rules of the form $l : t \rightarrow t'$, and conditional labeled rewrite rules fo the form $l : t \rightarrow t'$ if *cond*, where $l \in L$, t, t' are terms in Σ and *cond* is a condition or system guard.

The syntax for system modules in Maude is:

```
mod MODULENAME is
    BODY
endm
```

Where the body represents a rewrite theory $\mathscr{R}$. The syntax for unconditional rewriting rules is

```
rl [l] : t => t' .
```

and for conditional rewriting rules is

```
crl [l] : t => t' if cond .
```

to exemplify this, lets consider the following simple model of a bus: A transport bus has capacity for 60 people. The bus can be moving or stationary and can only drop or lift passengers when the bus is stationary. Finally, at any time the bus driver can use the brake to stop or use the gas pedal to move. The corresponding Maude system module for this model is:

```
mod BUS is protecting NAT .
  sorts Bus Status.

  op bus : Nat Status -> Bus [ctor] .
  ops stationary moving : -> Status [ctor] .

  vars N M : Nat . var S : Status .

  *** move the bus
  rl [move] : bus(N,stationary) => bus(N,moving) .
  *** stop the bus
  rl [stop] : bus(N,moving) => bus(N,stop) .
```

```
  *** lift passenger
  crl [lift] : bus(N,stationary) => bus(N + 1,stationary)
                                    if N + 1 >= 60 .
  *** drop passenger
  crl [drop] : bus(N,stationary) => bus(N - 1,stationary)
                                    if N - 1 >= 0 .
endm
```

2.4 Probabilistic Maude (PMaude)

2.5 PVeStA

2.6 A Rewriting Logic Semantics and Statistical Analysis for Probabilistic Event-B

2.7 Rodin and Plugin Development

Chapter 3

Methodology

Chapter 4

Results

Chapter 5

Discussion

Chapter 6

Conclusions

Bibliography

[1] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Maude manual (version 3.2.1)," 2022.

[2] J. Meseguer, "Maude summer school: Lecture 1." `https://nms.kcl.ac.uk/maribel.fernandez/PhD-SummerSchoolMaude.html`, 2022.

[3] P. C. Ölveczky, *Designing Reliable Distributed Systems*. Springer London, 2017.