EventB2Maude: Probabilistic Modeling and Statistical Model Checking for Event-B with Probabilistic Rewrite Theories and MultiVeStA

Daniel F. Osorio-Valencia

April 11, 2023

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 the 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].

In order to cope with new system behaviors not considered in the original Event-B methodology, it has been extended throughout its history 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 probabilistic Event-B [9], which aims to introduce probabilistic behavior to Event-B models. The main motivation for this extension is that 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 [10]. 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 [10]. Other extensions try to introduce probabilistic choices by using qualitative probabilistic assignments [11] or quantitative probabilistic choice [12], 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 [13] is a modeling language used to define formal models of distributed systems, based on rewriting logic [14]. A probabilistic extension of Maude, named PMaude [15], permits probabilistic modeling of concurrent systems. Paired with this extension and the query language known as QuaTEx [15], statistical model checking tools like PVeStA [16] allow to verify 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 [17] 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 $\mathscr{R}_{\mathscr{M}}$, 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 named EventB2Maude [18], which takes as input Event-B models and turns them into PMaude models that are executable using PVeStA.

Even though PVeStA can be used over the generated models by the EventB2Maude tool to do statistical model checking, the lack of documentation about PVeStA, makes the task of representing and proving complicated system properties of PMaude models harder. For this reason, other tools like MultiVeStA [19], which is an extension of PVeStA, can be considered. MultiVeStA is also a statistical model checking tool, that allows to represent with great ease properties in the MultiQuaTEx query language [19] (an extension of the QuaTEx language), with the advantage of having more documentation.

Therefore, the present work has 2 motivations. The first one, is facilitating the model checking task, using the EventB2Maude tool, by extending it so the generated PMaude models can be verified using MultiVeStA . The second one, is to complement the available documentation of MultiVeStA, with a simple guide that specifies the required steps to adapt probabilistic rewrite theories specified in Maude, to work with MultiVeStA. It is expected that this work contributes to future projects that aim to do statistical model checking of PMaude specifications with MultiVeStA, or other projects that include the EventB2Maude tool.

Chapter 2

Preliminaries

The main purpose of this chapter is to present the needed theoretical and technical background to understand how the encoding from probabilistic Event-B models to probabilistic rewrite theories, presented in [17], works. Moreover, it will also allow to understand the proposed adaptations to the tool in [18], to perform simulations with MultiVeStA over the encoded Event-B models. Therefore, a short but sufficient definition of various concepts and tools, will be given. To guide the reader through the different sections of this chapter and facilitate its reading, the following section dependency is given:

$$\begin{tikzcd}
2.1 \arrow[r] & 2.2 \arrow[r] & 2.3 \\
 & 2.6 \arrow[dr, bend left] & \\
2.4 \arrow[ur, bend left] \arrow[r] & 2.5 \arrow[r] & 2.7
\end{tikzcd}$$

Figure 2.1: Section dependencies

2.1 Event-B

Event-B [4, 20] 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,

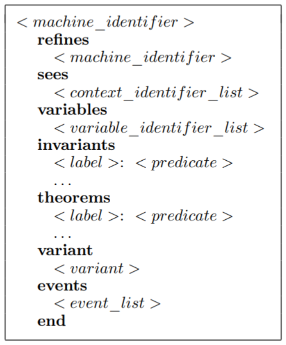
```
< context_identifier >
  extends
    < context_identifier_list >
  sets
    < set_identifier_list >
  constants
    < constant_identifier_list >
  axioms
    < label >:  < predicate >
    ...
  theorems
    < label >:  < predicate >
    ...
  end
```

Figure 2.2: Context structure, taken from [4]

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 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 depicted 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 of 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* >
  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]

- *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.
- 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:

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

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

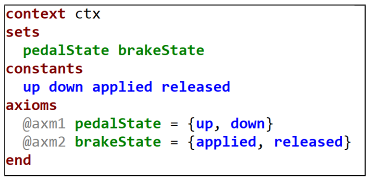
- The "**where**" clause contains the guards of the event. Guards are represented as a conjunction of logic predicates $G(\bar{t}, \bar{v})$ over the parameters of the event and variables of the system. These guards specify the conditions that must hold for the event to be enabled.
- The "**then**" clause contains the list of actions of the event. Actions define variable assignments that change the variable values (and thus the state of the system) 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 $x := E(\bar{t}, \bar{v})$ states that if the event is executed, then the value of variable x in the next model state will be $E(\bar{t}, \bar{v})$, 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 $x :\mid Q(\bar{t}, \bar{v}, x, x')$ denoted as predicate assignment. It assigns to variable x the value x', s.t. x' satisfies the predicate $Q(\bar{t}, \bar{v}, x, x')$.

