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

Nowdays, computer systems are present in people's everyday life. Airplanes, cars, factories, banks or even household appliances integrate these systems in order to function correctly. Considering the influence of computer systems in our society, it is important to construct them in a rigorous manner and verify their correctness, to avoid errors or bugs that can affect negatively people's life. One way of accomplishing this goal is by using formal methods, which use mathematical models for analysis and verification of software or hardware [1]. Formal methods provide a general structure for defining real world systems as abstract models with mathematical rigor, that can be proven to be correct and implemented as specific pieces of software or hardware.

One of the formal methods used in the software industry is Event-B. It derives from the B method [2], it is semantically based on labeled transition systems (LTS) [3] and it is used for specifying discrete distributed systems [4]. This method, in conjunction with the Rodin platform [5], provide a framework for specifying Event-B models and proving mathematical properties over them. Some examples of the uses of Event-B in the design and verification of real world systems include the safety analysis on the CBTC system Octys of the Paris metro lines [6] and the formal development of the software for the BepiColombo space mission [7].

Even though Event-B proves to be a solid tool for formal specification and verification of systems, throughout its history the Event-B method has been extended with new features that increase its ability to model new systems. For example, Hybrid Event-B incorporates continuous behaviors to the discrete structure of Event-B, which facilitates the construction of cyber physical systems [8]. Another example is Distributed B: it is an extension that provides the necessary tools for modeling grid systems and verifying their correctness [9].

Another example of a Event-B extension, is probabilistic Event-B [10], which aims to introduce probabilistic behavior in Event-B. As systems grow in complexity, there is an increasing demand for probabilistic modeling features inside Event-B, where properties like

reliability and responsiveness need to be taken into account in the formal verification of these systems [11]. Therefore, several attempts have been made to move from standard Event-B to probabilistic Event-B. For instance, there is a fully probabilistic extension of Event-B that replaces all non-deterministic choices with probabilistic ones [11]. Other extensions try to introduce probabilistic choice by using qualitative probabilistic assignments [12] or quantitative probabilistic choice [13], instead of the non-deterministic assignments used in regular Event-B.

Furthermore, Event-B is not the only specification language where probabilistic extensions have been proposed: Maude [14] is a modeling language used to define formal models of distributed systems, based on rewriting logic [15]. A probabilistic extension of Maude, named PMaude [16], permits probabilistic modeling of concurrent systems. Paired with this extension and the query language known as QuaTEx [16], statistical model checking tools like PVeStA [17] allow statistical model checking of properties expressed as quantitative temporal logics for PMaude specifications.

Considering the need for specification and verification of probabilistic systems, the available probabilistic extensions of Event-B, and the availability of a framework for probabilistic system specification and verification provided by PMaude and PVeStA, the authors in [18] present a rewriting logic semantics for a probabilistic extension of Event-B. The previously mentioned paper, provides an automated process for translating an Event-B specification into a probabilistic rewrite theory $\mathcal{R}_{\mathcal{M}}$, where Monte Carlo simulations can be run using the PVeStA tool, to verify properties over the model written as QuaTEx queries. The translation process is also implemented as a tool that takes an Event-B model and turns it into a PMaude model that is executable using PVeStA, and it is available in [19].

Despite having a theoretical foundation and an implemented tool for the translation of probabilistic Event-B models to probabilistic rewrite theories, it is also necessary to implement it for the existing Event-B tools like Rodin. Projects like the EBPR project [20] aim to enhance the existing Rodin toolset by allowing to import and use externally defined domain theories as Rodin plugins. Hence, in order to contribute to the Rodin and Event-B capabilities, it is necessary to render extensions, such as the one mentioned previously in [18], into practical Rodin plugins.

