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 their reading, the following section dependency is given:

2.1 Event-B

Event-B [1, 2] is a formal method for specifying and verifying properties about systems based on set theory and predicate logic. Specifications in Event-B are referred as models and they contain the mathematical development for a discrete transition system. The main components of an Event-B model are machines and contexts: machines contain the dynamic elements of the model, while contexts contain the static elements. Specifically, machines contain the system variables, invariants, theorems, variant and events. On the other hand, contexts contain carrier sets, constants, axioms and theorems. Both machines and contexts have different interactions between them. A machine can "see" one or several context, meaning that the machine includes the static elements specified in these contexts. Furthermore, machines can "refine" other machines, which means that if a machine M_2 refines a machine M_1, then M_2 can use all the dynamic elements specified in M_1 and also create new ones that respect the properties defined in M_1 (for example properties defined as invariants). This relation also occurs in the same way between contexts, when one context "extends" another one. In terms of syntax, the general structure for contexts is the one expressed in figure 2.1, such that:

- *context_identifier* is a string that identifies the context. The context identifier must be different to all the other identifiers of the components of the system (other machines or contexts).
- The "extends" clause lists all the contexts identifiers that the current context is extending.
- The "sets" clause takes a list of set identifiers. Each one of these set identifiers correspond to the name of one of the carrier sets of the model. A carrier set is a user defined set or type, that must be non-empty. The set of set identifiers is defined as $\bar{s} = \{s_1...s_n\}$.
- The "constants" clause lists all the constants of the model. The set of constants identifiers is defined as $\bar{c} = \{c_1...c_n\}$.
- The "axioms" clause introduces the list of axioms that the model must satisfy. The Axioms are a conjunction logic predicates over sets $\bar{s}$ and constants $\bar{c}$ defined as $A(\bar{s}, \bar{c})$. These logic predicates state the properties that the constants and sets must meet.
- The "theorems" clause lists some logic predicates, that must be proved within the context using the axioms. These are logic predicates defined as $T_i(\bar{s}, \bar{c})$.

```
< context_identifier >
  extends
    < context_identifier_list >
  sets
    < set_identifier_list >
  constants
    < constant_identifier_list >
  axioms
    < label >:  < predicate >
    ...
  theorems
    < label >:  < predicate >
    ...
  end
```

Figure 2.1: Context structure, taken from [1]

For the Machine, the general structure of it is specified in figure 2.2. Each one of the components corresponds to:

- *machine_identifier* is a string that identifies the machine. It must be different from all the other components identifiers.
- The "refines" clause contains the identifier of the machine that this machine refines. The machine that is refined is called *abstract machine*.
- Clause "sees" lists the context that the machine is referencing. When a machine "sees" a context, it means that it can use the sets and constants defined in the context.

```
< machine_identifier >
  refines
    < machine_identifier >
  sees
    < context_identifier_list >
  variables
    < variable_identifier_list >
  invariants
    < label >:  < predicate >
    ...
  theorems
    < label >:  < predicate >
    ...
  variant
    < variant >
  events
    < event_list >
  end
```

Figure 2.2: Machine structure, taken from [1]

- The clause "variables" lists all the variables of the system. The set of variable identifiers is defined as $\bar{v} = \{v_1 ... v_n\}$.
- The "invariants" clause lists all the invariants that the model must satisfy. Invariants are represented as a conjunction of logic predicate over variables $\bar{v}$, defined as $I(\bar{v})$. These invariants define the properties that the variables must hold in every configuration or state of the system.
- The "theorems" clause lists all the theorems. Each theorem is a logic predicates that must be proven using the axioms of the context and the invariants of the machine. They have the form $T_i(\bar{v})$.
- The "variant" clause is used when machines have *convergent events*. It specifies an expression $V(\bar{v})$ over the variables used for proving convergence of the model, i.e. proving that the model doesn't loops forever.
- The "events" clause lists the events of the model. Each event represents a system transition, that changes the current system state to another.