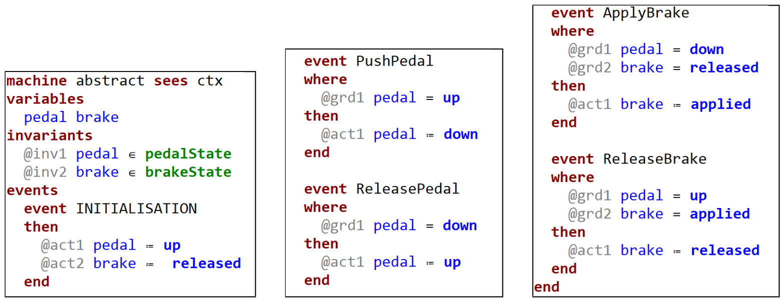
The set of events of a model is called as $Evts$. This set includes the $INITIALISATION$ event or $Init$, which assigns the initial value to all the variables of the machine and creates the initial state of the model. The initial state must be deterministic in all Event-B specifications. A whole Event-B 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 [10], will be used.

Example 2.1.1 A brake system (derived from the example presented in [10]) 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



```
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 2.5: Machine of the brake example

The context identifier for this model is *ctxBrakeSystem*, 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 $axm1$ and $axm2$. Therefore, the axioms of the brake model are defined by $A(\bar{s}, \bar{c}) = axm1 \wedge axm2$. In particular, this context doesn't extend other components or uses theorems.

For the model's machine, the machine identifier is *abstractBrakeSystem* and it "sees" the sets, constants and axioms in *ctxBrakeSystem*. 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 $inv1$ and $inv2$ 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 $I(\bar{v}) = inv1 \wedge inv2$. Each one of the events in the model represent a state transition and a system action:

- The *INITIALISATION* determines the initial state of the system. 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 $G(\bar{t}, \bar{v}) = grd1$ states that the pedal must be *up*, to execute the action $S(\bar{t}, \bar{v}) = act1$. This action assigns the value *down* to the *pedal* variable. Conversely, the *ReleasePedal* event 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 $G(\bar{t}, \bar{v}) = grd1 \wedge grd2$ states that the pedal must be *down* and the brake *released*, to execute the action $S(\bar{t}, \bar{v}) = act1$. 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 of having multiple enabled transitions with also many possible parameter valuations that satisfy the guards of the event. The way Event-B resolves this multiple choice dilemma is with *non-deterministic choice* and there are three 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 value x', that satisfies the predicate $Q(\bar{t}, \bar{v}, x, x')$. 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 the variable x takes the value of one of the multiple expressions $E_i(\bar{t}, \bar{v})$ in the set. The selection of which expression will be assigned, is done non-deterministically.

2.3 Probabilistic Event-B

Based on the three different forms of non-determinism explained above, probabilistic Event-B [10] 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:

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

Where $W(\bar{v})$ is an expression over the variables that determines the weight of the event, $\bar{t}$ 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 $Init$ is the initialization event ($Init$ must be a deterministic event). To exemplify probabilistic Event-B, let's consider an extension of the previously explained brake model.

Example 2.3.1 This new example, presented also in [10], 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.

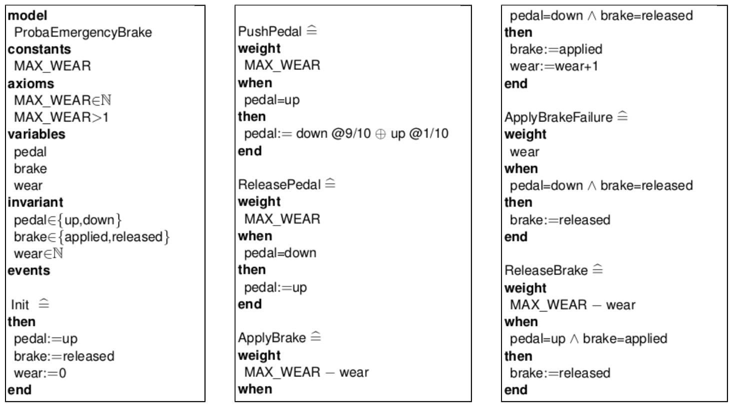
R5. Brake wear: due to brake wear, the risk of brake failure increases each time the brake is applied.

The resulting probabilistic model is depicted 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 a brake 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.