Chapter 2

Literature Review

The main purpose of this chapter is to present the needed theoretical and technical background to understand how the tool in [19] and the plugin work. Therefore, a short but sufficient definition of various concepts and tools, like the Event-B method, Maude, rewriting logic, VeStA and others, will be given. To guide the reader through the different sections of this chapter and facilitate its reading, the following section dependency is given:
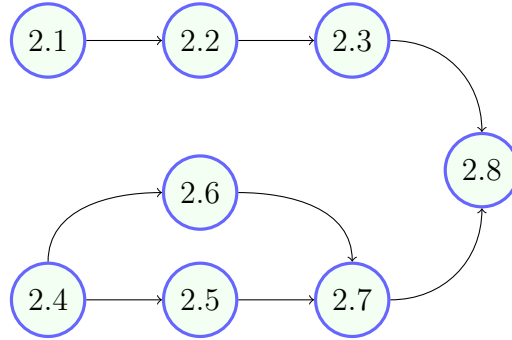

Figure 2.1: Section dependencies

2.1 Event-B

Event-B [4, 21] is a formal method for specifying and verifying properties about systems based on set theory and predicate logic. Specifications in Event-B are referred to as models, and they are semantically modeled by a discrete labeled transition system or LTS [3]. 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 ones. 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_identifier >
    extends
        < context_identifier_list >
    sets
        < set_identifier_list >
    constants
        < constant_identifier_list >
    axioms
        < label >:  < predicate >
        . . .
    theorems
        < label >:  < predicate >
        . . .
    end
```

Figure 2.2: Context structure, taken from [4]

contexts, 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 expressed in figure 2.2. The definition for each one of the elements of the syntax is:

- *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 of a model 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 $T_i(\bar{s}, \bar{c})$.

The general structure of the machine is specified in figure 2.3. Each one of its components correspond to:

- *machine_identifier* is a string that identifies the machine. It must be different from

7

```
< 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 2.3: Machine structure, taken from [4]

all the other components identifiers.

- The "refines" clause contains the identifier of the machine that this machine refines.

- 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.

- 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 predicates 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.

- 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:

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

- 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 $\bar{t} = \{t_1, ..., t_n\}$.

I would change all the "any", "variables", etc with ``\textbf{any}"

8

- 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 $S(\bar{t}, \bar{v})$.

event assignments 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 non-deterministically 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 [11], 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 2.4 and 2.5.

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

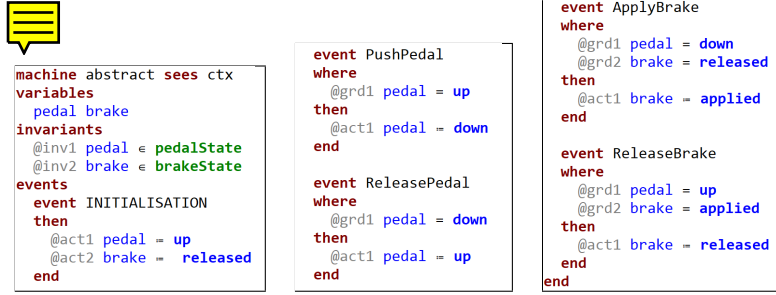Figure 2.4: Context of the brake example

Figure 2.5: Machine of the brake example

The context identifier 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 pedal's state 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 its 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 transition possibilities with multiple parameters that suffice all the model constraints. The way Event-B resolves this multiple choice dilemma is with *non-deterministic choice* and there are types of it:

- **Choice of enabled events:** When multiple events are enabled for execution, i.e. when multiple events satisfy their event guards, 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 $x :\mid Q(\bar{t}, \bar{v}, x, x')$: states that the variable x takes the values .... x' that satisfies the predicate $Q(\bar{t}, \bar{v}, x, x')$, is assigned. When there are multiple x' values that satisfy the predicate, then the new value of x is chosen non-deterministically.

  - Enumerated assignment $x :\in \{E_1(\bar{t}, \bar{v})...E_n(\bar{t}, \bar{v})\}$: states that to the variable x, one of the multiple expressions $E_i(\bar{t}, \bar{v})$ in the set is assigned to it. The selection of which expression will be assigned, is done non-deterministically.

2.3 Probabilistic Event-B

Based on the types of non-determinism present in Event-B, probabilistic Event-B [11] introduces 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 represents 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. This means that if there are n enabled events, then the probability of choosing event e_i is $P(e_i) = \frac{W(e_i)}{\sum_{j=1}^{n} W(e_j)}$

- **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 is $P(t_i) = \frac{1}{n}$ 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 a 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:

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

Where $W(\bar{v})$ is an expression over the variables that determines the weight of the event, $\bar{t}$ is the set of parameters 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, let's consider an extension of the previously explained brake model. This new example, presented also in [11], 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 is displayed in figure 2.6. This new model incorporates all of the previously defined constraints in the following way:

- Constraints **R1** and **R2** are modeled in the probabilistic event *PushPedal*, in which an enumerated probabilistic assignment is used to assign the value of variable *pedal*. This assignment *pedal := down* @9/10 $\oplus$ *up* @1/10 states that when the event *PushPedal* is executed, the probability of changing its value to *down* is 90% and the probability of remaining with the value *up* is 10%.