Events are very important components of an Event-B model and they have the following structure:

$$\textbf{event } e \textbf{ any } \bar{t} \textbf{ where } G(\bar{t}, \bar{v}) \textbf{ then } S(\bar{t}, \bar{v}) \textbf{ end}$$

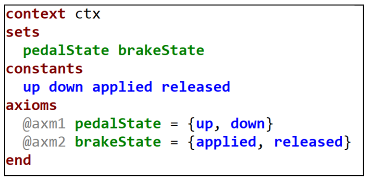
- e is the *event_identifier*, which is a string that identifies the event.
- The "any" clause lists the parameters of the event. The set of parameters of an event is represented by $\bar{t} = \{t_1, ..., t_n\}$.
- The "where" clause contains the guards of the event. Guards are represented as a conjunction of logic predicates $G(\bar{t}, \bar{v})$ over the parameters of the event and variables of the system. These guards specify the conditions that must hold for the event to be enabled.
- The "then" clause contains the list of actions of the event. Actions define variable assignments that change the variable values when the event is

executed. These actions are represented as a set of assignments $S(\bar{t}, \bar{v})$.

Assignments in the action of an event can be categorized in three types:

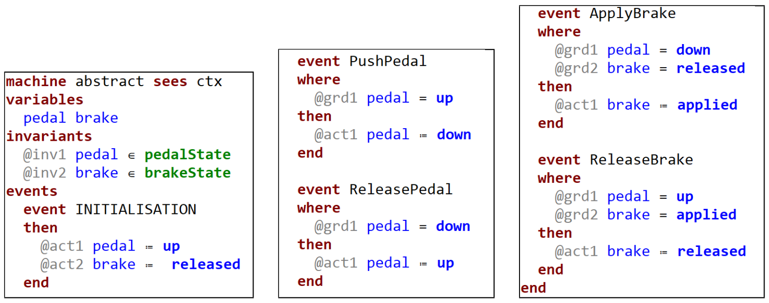
- Deterministic assignment $x := E(t, v)$ states that if the event is executed, then the value of variable x in the next model state will be $E(t, v)$, where E is an expression over the parameters of the event and the variables of the system.
- Non-deterministic assignment $x :\in \{E_1(\bar{t}, \bar{v}) ... E_n(\bar{t}, \bar{v})\}$ called enumerated assignment. In this case, one of the expressions E_i of the set is assigned to variable x, when the event is executed.
- Non-deterministic assignment $x : | \ Q(\bar{t}, \bar{v}, x, x')$ called predicate assignment. It assigns to variable x the value x', s.t. x' satisfies the predicate $Q(\bar{t}, \bar{v}, x, x')$.

The set of events of a model is called as $Evts$. This set includes the *INITIALISATION* event or $Init$, which assigns the initial value to all the variables of the machine and creates the initial state of the model. A whole Event-B model can be represented as a context $\mathscr{C} = (\bar{s}, \bar{c}, A(\bar{s}, \bar{c}))$ and a machine $\mathscr{M} = (\bar{v}, I(\bar{v}), V(\bar{v}), Evts, Init)$. To clarify this notation, an example of an Event-B model of a brake system, presented in [3], will be used. A brake system consists of two parts: a pedal and a brake. The pedal can be up or down and the brake can be applied or released. When the pedal is up, the brake is released. When the pedal is down, the brake is then applied. The resulting Event-B specification for such model is represented in figure 2.3 and 2.4. The context identifier



```
context ctx
sets
  pedalState brakeState
constants
  up down applied released
axioms
  @axm1 pedalState = {up, down}
  @axm2 brakeState = {applied, released}
end
```

Figure 2.3: Context of the brake example



```
machine abstract sees ctx
variables
  pedal brake
invariants
  @inv1 pedal ∈ pedalState
  @inv2 brake ∈ brakeState
events
  event INITIALISATION
  then
    @act1 pedal ≔ up
    @act2 brake ≔  released
  end
```

```
  event PushPedal
  where
    @grd1 pedal = up
  then
    @act1 pedal ≔ down
  end

  event ReleasePedal
  where
    @grd1 pedal = down
  then
    @act1 pedal ≔ up
  end
```

```
  event ApplyBrake
  where
    @grd1 pedal = down
    @grd2 brake = released
  then
    @act1 brake ≔ applied
  end

  event ReleaseBrake
  where
    @grd1 pedal = up
    @grd2 brake = applied
  then
    @act1 brake ≔ released
  end
end
```