```
model
 ProbaEmergencyBrake
constants
 MAX_WEAR
axioms
 MAX_WEAR∈ℕ
 MAX_WEAR>1
variables
 pedal
 brake
 wear
invariant
 pedal∈{up,down}
 brake∈{applied,released}
 wear∈ℕ
events

 Init ≘
then
 pedal:=up
 brake:=released
 wear:=0
end

PushPedal ≘
weight
 MAX_WEAR
when
 pedal=up
then
 pedal:= down @9/10 ⊕ up @1/10
end

ReleasePedal ≘
weight
 MAX_WEAR
when
 pedal=down
then
 pedal:=up
end

ApplyBrake ≘
weight
 MAX_WEAR − wear
when
 pedal=down ∧ brake=released
then
 brake:=applied
 wear:=wear+1
end

ApplyBrakeFailure ≘
weight
 wear
when
 pedal=down ∧ brake=released
then
 brake:=released
end

ReleaseBrake ≘
weight
 MAX_WEAR − wear
when
 pedal=up ∧ brake=applied
then
 brake:=released
end
```

Figure 2.6: Probabilistic machine of the brake example, taken from [10]

2.4 Maude

Maude [21, 22, 23] is a high performance declarative language, that allows the specification of programs or systems, and their formal verification. Maude's specifications 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 specification. The body of the module contains *sorts* (written in Maude as `sorts`), where each sort correspond to a specific data type. 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 specification's 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 illustrate how a Maude specification is contructed, the following code corresponds to a functional module that defines the natural numbers and the addition operation, borrowed from [22, 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) .
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 *equational theories* [23, 22], 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* [23, 22] 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 sufficed:
  - there is a subterm w in t, expressed as $t[w]$.

– there is a rule $(u \to v) \in \overrightarrow{E}$ and a substitution θ s.t. : $w = u\theta$, $w' = v\theta$, $t' = t[w'] = t[v\theta]$.

As an example, taken from [23], 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 $t \to_{\overrightarrow{E}} t'$. Aside from building specifications 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* [22, 24] $\mathscr{R} = (\Sigma, E, L, R)$ 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 $l : t \to t'$, and conditional labeled rewrite rules of the form $l : t \to t'$ 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, let's consider the following simple model of a bus.

Example 2.4.1 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, four 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 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 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 [21, 24, 22] 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 [21] can be examined.

Example 2.5.1 The `BANK-ACCOUNT` system module is defined as:

```
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 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 [15], 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 $r \in R$.

Example 2.6.1 To exemplify this new notion, let's consider the PMaude module, presented in [15]:

```
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
  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 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 - `$\frac{C}{1000}$`)`. 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 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 [15] 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 VeStA, PVeStA, MultiVeStA, QuaTEx and MultiQuaTEx

VeStA [25] 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 [15], 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:

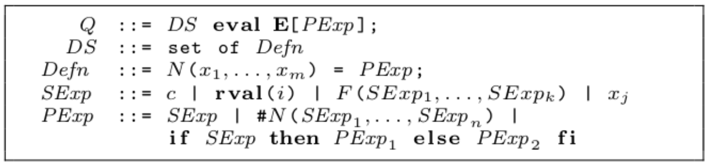
1. Create a probabilistic system in the supported modeling languages, e.g. probabilistic rewrite theories specified in PMaude.

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 [15, 19], 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).



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

Figure 2.7: QuaTEx syntax, taken from [19]

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 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 [16] present an extension of VeStA called PVeStA. This tool allows to run parallelized algorithms for statistical model checking using a client-server architecture. Furthermore [19] 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. In Chapter 5, a detailed explanation on how to use MultiVeStA with PMaude specifications, will be given.

To illustrate how statistical analysis of probabilistic rewrite theories specified in Pmaude works, the MultiVeStA tool will be used 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. Implement a scheduler that chooses deterministically the event that is going to be executed.

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

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

5. Define the MultiQuaTEx queries that represent the model's properties that want to

be checked.

6. Specify the simulation parameters.
7. 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 [15, 19], and for this example three `rval` predicates are defined as:

- `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() ] ;
```

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. Once the simulations are executed with the clock probabilistic model, both MultiQuaTEx formulas 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 $[\frac{\delta}{2}-961.13, \frac{\delta}{2}+961.13] = [960.63, 961.63]$.

Chapter 3

A Rewriting Logic Semantics for Probabilistic Event-B

The rewriting logic approach to probabilistic Event-B [17], introduces a method for translating probabilistic Event-B models to PMaude models that are executable in PVeStA. Formally speaking, it is a map $\llbracket \cdot \rrbracket : (\mathscr{C}, \mathscr{M}) \to \mathscr{R}_{\mathscr{M}}$ from a probabilistic event model to a rewrite theory. There are two main steps for this transformation:

1. Specify a rewrite theory $\mathscr{R}$ to encode the static parts of the model, i.e. the context and the initialization of the machine.
2. Then, $\mathscr{R}$ is extended with equations and rewrite rules that correspond to the events in the machine $\mathscr{M}$.

The rewrite theory $\mathscr{R}$ works as a framework that allows to represent each one of the Event-B's elements inside Maude. To explore the implementation of it, the reader can refer to [18], where the rewrite theory is specified with multiple Maude system modules in the folder named as `m-theory`. For the sake of simplicity, a general definition of $\mathscr{R}$ will be given.