- For **R4**, a new constant MAX_WEAR s.t. $MAX\_WEAR \in \mathbb{N}$ and $MAX\_WEAR > 1$, is introduced to the context of the model. This constant determines the maximum number of times the break can be applied. In addition, the variable $wear \in \mathbb{N}$ tracks the number of times the brake has been used. To make sure that the number of times the break has been applied doesn't exceeds the maximum wear, i.e. $wear < MAX\_WEAR$, the weights of probabilistic events *ApplyBrake* and *ReleaseBrake* are modeled by the expression $MAX\_WEAR - wear$. Therefore, when $MAX\_WEAR = wear$, the weight of both events will be 0. This will make their probability of execution also 0, based on how probabilistic choice of enabled events is calculated.

- To model **R3**, a new event *ApplyBrakeFailure* is introduced. When executed, this event simulates ake failure, by leaving the brake in state *released,* when the pedal is *down*.

- Finally, the constraint **R5** is defined by the weights of event *ApplyBrake* and *ApplyBrakeFailure*. As mentioned before, the weight of event *ApplyBrake* is modeled with the expression $MAX\_WEAR - wear$, and the expression for the weight of event *ApplyBrakeFailure* is *wear*. Thus, when the value of variable *wear* increases, then the probability of executing event *ApplyBrake* decreases and the probability of *ApplyBrakeFailure* increases.

Figure 2.6: Probabilistic machine of the brake example, taken from [11]

2.4   Maude

Maude [22, 23, 24] is a high performance declarative language, that allows the specification of programs or systems, and their formal verification. Maude's 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 a 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 [23]:

```
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. In this case, the operation is defined inductively using 2 equations. Using this module, it is possible to compute the value for addition for two natural numbers using the command `red`. For example, if the command `red s(s(s(0))) + s(s(0))` is used (the command 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) .
```

14

```
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, in 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* [24, 23], 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* [24, 23] and behaves in the following way:

- With the equations E of (Σ, E), *term rewriting rules* are defined as $\vec{E} = \{u \rightarrow v \mid (u = v) \in E\}$.

- A term t, which are expressions formed using the syntax in Σ, is rewritten to t' in one step $t \rightarrow_{\vec{E}} t'$ if and only if, the following conditions are sufficed:

  - there is a subterm w in t, expressed as $t[w]$.

  - there is a rule $(u \rightarrow v) \in \vec{E}$ and a substitution 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))

- $\theta = \{\texttt{N} \mapsto \texttt{s(s(s(0)))}, \texttt{M} \mapsto \texttt{0}\}$

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

- $w' = \texttt{s(N+M)}\,\theta = \texttt{s(s(s(s(0)))} \ + \ \texttt{0)}$

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

the resulting term rewriting is $t \rightarrow_{\vec{E}} t'$. Aside from building <u>programs</u> 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* [23, 25] $\mathscr{R} = (\Sigma, E, L, R)$ 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 of 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 .
```

<u>to</u> exemplify this, let's 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`.
They contain 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 it can only drop a person when the
  number of people inside the bus is 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 rule 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 the system, Maude also

17

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 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 a 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 in Maude

Object-based programming in Maude [22, 25, 23] 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 object's identifier, C is a class identifier, $a_1...a_n$ 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 [22] 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 corre-