Figure 2.4: Machine of the brake example

for this model is *ctx*, the set of set identifiers is $\bar{s} = \{pedalState, brakeState\}$ and the set of constants is $\bar{c} = \{up, down, applied, released\}$. The *pedalState* carrier set contains the possible states of the pedal (up or down) and the *brakeState* carrier set contains the possible states of the brake (applied or released). Both of these sets, are defined using the axioms $axm1 = A_1(\bar{s}, \bar{c})$ and $axm2 = A_2(\bar{s}, \bar{c})$. Therefore, the axioms of the brake model are defined by $A(\bar{s}, \bar{c}) = A_1(\bar{s}, \bar{c}) \wedge A_2(\bar{s}, \bar{c})$. In particular, this context doesn't extend other components or uses theorems.

For the model's machine the machine identifier is *abstract* and it "sees" the sets, constants and axioms in *ctx*. The set of variables in the brake model is defined as $\bar{v} = \{pedal, brake\}$. The *pedal* variable represents the state in which the pedal is and the *brake* variable represents the brake's state. Invariants $inv1 = I_1(\bar{v})$ and $inv2 = I_2(\bar{v})$ define the domain for variables *pedal* and *brake* respectively, by assigning to each one of them their respective carrier set. The resulting invariant for the brake model is $I(\bar{v}) = I_1(\bar{v}) \wedge I_2(\bar{v})$. Each one of the events in the model represent a state transition and a system action:

- The *INITIALISATION* event is executed when the model simulation starts. In this case, the action $S(\bar{t}, \bar{v})$ of the *INITIALISATION* event is defined by the two assignments $pedal := up$ and $brake := released$.
- The *PushPedal* event represents the action of pushing the pedal. The guard of the event $G(\bar{t}, \bar{v}) = G_1(\bar{t}, \bar{v}) = grd1$ states that he pedal must be *up*, to execute the action $S(\bar{t}, \bar{v}) = S_1(\bar{t}, \bar{v}) = act1$. This action assigns the value *down* to the *pedal* variable. Conversely, the *ReleasePedal* event guard verifies that the *pedal* variable is *down*, and changes it value to *up* when executed.
- The *ApplyBrake* event represents the action of applying the brake. The guard of the event $G(\bar{t}, \bar{v}) = G_1(\bar{t}, \bar{v}) \wedge G_2(\bar{t}, \bar{v}) = grd1 \wedge grd2$ states that the pedal must be *down* and the brake *released*, to execute the action $S(\bar{t}, \bar{v}) = S_1(\bar{t}, \bar{v}) = act1$. This action assigns the value *applied* to the *brake* variable. On the other hand, the *ReleaseBrake* check for the opposite conditions ($pedal = up$ and $brake = applied$) and changes the value of the *brake* variable to *released*.

2.2 Non-deterministic choices in Event-B

When an Event-B model is simulated or verified, there exists the possibility that in a system state there are multiple possibilities to determine the next transition and state of the model. The way Event-B resolves this multiple choice dilemma is with *non-deterministic choice* and there are 3 types of it:

- **Choice of enabled events:** When multiple events are enabled for execution, one of them is chosen non-deterministically.
- **Choice of parameter values:** In an event with parameters, it is possible

to have multiple valuations for parameters $\bar{t}$ such that the guard of the event $G_i(\bar{t}, \bar{v})$ is satisfied. Therefore, the parameter that will be used in the execution of the event is chosen non-deterministically

- **Non-Deterministic assignments:** As mentioned before, there are 3 types of event assignments: Deterministic assignment, Predicate assignment and Enumerated assignment. Both predicate and enumerated assignments, are non-deterministic for the following reason:
  - Predicate assignment $x :\mid Q(\bar{t}, \bar{v}, x, x')$: means that the variable x is assigned the value x' that satisfies the predicate $Q(\bar{t}, \bar{v}, x, x')$. In case there are multiple x' values that satisfy the predicate, then the new value of x is chosen non-deterministically.
  - Enumerated assignment $x :\in \{E_1(\bar{t}, \bar{v}) ... E_n(\bar{t}, \bar{v})\}$: means that the variable x is assigned on of the multiple expressions $E_i(\bar{t}, \bar{v})$ in the set. This choice of which expression will be chosen, is done non-deterministically.