$\mathscr{R}$ consists of four Maude system modules: `SAMPLER`, `EB-CORE`, `EBCONTEXT`, and `EBMACHINE`. `SAMPLER` contains the definition of functions for probabilistic sampling. `EB-CORE` functions as a prelude, that contains definitions for the representation of basic Event-B constructs such as elements, sets, relations, and operations on them. `EBCONTEXT` defines the structure to encode Event-B's contexts in Maude and `EBMACHINE` does the same for Event-B's machines. The way these modules interact, is represented in Figure 3.1.

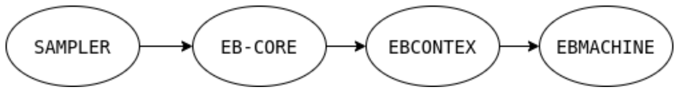


Figure 3.1: Interaction between system modules

The figure shows that the `EBMACHINE` module includes the `EBCONTEXT` module, `EBCONTEXT` includes `EB-CORE`, and `EB-CORE` includes `SAMPLER`. When the translation of an Event-B model is done, the resulting specification includes the equations and rewrite rules necessary to extend $\mathscr{R}$ into $\mathscr{R}_{\mathscr{M}}$. Therefore, the system module that represents the translated Event-B model, must include the `EBMACHINE` module, which also includes all the other mentioned modules. In the end, the interaction between these modules, represent the rewrite theory $\mathscr{R}_{\mathscr{M}}$, as seen in the Figure 3.2. Using this framework for representing Event-B models in

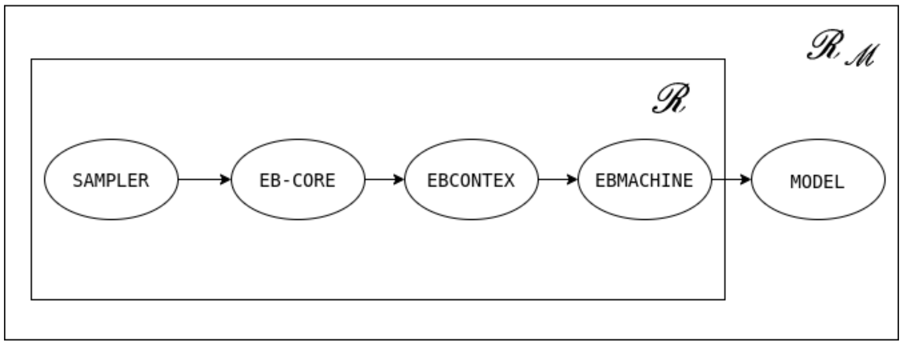


Figure 3.2: Interaction between $\mathscr{R}$ and the translated model

Maude, it is possible to define the complete encoding $[\![\cdot]\!]$. To guide the reader through the complete encoding, the example of the probabilistic brake system in section 2.3, Figure 2.6, will be used. For the following sections, a *map* will be defined as a set of pairs of the form $(x_1 \mapsto y_1, x_2 \mapsto y_2, ..., x_n \mapsto y_n)$

3.1 Encoding Contexts

Let $\mathscr{C}$ be a context. The corresponding encoding of $\mathscr{C}$ in $\mathscr{R}$ corresponds to the term:

$$\langle\ [\![\mathscr{C}]\!]_{id} : Context \mid sets : [\![\mathscr{C}]\!]_{sets},\ constants : [\![\mathscr{C}]\!]_{constants}\ \rangle$$

Where $[\![\mathscr{C}]\!]_{id}$ encodes the context's identifier, $[\![\mathscr{C}]\!]_{sets}$ encodes the context's deferred sets, and $[\![\mathscr{C}]\!]_{constants}$ encodes the context's constants. The resulting term has sort `CONFIGURATION`, and represents an object of the class `Context`, with attributes `sets` and `constants`. For the probabilistic brake system, the resulting term would be the following:

```
<'ctxBrakeSystem: Context |
                  sets: ('SBRAKE |-> (elt("applied"), elt("released")),
                         'SPEDAL |-> (elt("down"), elt("up"))),
                  constants: ('MAXWEAR |-> val(elt(10))) >
```

- $[\![\mathscr{C}]\!]_{id}$ returns the identifier of the context as a a *quoted identifier* [21], which is a predefined string in Maude that is used to identify objects. In the brake model, $[\![\mathscr{C}]\!]_{id} =$`'ctxBrakeSystem`