sponds 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 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 [16], 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, both 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 $r \in R$. To exemplify this new notion, let's consider the PMaude module, presented in [16]:

```
pmod EXPONENTIAL-CLOCK is
  --- import positive real number module
  protecting POSREAL .

  --- imports PMaude module that defines
  --- EXPONENTIAL, BERNOULLI, GAMMA, etc.
  protecting PMAUDE .

  --- declare a sort Clock
```

```maude
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 - C/1000)
                    else
                            broken(T,C - C/1000)
                    fi
            with probability B:=BERNOULLI(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's battery. 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 current's clock's charge. If `B` is false, then the clock will break and change to the state `broken(T,C - 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 an 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 [16] 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 - C/1000)
              fi .

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

2.7 VeStA, PVeStA, MultiVeStA, QuaTEx and MultiQuaTEx

VeStA [26] is a tool (implemented in Java) for statistical analysis of probabilistic systems. It supports statistical model checking and statistical evaluation of expected values of temporal expressions. These expressions can be constructed using the query language of *Quantitative Temporal Expression* or QuaTEx [16], and analyzed by VeStA using *Monte Carlo simulations*. The general process to use VeStA and analyze the properties of a probabilistic model, is the following:

1. Create a probabilistic system in the supported modeling languages, e.g. probabilistic rewrite theories specified in PMaude.

2. Define the model properties that are going to be analyzed using the supported temporal expressions, e.g. QuaTEx.

3. Run Monte Carlos simulations using VeStA, over the model and the defined properties, specifying the simulation parameters.

4. Get the expected value of the temporal expressions specified in step 2.

The syntax of a QuaTEx expression is defined in figure 2.7. The reader can find an in depth explanation of this syntax and how QuaTEx queries evaluate in [16, 27], but for now we will define a QuaTEx expression as the expected value of a temporal logic predicate over state observations that returns a real number. For example the QuaTEx expression `eval E[PExp]` returns the expected value of the real number returned by the temporal logic predicate $PExp$ (or also called *path expression* in the syntax of QuaTEx).

$$
\begin{array}{lll}
Q & ::= & DS \; \textbf{eval } \mathbf{E}[PExp]; \\
DS & ::= & \textbf{set of } Defn \\
Defn & ::= & N(x_1, \ldots, x_m) \; = \; PExp; \\
SExp & ::= & c \; | \; \mathbf{rval}(i) \; | \; F(SExp_1, \ldots, SExp_k) \; | \; x_j \\
PExp & ::= & SExp \; | \; \#N(SExp_1, \ldots, SExp_n) \; | \\
& & \textbf{if } SExp \textbf{ then } PExp_1 \textbf{ else } PExp_2 \textbf{ fi}
\end{array}
$$

Figure 2.7: QuaTEx syntax, taken from [27]

Using the previous definition of QuaTEx queries, let's suppose that we want to find the expected value of a QuaTEx query `eval E[PExp]` using VeStA (we will refer to the expected value of the QuaTEx query obtained by VeStA as $\bar{x}$, and to the real expected value

24

as x). To obtain $\bar{x}$, VeStA uses 2 user-defined parameters α and δ. With these parameters, VeStA runs n simulations, until n is large enough to obtain a *confidence interval* (CI) with probability $(1 - \alpha) * 100\%$ bounded by δ, i.e an interval $[\bar{x} - \frac{\delta}{2}, \bar{x} + \frac{\delta}{2}]$ where the probability that $x \in [\bar{x} - \frac{\delta}{2}, \bar{x} + \frac{\delta}{2}]$ is $(1 - \alpha) * 100\%$. In other words, if VeStA calculates the expected value of a QuaTEx query as $\bar{x}$, then the probability that the real expected value of the query (represented as x) is in the interval $[\bar{x} - \frac{\delta}{2}, \bar{x} + \frac{\delta}{2}]$, is $(1 - \alpha) * 100\%$.

VeStA and Quatex have also seen many upgrades during time. For example, the authors in [17] present an extension of VeStA called PVeStA. This tool allows to run parallelized algorithms for statistical model checking using a client-server architecture. Furthermore [27] presents an extension of both VeStA and PVeStA called MultiVeStA, that extends the QuaTEx language to MultiQuaTEx, by allowing to query more state measures at a time. This improves the usability and the performance of the language. The extension also integrates existing discrete event simulators in addition to the originally supported ones, and improves the presentation of results.

To illustrate how statistical analysis of probabilistic rewrite theories specified in Pmaude works, we'll use the MultiVeStA tool to run simulations over the clock model specified in the previous section and verify properties over it. To do this, the steps to modify and run the clock example are:

1. Define the PMaude model's states as configurations using object-oriented programming in Maude and define the system transitions as rewriting rules between configurations.

2. Include the necessary elements of MultiVeStA inside the model's definition.

3. Define an analysis module to define the model observations used by the MultiQuaTEx queries.

4. Define the MultiQuaTEx queries that represent the model's properties that want to be checked.

5. Specify the simulation parameters.

6. Run the simulations using the the probabilistic model, the MultiQuaTEx queries and the simulation parameters.

For the first step, we will create a class named `Clock` defined as:

```
< Oid : Clock | time: Nat, battery: Float, state: State > .
```

where the sort `State` is defined by two constants `working` and `broken`. The initial state of the system is then defined as:

```
< myClock : Clock  | time: 0, battery: 1000.0, state: working >
```

And the rewrite rule that simulates the clock's ticks is:

```
rl [clockTick] :
    < cl : Clock  | time: T, battery: C, state: working >
  =>
    if sampleBernoulli(C / 1000.0) then
      < cl : Clock  | time: s(T), battery: (C - (C / 1000.0)),
        state: working >
    else
      < cl : Clock  | time: T, battery: C, state: broken >
    fi .
```

In this model, instead of having the exponential function to determine the time increment, it will be fixed to 1. Afterwards, it is necessary to implement the structures required by MultiVeStA to run the simulations. This structures control the simulation's time and schedules the following rules to be applied. Then, the state observations can be defined. These observations correspond to the rval(*i*) predicate defined in [16, 27], and for this example three rval predicates are ~~going to be~~ defined:

- s.rval("eTime"): returns the time value of the clock in state *s*, as a float number.

- s.rval("eBattery"): returns the battery level of the clock in state *s*.

- s.rval("isBrk"): returns 1.0 if the clock is broken in state *s* or 0.0 if the clock is working in state *s*.

Using these predicates, two MultiQuaTEx formulas are defined:

```
PTime() = if ( s.rval("isBrk") == 1.0 )
          then s.rval("eTime") else # PTime() fi ;
eval E[ PTime() ] ;


#-------------------------------------------------------------

PBattery() = if ( s.rval("isBrk") == 1.0 )
             then s.rval("eBattery") else # PBattery() fi ;
eval E[ PBattery() ] ;
```

This notation using pareth

The first formula states that the if the clock is broken (i.e. s.rval("isBrk") == 1.0) then return the current time of the clock (i.e. s.rval("eTime")). If not, the query recursively calls itself in the following system state (i.e. # PTime()). The other query has the same behavior as the previous one, but instead of s.rval("eTime") is s.rval("eBattery"). Therefore, the answer returned by the first query corresponds to the expected value of the clock's time when it breaks, and the answer returned by the second query is the expected value of the clock's battery when it breaks. After the simulations were executed with the clock probabilistic model, both MultiQuaTEx formulas

26

and parameters $\alpha = 0.05$ and $\delta = 1$, the obtained results for both queries were:

Property	ObtainedValue	Variance	CI
PTime()	39.7909482758621	422.729500809641	0.999968586222357

Property	ObtainedValue	Variance	CI
PBattery()	961.13204082891	387.863387550541	0.999651593005253

This means that the expected value of the time when the clock breaks has a probability of $(1 - \alpha) * 100\% = 95\%$ to be inside the interval $[\frac{\delta}{2} - 39.79, \frac{\delta}{2} + 39.79] = [39.29, 40.29]$, and the expected value of the battery level when the clock breaks has a probability of $(1-\alpha)*100\% = 95\%$ to be inside the interval $[\frac{\delta}{2}-961.13, \frac{\delta}{2}+961.13] = [960.63, 961.63]$.

2.8 A Rewriting Logic Semantics for Probabilistic Event-B

The rewriting logic approach to probabilistic Event-B [18], introduces a method for translating probabilistic Event-B models to PMaude models that are executable in PVeStA. Formally speaking, it is a map $[[\cdot]] : \mathcal{M} \to \mathcal{R}_{\mathcal{M}}$ from a probabilistic event machine to a probabilistic rewrite theory. There are two main steps for this transformation:

1. The first one is to specify a rewrite theory $\mathcal{R}$ that defines the types, sets, constants of the Event-B's context, and the infrastructure needed to encode variables and events of a probabilistic Event-B machine.

2. After that, $\mathcal{R}$ is extended with equations and rewrite rules that correspond to the events in the machine $\mathcal{M}$.

This rewrite theory $\mathcal{R}$, is constructed with two main components: A Maude specification of an Event-B context and an Event-B machine. The Maude version of the context is:

```
mod EBCONTEXT is
  pr CONFIGURATION . pr EB-TYPE  . pr MAP{Qid, EBSet} . pr MAP{Qid, EBType} .
  --- Names for contexts
  subsort Qid < Oid .
  --- Class for contexts
  op Context  : -> Cid .
  --- Context's user-defined sets
  op sets :_      : Map{Qid , EBSet}  -> Attribute .
  --- Context's constants
  op constants :_ : Map{Qid , EBType} -> Attribute .
  --- Building a context
```

```
  op init-context : Qid -> Object .
  var Q : Qid .
  eq init-context(Q) = < Q : Context | sets : init-sets,
                          constants : init-constants  > .
  --- Operators to be instantiated by the implementation of the context
  op init-sets : ->  Map{Qid , EBSet} . --- Building deferred sets
  op init-constants : ->  Map{Qid , EBSet} . --- Initializing constants
endm
```

~~Where~~ EBtype encodes the values of the model constants, by wrapping all the possible
types a constant can have and EBSet contains the definition of finite ~~sets~~ This Maude
version of the context is based on the CONFIGURATION theory, previously explained in sec-
tion 2.5. A context is represented as an object of the class Context, with two attributes
that are sets and constants. The attribute sets contains all the sets identifiers
paired with their respective values, and constants contains all the constant identifiers
paired with their respective values. The context attributes' values are initialized using the
init-context, init-sets and init-constants operators. In case of the Event-
B machine, it is represented with the following Maude specification:

```
mod EBMACHINE is
  inc EBCONTEXT  . --- Context specification.
  op Machine  : -> Cid . --- Class for machines
  op variables :_ : Map{Qid , EBType} -> Attribute . --- Machine's variables
  --- Instantiated by the machine at hand
  op init-variables : ->  Map{Qid , EBType} .
  op init-machine : Qid Qid -> Object . --- Initial state of the machine
  vars QM QC : Qid .
  --- Configuration defining the machine QM and its context QC
  eq init-machine(QC, QM)  =
                init-context(QC)                            --- Context
                < QM : Machine | variables : init-variables > --- Variables
                < events : Events | state: init-events >.    --- Events
[...] --- event specification explained below
```

Like the context, the machine is also represented using object-oriented programming in
Maude. A machine is an object of the class Machine with only one attribute variables,
that correspond to all the variable identifiers paired with their respective value. The pre-
viously explained definitions for the Maude version of context and machine, correspond
to the rewrite theory $\mathcal{R}$, which can be extended in two steps, to obtain $\mathcal{R}_\mathcal{M}$: adding the
initialization event, that will fill the initial configuration of the model with the initial values
of all the variables, and the second step is to add the different equations and rewrite rules
that encode the probabilistic Event-B events. The former is represented as an equation
that initializes init-variables, of the form:

```
    eq init-variables = ('v1 |-> val1, ... , ('vn    |-> valn)
```

Where every $v_1...v_n$ are the model's variables and $val_1..val_n$ are their corresponding values. The latter is achieved by:

1. A deterministic mechanism that determines if an event is enabled or not, given the current state of the machine

2. A probabilistic rule that chooses the next event to be executed, according to the weights of the enabled events

3. For each event, a rule that chooses probabilistically the value of the event parameters and probabilistic assignments. After that, the state of the machine is updated accordingly. This rule corresponds to an observable transition of the machine $\mathcal{M}$.

Events' states (1) The possible states for an event are defined as:

```
sort EvState . --- States of events
ops blocked unknown execute : -> EvState .
op enable : NzNat -> EvState . --- Enable with a given weight w > 0
```

where `blocked` means that the event can't be executed in the current machine configuration, `unknown` means that the event's state hasn't been computed, `execute` implies that the event will be executed in the following transition and `enable` states that an event is enabled for execution in the given machine configuration, with an specific weight. The definition for an event is:

```
sort Event LEvent . --- Events and list of events
op ev : Qid EvState -> Event . --- ID of the event and its state
op state:_ : LEvent -> Attribute . --- Attribute for objects of class Events
op init-events : -> LEvent . --- To be instantiated by the machine at hand
```

Each event has the form $ev(e_i, st_i)$, where e_i is the event identifier and st_i is the state of the event. When the machine is initialized, every event e_i (represented as the event list `init-events`) starts as $ev(e_i, unknown)$. In the following machine configurations, the events' states are updated deterministically with an equation from `unknown` to `enabled` or `blocked`, depending on the guards of the event and the current state of the machine.

Determine next event to execute (2) In order to probabilistically select the next event that will be executed, using the event weights of the enabled events, the following conditional rule is used:

```
crl [next-event] :   < events : Events | state: LE > =>
                     < events : Events | ev(pick(LEA,
                       rand(max-value(LEA))), execute) >
if     all-ready(LE) --- All e in LE is in state either blocked or enabled
   /\  one-firable(LE) --- At least one event is enabled
   /\  LE' := filter(LE) --- Extract the enabled events
```

29

```
/\  LEA := accumulate(LE') . --- Accumulate the weights
```

this rule states that if there exists one or more enabled events (no deadlock), then one of the enabled events is picked, following a uniform distribution on the interval $[0, W)$ where W is the sum of all the weights of enabled events. The state of the selected event will the be changed to `execute`.

Actions in events (3) After an event is chosen for execution, the variable values need to change according to the event actions. Both deterministic an probabilistic assignments can be done by simply changing the state configuration using a standard rewrite rule, with the slight difference that for probabilistic assignments or assignments that involve a parameter, a function that simulates a uniform distribution (like the one used for selecting events) will be used to select the new variable value or the value of the parameter in the assignment.

Chapter 3

Methodology

Chapter 4

Results

Chapter 5

Discussion

Chapter 6

Conclusions

Bibliography

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys*, vol. 41, 10 2009.

[2] J.-R. Abrial, *The B-book : assigning programs to meanings.* Cambridge University Press, 1996.

[3] D. Bert and F. Cave, "Construction of finite labelled transition systems from b abstract systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1945 LNCS, pp. 235–254, 2000.

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

[5] Event-B.org, "Event-b and the Rodin platform." `http://www.event-b.org/`.

[6] M. Comptier, D. Deharbe, J. M. Perez, L. Mussat, T. Pierre, and D. Sabatier, "Safety analysis of a cbtc system: A rigorous approach with event-b," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10598 LNCS, pp. 148–159, 2017.

[7] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Supporting reuse in event b development: Modularisation approach," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5977 LNCS, pp. 174–188, 2010.

[8] R. Banach, M. Butler, S. Qin, N. Verma, and H. Zhu, "Core hybrid event-b i: Single hybrid event-b machines," *Science of Computer Programming*, vol. 105, pp. 92–123, 7 2015.

[9] P. Boström and M. Waldén, "An extension of event b for developing grid systems," *Lecture Notes in Computer Science*, vol. 3455, pp. 142–161, 2005.

[10] C. Morgan, T. S. Hoang, and J. R. Abrial, "The challenge of probabilistic event b - extended abstract," *Lecture Notes in Computer Science*, vol. 3455, pp. 162–171, 2005.

[11] 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.

[12] S. Hallerstede and T. S. Hoang, "Qualitative probabilistic modelling in event-b," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4591 LNCS, pp. 293–312, 2007.

[13] A. Tarasyuk, E. Troubitsyna, and L. Laibinis, "Towards probabilistic modelling in event-b," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6396 LNCS, pp. 275–289, 2010.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*. Springer Berlin Heidelberg, 2007.

[15] R. Bruni and J. Meseguer, "Semantic foundations for generalized rewrite theories," *Theoretical Computer Science*, vol. 360, pp. 386–414, 2006.

[16] 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.

[17] 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.

[18] C. Olarte, D. Osorio, and C. Rocha, "A rewriting logic semantics and statistical analysis for probabilistic event-b,"

[19] C. Olarte, D. Osorio, and C. Rocha, "The tool." `https://github.com/carlosolarte/EventB2Maude`, 2022.

[20] A. N. de la Researche, "The EBRP project." `https://www.irit.fr/EBRP`.

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

[22] 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.

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

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

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

[26] K. Sen, M. Viswanathan, and G. Agha, "Vesta: A statistical model-checker and analyzer for probabilistic systems," vol. 2005, pp. 251 – 252, 10 2005.

[27] A. Vandin and S. Sebastio, "Multivesta: Statistical model checking for discrete event simulators," 01 2014.