2.3 Probabilistic Event-B

Based on the 3 types of non-determinism present in Event-B, probabilistic Event-B [3] aims to introduce probabilistic choices to replace non-deterministic ones in the following way:

- **Probabilistic choice of enabled events:** To solve the non-determinisim, an expression $W_i(\bar{v})$ over the variables represent the *weight* of a specific event e_i. Therefore, when multiple events are enabled, the probability of choosing one of them will be the ratio of its weight over the sum of all the weights of enabled events.
- **Probabilistic choice of parameter values:** In order to choose a parameter value probabilistically, a discrete uniform distribution can be used as a default choice to assign probabilities to the parameters. For that reason, the probability of choosing a parameter valuation $P(t_i) = \frac{1}{n}$ where n is the number of parameter valuations that satisfy the guard of the event.
- **Predicate probabilistic assignment:** A predicate probabilistic assignment written as $x :\oplus Q(\bar{t}, \bar{v}, x, x')$ chooses the new value of x with an uniform distribution. Hence, the probability of choosing a variable value x'_i is $P(x'_i) = \frac{1}{n}$ where n is the number of variable valuations that satisfy the predicate $Q(\bar{t}, \bar{v}, x, x')$. This probabilistic assignment replaces the non-deterministic predicate assignment.
- **Enumerated probabilistic assignment:** A enumerated probabilistic assignment written as $x := E_1(\bar{t}, \bar{v})@_{p_1} \oplus ... \oplus E_n(\bar{t}, \bar{v})@_{p_n}$ assigns a specific probability p_i to each expression E_i, where $0 < p_i \leq 1$ and $p_1 + ... + p_n = 1$. This probabilistic assignment replaces the non-deterministic enumerated

assignment.

Based on this changes, the new structure for probabilistic events can be defined as:

$$e \mathrel{\widehat{=}} \textbf{weight } W(\bar{v}) \textbf{ any } \bar{t} \textbf{ where } G(\bar{t},\bar{v}) \textbf{ then } S(\bar{t},\bar{v}) \textbf{ end}$$

Where $W(\bar{v})$ is an expression over the variables that determines the weight of the event, $G(\bar{t},\bar{v})$ the guard of the event and $S(\bar{t},\bar{v})$ is the *probabilistic action.* The probabilistic action contains only deterministic assignments, predicate probabilistic assignments and enumerated probabilistic assignments. The resulting Machine for a probabilistic model is then $\mathscr{M} = (\bar{v}, I(\bar{v}), V(\bar{v}), PEvts, Init)$ where $PEvts$ is a set of probabilistic events and $Init \in PEvts$. To exemplify probabilistic Event-B, lets consider an extension of the brake model example. This new example, presented also in [3], adds new constraints:

R1. Pedal failure: when the driver tries to switch "down" the pedal, it may stay in the same position.

R2. Risk of pedal failure: the risk of pedal failure is set to 10%.

R3. brake failure: the brake may not be applied although the pedal has been switched down.

R4. Maximum brake wear: the brake cannot be applied more than a fixed number of times.

R5. Brake wear: due to brake wear, the risk of brake failure increases each time the brake is applied.

The resulting probabilistic model has the same context presented in figure 2.3 but has a modified machine. This machine is displayed in figure x .

2.4 Maude

Maude [4, 5, 6] is a high performance declarative language, that allows the specification of programs or systems, and their formal verification. 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 [5]:

```
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(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(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 {`N` $\mapsto$ `s(s(s(0)))`,`M` $\mapsto$ `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* [6, 5], 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* [6, 5] and behaves in the following way:

- With the equations E of (Σ, E), *term rewriting rules* are defined as $\overrightarrow{E} = \{u \to v \mid (u = v) \in E\}$.
- A term t, which are expressions formed using the syntax in Σ, is rewritten to t' in one step $t \to_{\overrightarrow{E}} t'$ if and only if, the following conditions are suffice:
  - there is a subterm w in t, expressed as $t[w]$.
  - there is a rule $(u \to v) \in \overrightarrow{E}$ 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))`
- $\theta = \{$`N` $\mapsto$ `s(s(s(0)))`,`M` $\mapsto$ `0`$\}$
- $w =$ `N+s(M)`$\theta =$ `s(s(s(0))) + s(0)`
- $w' =$ `s(N+M)`$\theta =$ `s(s(s(s(0))) + 0)`
- $t' =$ s$([w'])$ $=$ `s(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* [5, 7] $\mathscr{R} = (\Sigma, E, L, R)$ 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,stationary) .
   *** drop passenger
   rl [drop] : bus(s(N),stationary) => bus(N,stationary) .
   *** lift passenger
```

```
  crl [lift] : bus(N,stationary) => bus(s(N),stationary)
                                    if s(N) <= 60 .
endm
```