- $[\![\mathscr{C}]\!]_{constants}$ returns a map of quoted identifiers, that represent the constants' names, with a term of the sort `EBType`, that represents the values of the constants. This sort functions as a wrapper for all the possible types in a Event-B model, i.e. sets, relations and basic data types. These basic data types are defined with the sort `EBElt`, that also functions as a wrapper for integers, boolean values and identifiers (represented as strings). In the brake model, $[\![\mathscr{C}]\!]_{constants} =$ `('MAXWEAR |-> val(elt(10)))`.
- $[\![\mathscr{C}]\!]_{sets}$ returns a map of quoted identifiers, that represent the deferred sets' names, with a set of sort `EBSet`, that represent the possible values of the deferred set. The sort `EBSet` is defined as a Maude set, where the elements of the set are of sort `EBElt`. For example, in the brake model the deferred set that contains the possible states for the *pedal* variable, is represented by the pair `'SPEDAL |-> (elt("down"), elt("up")))`.

3.2 Machine Initialization in Maude

Let $\mathscr{M}$ be a probabilistic machine. The corresponding encoding of the initialization of $\mathscr{M}$ in $\mathscr{R}$ is the term:

$$\langle\ [\![\mathscr{M}]\!]_{id} : Machine \mid variables : [\![\mathscr{M}]\!]_{initVars}\ \rangle$$

Where $[\![\mathscr{M}]\!]_{id}$ encodes the machine's identifier and $[\![\mathscr{M}]\!]_{initVars}$ encodes the model's variables to their initial values, defined by the *Init* event. The resulting term has sort `CONFIGURATION`, and represents an object of the class `Machine`, with attribute `variables`. For the probabilistic brake system, the resulting term: would be the following:

```
<'BrakeSystem: Machine | variables: ('brake |-> val(elt("released")),
                                     'pedal |-> val(elt("up")),
                                     'wear |-> val(elt(0))) >
```

- $[\![\mathscr{M}]\!]_{id}$ returns the identifier of the machine as a a quoted identifier. In the brake model, $[\![\mathscr{M}]\!]_{id} =$`'BrakeSystem`.
- $[\![\mathscr{M}]\!]_{initVars}$ returns a map of quoted identifiers, that represent the names of the variables, with a term of sort `EBType`, that represent the values of the variables. For example, in the brake model the variable *pedal* and its value in the initial state, are represented by the pair `'pedal |-> val(elt("up"))`.

3.3 Events' States in Maude

In order to simulate the behavior of probabilistic Event-B in Maude, it is necessary to have a deterministic method to choose the next event that is going to be executed during simulation. To do this, it is necessary to define a structure that contains the states of the events in each system configuration. These possible event's states are:

- ***unknown*:** the event's guards have not been evaluated.
- ***enabled(n)*:** the event's guard is true with weight n.
- ***blocked*:** the event's guard is false.
- ***execute*:** the event has been chosen for execution in the next system transition.

This structure maps each one of the events of the model to their corresponding event state in a given system configuration. Therefore, it allows to check after each system transition the states of the events. In the meantime, this section focuses on how this structure is initialized, but the use for this structure will be detailed in the following sections.

Let $\mathscr{M}$ be a probabilistic machine. The corresponding encoding of the initialization of the events' states in $\mathscr{R}$ is the term:

$$\langle\ \mathit{events} : \mathit{Events} \mid \mathit{state} : [\![\mathscr{M}]\!]_{\mathit{initEvtSt}}\ \rangle$$

Where $[\![\mathscr{M}]\!]_{\mathit{initEvtSt}}$ encodes the initialization of the event's states. The resulting term has sort `CONFIGURATION`, and represents an object of the class `Events`, with attribute `state`. $[\![\mathscr{M}]\!]_{\mathit{initEvtSt}}$ returns a set $ev(evt_1, st_1), ..., ev(evt_n, st_n)$ where each evt_i is a quoted identifier that represents the name of the event, and st_i is a term of sort `EvState`, that represents the state of the event. In this case, all of the $st_i =$ `unknown` , since in the initialization of the model none of the guards of the event have been evaluated. For the brake system, the resulting term after the encoding would be:

```
< events : Events | state: (ev('PushPedal, unknown)
                            ev('ReleasePedal, unknown)
                            ev('ApplyBrake, unknown)
                            ev('ApplyBrakeFailure, unknown)
                            ev('ReleaseBrake, unknown)) >
```

3.4 Initial State and System States in Maude

As seen in the preliminaries chapter, the states of the specification of a system in Maude, are represented using terms. For the translated models from probabilistic Event-B models, the initial system state of the system is:

$$\begin{aligned}&\langle\ [\![\mathscr{C}]\!]_{id} : \mathit{Context} \mid \mathit{sets} : [\![\mathscr{C}]\!]_{\mathit{sets}},\ \mathit{constants} : [\![\mathscr{C}]\!]_{\mathit{constants}}\ \rangle\\ &\langle\ [\![\mathscr{M}]\!]_{id} : \mathit{Machine} \mid \mathit{variables} : [\![\mathscr{M}]\!]_{\mathit{initVars}}\ \rangle\\ &\langle\ \mathit{events} : \mathit{Events} \mid \mathit{state} : [\![\mathscr{M}]\!]_{\mathit{initEvtSt}}\ \rangle\end{aligned}$$