In this model the states of the system are represented with instances of the sort `Bus`, and it contains a natural number that represents the number of people inside the bus and a `Status`, which represents the state of the bus (it can be stationary or moving). In this case, no equations are used, therefore the set of equations $E = \emptyset$ and Σ will contain the sorts `Bus` and `Status` with their respective operators. To model the different events in the model, 4 rewriting rules were used:

- An unconditional rule labeled `move`, that represents the event of using the gas pedal to move the bus by changing the status from `stationary` to `moving`. Note that this rule can only be applied when the bus is `stationary`, as stated by the rule first term `bus(N,stationary)`.
- An unconditional rule labeled `stop`, that represents the event of using the brakes to stop the bus. This changes the status of the bus from `moving` to `stationary`. As the previous rule, it can only be applied when the first term is matched, i.e. when the bus status is `moving`.
- An unconditional rule labeled `drop`, that represents the event of dropping people off the bus. The subterm `s(N)` assures that the it can only drop a person when the number of people in one or more. This rule rewrites the state of the system, by reducing the number of passengers in the bus by one.
- A conditional rule labeled `lift`, that represents the event of lifting a passenger. When this rules is applied, the number of passengers inside the bus is increased by one. To prevent exceeding the maximum capacity of the bus, the condition `if s(N) <= 60` is used.

With this system module, that represents a rewrite theory $\mathscr{R}$, the simple bus model can be specified and verified using other functionalities in Maude like model checking with the commands `rewrite` and `search`. The `rewrite` command or `rew` takes an initial state of the system and uses the rewriting rules until termination, i.e. until no other rewriting rule can be applied to the system state. In the case of the bus example, the rules `move` and `stop` can be applied infinitely. Therefore, to be able to simulate this system Maude also allows to use bounded rewriting. With this method it is possible to specify the number of rewriting rules to be applied to the initial state of the system. For example, the command `rew [3] bus(0, stationary) .` will apply 3 rewriting rules to to the state `bus(0, stationary)`, which refers to a stationary bus with no passengers. The resulting execution is:

```
*********** rule
rl bus(N, stationary) => bus(N, moving) [label move] .
```

```
bus(0, stationary)
--->
bus(0, moving)
*********** rule
rl bus(N, moving) => bus(N, stationary) [label stop] .

bus(0, moving)
--->
bus(0, stationary)
*********** rule
crl bus(N, stationary) => bus(s N, stationary) if s N <= 60 = true [label lift]

bus(0, stationary)
--->
bus(1, stationary)
result Bus: bus(1, stationary)
```

Lastly, with the `search` command it is possible to verify if a given state is reachable. For example, to check if the state `bus(10,stationary)` can be reached from the initial state `bus(0,moving)`, the command `search bus(0,stationary) =>* bus(10, moving) .` can be used. The result of this command is:

```
search in BUS : bus(0, stationary) =>* bus(10, moving) .

Solution 1 (state 21)
states: 22  rewrites: 50 in 0ms cpu (0ms real) (~ rewrites/second)
empty substitution

No more solutions.
```

If the search returns a solution, it means that the state is reachable. Furthermore it is also possible to check if the system will exceed the maximum capacity defining an system invariant I that states this property. This can be done, by adding the following code to the bus module:

```
*** Define predicates
  var X : Bus .
  op predicate : Bus -> Bool .
  eq predicate(bus(N,S)) = if N <= 60 then true else false fi .
```

This invariant can be checked with the `search` command using:

```
search bus(0,stationary) =>* X  s.t. predicate(X) =/= true .
```

which searches for a state `X` where the bus has more than 60 passengers. The resulting execution of the command returns no solution:

```
search in BUS : bus(0, stationary) =>* X such that predicate(X) =/= true = true

No solution.
```

This means that the bus system can not exceed the maximum capacity of the bus.

2.5 Object-Based Programming

Object-based programming in Maude [4, 7, 5] is supported by a predefined module CONFIGURATION. This module contains the necessary sorts and syntax to define the objects, messages, system configurations and objects interactions, of an object-based system. This module is defined as:

```
mod CONFIGURATION is
  *** basic object system sorts
  sorts Object Msg Configuration .
  *** construction of configurations
  subsort Object Msg < Configuration .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
[ctor config assoc comm id: none] .
```

The basic sorts are Object, Msg and Configuration. A term of sort Object represents an instance of a system object, a term of sort Msg represents a message shared by the system objects and a term of sort Configuration represents a snapshot of the current system state, represented as a multiset of objects and messages. These configurations are built with the multiset union operation (defined with syntax __) between objects, messages or other configurations, and the empty configuration is defined as none. The module configuration also implements a predefined syntax for object construction. They have the form:

$$\langle O : C \mid a_1 : v_1, ...a_n : v_n \rangle$$

Where O is the objects identifier, C is its class, $a_1...a_n$ are attribute identifiers and $v_1...v_n$ are the values of each one of the attributes. This defined syntax in Maude is implemented as:

```
  *** Maude object syntax
  sorts Oid Cid .
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
  [ctor assoc comm id: none] .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
endm
```

Where `Oid` corresponds to the object identifier O, `Cid` is the class identifier C and the attributes of the object are represented as a multiset. Messages' syntax is defined by the user. To understand how systems are modeled in Maude with configurations, the bank account example in [4] can be examined:

```
mod BANK-ACCOUNT is
     protecting INT .
     protecting CONFIGURATION .
     op Account : -> Cid [ctor] .
     op bal :_ : Int -> Attribute [ctor gather (&)] .
     op from_to_transfer_ : Oid Oid Nat -> Msg [ctor] .

     vars A B : Oid .
     vars M N L : Nat .

     *** Definition of transfer rewriting rule
     crl [transfer] :
       (from A to B transfer M)
       < A : Account | bal : N >
       < B : Account | bal : L >
       => < A : Account | bal : N - M >
          < B : Account | bal : L + M >
       if N >= M .

     *** Definition of the initial configuration
     op bankConf : -> Configuration .
     ops A-001 A-002 : -> Oid .
     eq bankConf
     = < A-001 : Account | bal : 250 >
       < A-002 : Account | bal : 1250 >
       (from A-002 to A-001 transfer 300) .
endm
```

The system consists of different bank accounts, that can transfer money to each other. The bank account class is named as `Account`, and it contains the attribute `bal` which corresponds to the amount of money available in an account. The message `from_to_transfer_` simulates the transfer request from one account to another, by specifying the the two account object identifiers or Oid (one for the account that sends the money and the other one for the account that receives the money) and the amount of money to be sent as an integer. The rewrite rule `transfer` matches a system configuration where the message `from A to B transfer M` and the two accounts with Oid `A` and `B` are present. After that, the rule modifies the attribute `bal` of both accounts according to the transaction parameters and erases the message from the configuration. For example, if the initial configuration of the system is the one defined by `bankConf`, the result after using the rewriting command `rew` `in BANK-ACCOUNT : bankConf .` would be:

```
result Configuration: < A-001 : Account | bal : 550 >
                      < A-002 : Account | bal : 950 >
```

2.6 PMaude

Probabilistic Maude or PMaude [8], is a Maude extension that introduces probabilities to the language. The underlying theory behind PMaude are *probabilistic rewrite theories* which correspond to an extension of rewrite theories: probabilistic rewrite theories can be expressed as tuple $\mathscr{R}_p = (\Sigma, E, L, R, \pi)$, where $(\Sigma, E, L, R,)$ is a rewrite theory and π is a function that assigns to each rewrite rule $r \in R$ a probability, given the current model state or configuration. This probability will determine if a rule may or may not be executed in the following system transition. The general form of probabilistic rewrite rules, for unconditional and conditional respectively is:

$$l : t(\overrightarrow{x}) \to t'(\overrightarrow{x}, \overrightarrow{y}) \text{ if } C(\overrightarrow{x}) \textbf{ with probability } \overrightarrow{y} := \pi_r(\overrightarrow{y})$$
$$l' : t(\overrightarrow{x}) \to t'(\overrightarrow{x}, \overrightarrow{y}) \textbf{ with probability } \overrightarrow{y} := \pi_r(\overrightarrow{y})$$