This term has sort `CONFIGURATION`, and combines the previously explained objects. To refer to this term, the following notation will be used:

$$\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}$$

Where $\mathfrak{C}$ is the term that represents the context, $\mathfrak{M}_0$ is the term that represents the machine in the initial state, and $\mathfrak{E}$ is the term that represents the events' states in the current system configuration. The rest of the system states, are symbolized with $\mathfrak{C}\ \mathfrak{M}_i\ \mathfrak{E}$, and are obtained with the application of equations and rewrite rules, defined in $\mathscr{R}_{\mathscr{M}}$, over the initial state $\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}$. With these elements, the definition of the rewrite theory $\mathscr{R}$ is concluded. The following sections provide the definition of the equations and rewrite rules that define $\mathscr{R}_{\mathscr{M}}$.

3.5 *evalSt* Equation

The *evalSt* equation, determines if an event is enabled or blocked, given the current state of the machine $\mathfrak{M}_i$ and the encoding of the guards of the event e_i, represented as $[\![\mathscr{M}]\!]^{e_i}_{guards}$:

$$\mathfrak{C}\ \mathfrak{M}_i\ \langle events : Events \mid state : ev(e_1, unknown)...ev(e_n, unknown)\rangle =$$
$$\mathfrak{C}\ \mathfrak{M}_i\ \langle events : Events \mid state : ev(e_1, eval(\mathfrak{M}_i, [\![\mathscr{M}]\!]^{e_1}_{guards}))...ev(e_n, eval(\mathfrak{M}, [\![\mathscr{M}]\!]^{e_n}_{guards}))\rangle$$

To do this, for each one of the event states $ev(evt_1, st_1), ..., ev(evt_n, st_n)$, the equation evaluates the new state of the event e_i with the equation *eval*. The equation takes as input the current state of the machine $\mathfrak{M}_i$, i.e. the value of each one of the variables, and then determines if the guards of event e_i are satisfied, using the encoded version of the guards $[\![\mathscr{M}]\!]^{e_n}_{guards}$. The returned value by *eval*, can only be `enabled(n)`, if the guards of the event are satisfied in $\mathfrak{M}_i$, or `blocked` otherwise. For example, if *evalSt* is applied to the first state $\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}$, the obtained term is:

```
𝔆 𝔐0 < events : Events | state: (ev('PushPedal, enable(10))
                                  ev('ReleasePedal, blocked)
                                  ev('ApplyBrake,blocked)
                                  ev('ApplyBrakeFailure, blocked)
                                  ev('ReleaseBrake, blocked)) >
```

3.6 *chooseEvt* Rule

The *chooseEvt* rule chooses probabilistically, the next event to be executed according to the weights of the enabled events:

$$\mathfrak{C}\ \mathfrak{M}_i\ \langle state : ev(e_1, st_1)...ev(e_n, st_n)\rangle \rightarrow \mathfrak{C}\ \mathfrak{M}_i\ \langle state : ev(e_i, execute)\rangle$$

To do this, the following steps must be applied:

1. Filter out the events $ev(e_i, st_i)$ where $st_i =$ `blocked`. Thus, the remaining set of event states will have the form $ev(e_j, enabled(n_j)), ..., ev(e_k, enabled(n_k))$.

2. From the enabled events, choose probabilistically one of the events e_i, according to their weights $n_j ... n_k$, as done in probabilistic Event-B.
3. The chosen event e_i is paired with the state `execute`, and the resulting set of event states is $ev(e_i, execute)$.

For example, if the rule is applied to the following state from the brake example:

```
𝕮 𝔐ᵢ <events : Events | state: ev('PushPedal, blocked)
                               ev('ReleasePedal, enabled(10))
                               ev('ApplyBrake,enabled(7))
                               ev('ApplyBrakeFailure, enabled(3))
                               ev('ReleaseBrake, blocked)>
```

The resulting term with the highest probability is:

```
𝕮 𝔐ᵢ <events : Events | state : ev('ReleasePedal, execute)>
```

Since the *ReleasePedal* event has the biggest weight.

3.7 $execEvt_{e_i}$ Rule

For each one of the events e_i in the original probabilistic Event-B model, a rule $execEvt_{e_i}$ is encoded. It is defined as:

$$\mathfrak{C}\ \mathfrak{M}_i\ \langle state : ev(e_i, execute)\rangle \rightarrow \mathfrak{C}\ \mathfrak{M}_j\ \langle state : [\![\mathscr{M}]\!]_{initEvtSt}\rangle$$

where $\mathfrak{M}_j$ is obtained by changing the variable values according to the encoded probabilistic assignments of the specific event e_i. Furthermore, after making the system transition from $\mathfrak{M}_i \rightarrow \mathfrak{M}_j$, the states of the events are reseted to `unknown`, as done in the initialization of the model. For example, if the current state is:

```
𝕮 <'BrakeSystem: Machine | variables: ('brake |-> val(elt("released")),
                                       'pedal |-> val(elt("up")),
                                       'wear |-> val(elt(0))) >
  <events : Events | state: ev('PushPedal, execute) >
```

After executing rule $execEvt_{PushPedal}$ the resulting state with highest probability is:

```
𝕮 <'BrakeSystem: Machine | variables: ('brake |-> val(elt("released")),
                                       'pedal |-> val(elt("down")),
                                       'wear |-> val(elt(0))) >
  < events : Events | state: (ev('PushPedal, unknown)
                              ev('ReleasePedal, unknown)
                              ev('ApplyBrake, unknown)
                              ev('ApplyBrakeFailure, unknown)
                              ev('ReleaseBrake, unknown)) >
```

Since the *down* value for the *pedal* variable, has a 90% chance of being selected, in the probabilistic assignment from the original model.

3.8 $\mathscr{R}_{\mathscr{M}}$ in a Nutshell

Using all the previously explained elements of the rewrite theory $\mathscr{R}_{\mathscr{M}}$, it is possible to summarize how the encoding produces a rewrite theory, that has the same behavior of a probabilistic Event-B model:

1. The probabilistic Event-B is encoded to the rewrite theory $\mathscr{R}_{\mathscr{M}}$, using $[\![\cdot]\!]$:

$$[\![\cdot]\!] : (\mathscr{C}, \mathscr{M}) \to \mathscr{R}_{\mathscr{M}}$$

2. The system is initialized with term $\mathfrak{C}\ \mathfrak{M}_{\mathsf{o}}\ \mathfrak{E}$ inside $\mathscr{R}_{\mathscr{M}}$:

$$\mathfrak{C}\ \mathfrak{M}_{\mathsf{o}}\ \mathfrak{E}$$

3. For any state of the model, symbolized by $\mathfrak{M}_{\mathsf{i}}$, the event guards are evaluated using the equation *evalSt*.

$$\mathfrak{C}\ \mathfrak{M}_{\mathsf{i}}\ \langle state : ev(e_1, unknown)...ev(e_n, unknown)\rangle \xrightarrow{evalSt} \mathfrak{C}\ \mathfrak{M}_{\mathsf{i}}\ \langle state : ev(e_1, st_1)...ev(e_n, st_n)\rangle$$

4. The blocked events are filtered and the next event is chosen probabilistically from all the enabled events, based on the event's weights:

$$\mathfrak{C}\ \mathfrak{M}_{\mathsf{i}}\ \langle state : ev(e_1, st_1)...ev(e_n, st_n)\rangle \xrightarrow{chooseEvt} \mathfrak{C}\ \mathfrak{M}_{\mathsf{i}}\ \langle state : ev(e_i, execute)\rangle$$

5. The rule associated to the encoded version of the chosen event e_i in the previous step is executed. The resulting change of the variable values, is captured in the rewrite $\mathfrak{M}_{\mathsf{i}} \to \mathfrak{M}_{\mathsf{j}}$. Moreover, the states of the events are initialized again.

$$\mathfrak{C}\ \mathfrak{M}_{\mathsf{i}}\ \langle state : ev(e_i, execute)\rangle \xrightarrow{execEvt_{e_i}} \mathfrak{C}\ \mathfrak{M}_{\mathsf{j}}\ \langle state : ev(e_1, unknown)...ev(e_n, unknown)\rangle$$

6. Repeat steps 3,4 and 5 until deadlock (i.e. none of the events can be executed since the guards of the events are unsatisfiable), or until the rule $execEvt_{e_i}$ has been executed a fixed number of times (this value is represented as a constant named `MAX-STEPS` inside $\mathscr{R}_{\mathscr{M}}$).