Where $\overrightarrow{x}$ is the set of variables of the current model state, $\overrightarrow{y}$ is the set of new variables accessible in the following model state and $C(\overrightarrow{x})$ is the conjunction of conditions over the set $\overrightarrow{x}$. Moreover, l, l' are labels in L, t, t' are terms written with Σ and π_r corresponds to the probability function assigned to the specific rule $r \in R$. Lets consider the PMaude module, presented in [8]:

```
pmod EXPONENTIAL-CLOCK is
  *** the following imports positive real number module
  protecting POSREAL .

  *** the following imports PMaude module that defines
  *** the distributions EXPONENTIAL, BERNOULLI, GAMMA, etc.
  protecting PMAUDE .

  *** declare a sort Clock
  sort Clock .
  *** declare a constructor operator for Clock
  op clock : PosReal PosReal => Clock .
  *** declares a constructor operator for a broken clock
  op broken : PosReal PosReal => Clock .

  *** T is used to represent time of clock,
  *** C represents charge in the clocks battery,
  *** t represents time increment of the clock
  vars T C t : PosReal . var B : Bool .
  rl [advance]: clock(T,C) =>
                   if B then
                         clock(T+t ,C - $\frac{C}{1000}$)
                   else
                         broken(T,C - $\frac{C}{1000}$)
                   fi
                with probability B:=BERNOULLI($\frac{C}{1000}$)
                                 and
                                 t:=EXPONENTIAL(1.0).

  rl [reset]: clock(T,C) => clock(0.0,C) .
endpm
```

This model represents a clock that works with a battery. The idea is to model the behavior of the clock, when the battery starts depleting: when the charge of the battery is high, then the probability that the clock breaks is low. Conversely, when the clock's battery is low, the clock has a higher probability of breaking. In this probabilistic system module, the clock is represented as a term `clock(T,C)`, where `T` is the time and `C` is the charge of the clock. The main probabilistic rewrite rule `advance` represents the "ticks" of the clock. If the boolean value `B` is true, then the clocks ticks normally and the new time will be the current time `T` plus an increment `t`. Also, the charge of the clock will be reduced by a thousandth of the currents clock's charge. If `B` is false, then the clock will break and change to the state `broken(T,C -` $\frac{C}{1000}$`)`. The constructor `broken` of sort `Clock`, represents the broken state of the clock. To incorporate the probabilistic choice of event for the clock's state (either ticking or breaking), the value `B` is chosen probabilistically, based on the charge of the clock. This is done by the `BERNOULLI` function, which receives a float number and returns a boolean value that is distributed according to the Bernoulli distribution with mean $\frac{C}{1000}$. Therefore, the lesser the charge left in the battery, the greater is the probability that the clock will break. The value `t` is also probabilistically determined, in this case by a exponential distribution function. There is also a second rewriting rule, that resets the clock to its initial state `clock(0.0, C)`. It is important to remark that this model has both probabilistic and non-deterministic choice: The state of the clock depends on a probability function but the choice of rewriting rules is done non-dereministically by Maude's fair scheduler.

PMaude modules can be transformed into regular system modules in Maude. This is done with three key modules: `COUNTER`, `RANDOM` and `SAMPLER`. The built-in `COUNTER` module in Maude consists of the rewriting rule

```
rl counter => N:Nat .
```

that rewrites the constant counter to a natural number. The module is built to guarantee that every time the constant counter is replaced with a natural number `N`, this natural number corresponds to the successor of the natural number obtained in the previous use of the rule. The built-in `RANDOM` module provides a random number generator function, called `random`. Lastly, the `SAMPLER` module specifies the sampling functions for different probability functions. For example, for the previous clock example, the needed functions will be:

```
op EXPONENTIAL : PosReal => PosReal .
op BERNOULLI : PosReal => Bool .
```

that are defined as:

```
rl EXPONENTIAL(R) => (- log(rand)) / R .
rl BERNOULLI(R) => if rand < R then true else false fi .
```

The value `rand` in both of the rules is defined as:

```
rl [rnd] : rand => float(random(counter + 1) / 4294967296) .
```

and it is rewritten in each step to a random number between 0 and 1. The number 4294967296 is used to divide the number returned by `random`, since it is the maximum number the function can return. The resulting Maude system module [8] is:

```
mod EXPONENTIAL-CLOCK-TRANSFORMED is
  *** The SAMPLER mode includes the COUNTER and RANDOM modules
  protecting SAMPLER .
  protecting POSREAL .

  sort Clock .
  op clock : Nat Float -> Clock [ctor] .
  op broken : Nat Float -> Clock [ctor] .

  vars T C : PosReal .

  rl clock(T,C) => if BERNOULLI($\frac{C}{1000}$) then
                         clock(T + EXPONENTIAL(1.0), C - $\frac{C}{1000}$)
                   else
                         broken(T,C - $\frac{C}{1000}$)
                   fi .

  rl [reset]: clock(T,C) => clock(0.0,C) .
endm
```

2.7 PVeStA

PVeStA [9] is a statistical model checking tool for probabilistic-real time systems, that are specified either as discrete or continuous Markov chains, or probabilistic rewrite theories in Maude. It permits the verification of properties expressed in quantitative temporal logics like QuaTex [8] using Monte Carlo simulations. PVeStA is implemented in Java and consists of two executable programs: a client `pvesta-client` and a server `pvesta-server`. The `pvesta-client` program takes as input the Maude probabilistic model, a list of formulas written in QuaTex, multiple execution parameters for the model checking simulation and a server list. Then it sends to every one of the running `pvesta-servers` (PVeStA's model checking supports parallelism) a simulation request for the specified probabilistic Maue model and QuaTex formulas. After that, the servers return the results back to the `pvesta-client` program. This previously explained process can be viewed in the following figure: To demonstrate how PVeStA works, a variation of the clock model shown

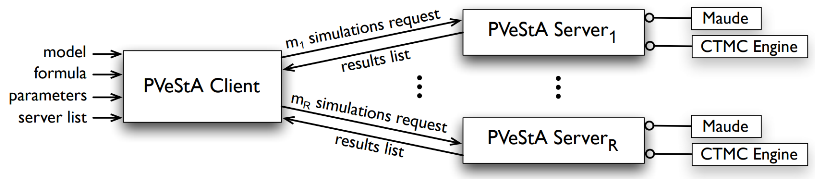


Figure 2.5: Components and interactions of PVeStA, taken from [9]

in the previous section will be used.

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

2.9 Rodin and Plugin Development

Chapter 3

Methodology

Chapter 4

Results

Chapter 5

Discussion

Chapter 6

Conclusions

Bibliography

[1] J. R. Abrial, *Modeling in event-b: System and software engineering*, vol. 9780521895569. Cambridge University Press, 1 2011.

[2] M. Butler, A. S. Fathabadi, and R. Silva, "Event-b and rodin," *Industrial Use of Formal Methods: Formal Verification*, pp. 215–245, 1 2013.

[3] M. A. Aouadhi, B. Delahaye, and A. Lanoix, "Moving from event-b to probabilistic event-b," *Proceedings of the ACM Symposium on Applied Computing*, vol. Part F128005, pp. 1348–1355, 4 2017.

[4] 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.

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

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

[7] J. Meseguer, "Maude summer school: Lecture 3-ii." `https://nms.kcl.ac.uk/maribel.fernandez/PhD-SummerSchoolMaude.html`, 2022.

[8] G. Agha, J. Meseguer, and K. Sen, "Pmaude: Rewrite-based specification language for probabilistic object systems," *Electronic Notes in Theoretical Computer Science*, vol. 153, pp. 213–239, 5 2006.

[9] M. AlTurki and J. Meseguer, "Pvesta: A parallel statistical model checking and quantitative analysis tool," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6859 LNCS, pp. 386–392, 2011.