Based on the presented encoding, it is possible to prove that the resulting rewrite theory $\mathscr{R}_{\mathscr{M}}$ is semantically correct, based on the probabilistic Event-B model $(\mathscr{C}, \mathscr{M})$:

Theorem 3.8.1 (adequacy) *Let* $(\mathscr{C}, \mathscr{M})$ *be an Event-B model and* $\mathscr{R}_{\mathscr{M}}$ *be a rewrite theory obtained by the encoding* $[\![\cdot]\!] : (\mathscr{C}, \mathscr{M}) \rightarrow \mathscr{R}_{\mathscr{M}}$*. Furthermore, let* s *and* s' *be states or configurations of* $\mathscr{M}$ *(i.e. valuations of the variables in* $\mathscr{M}$*), and* $s \xrightarrow{e_i,p} s'$ *be a system transition from state* s *to* s'*, using the event* e_i *with probability* p*. Then* $s \xrightarrow{e_i,p} s'$ *iff* $[\![\mathscr{M}]\!]_s \rightarrow_p [\![\mathscr{M}]\!]_{s'}$*, where* $[\![\mathscr{M}]\!]_s$ *and* $[\![\mathscr{M}]\!]_{s'}$ *correspond to the encoded version of states* s *and* s' *respectively, and* $\rightarrow_p$ *denotes one-step rewriting in* $\mathscr{R}_{\mathscr{M}}$ *with probability* p*.*

Proof. The reader can refer to [17] for the proof sketch of this theorem $\square$.

Chapter 4

MultiVeStA: Statistical Model Checking of PMaude Specifications

As mentionde before, MultiVeStA is an extension of both VeStA and PVeStA that can be used to perform statistical model checking of PMaude specifications. Therefore, the main purpose of this chapter is to present a general introduction to understand how MultiVeSta works, and how it can be used. This chapter is divided in the following sections:

1. Components of MultiVeStA
2. how MultiVeSta algorithm works
3. PMaude an MultiVeStA integration

4.1 MultiVeSta's components and processes

First and foremost, the MultiVeSta tool was developed by Andrea Vandin, and it is available to download in a public repository in GitHub [26]. The main components of the tool are:

- **Maude-3.0+yices2:** A folder that contains the Maude distribution.
- **apmaude.maude:** A Maude file that contains the necessary sorts, equations and rules to extend any PMaude specification, so it can be checked using the tool.
- **multivesta.jar:** A java executable that runs the simulations using the specified PMaude model, the MultiQuaTEx query, and the simulation parameters.

To run simulations of a specific PMaude specification, it is necessary to pass to the tool:

- **model.maude:** A Maude file, that contains the specification of the PMaude model that is going to be verified.

- **query.multiquatex:** A MultiQuaTEx file, with the quantitative temporal logic formula written in the MultiQuaTEx language, that will be used to verify the model.
- **Simulations Parameters:** There are multiple parameters to take into account when performing simulations with the tool:
  - α and δ, already explained in section 2.7.
  - bs, which defines the number of simulations that each batch will contain (this parameter will be further detailed in the current section).

With these components, it is then possible to run the simulations and obtain the results of the query, as seen in Figure 4.1. The way MultiVeStA obtains these simulation results,

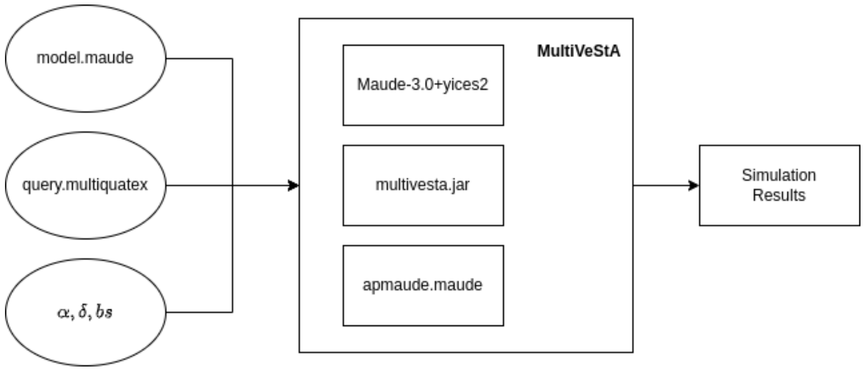


Figure 4.1: MultiVeSta's components

can be defined in the following steps:

1. MultiVeStA performs bs number of simulations and calculates the confidence interval based on the results obtained in each one of the simulations. Each simulation consists of:
   (a) Get the initial state of the system s_0 by running *model.maude*, which includes the theory specified in *apmaude.maude*,using the *Maude-3.0+yices2* distribution.
   (b) Evaluate the MultiQuaTEx formula in *query.multiquatex* over s_0. If the formula is satisfied, then the simulation ends and the result of the query is returned. Otherwise, MultiVeStA performs a one step in the simulation, to get the next state s_i.
   (c) Repeat step b with the new state s_i, until reaching a state where the formula is satisfied and the results are returned.

   As shown in section 2.7, the result of a MultiQuaTEx formula is a floating point value. Therefore, with each one of the values returned by the query in each simulation, the confidence interval of the batch is calculated.

2. If the confidence interval satisfies the parameter α and δ i.e. the real value of the query x has a probability $(1-\alpha)*100\%$ of being in the interval $[\bar{x}-\frac{\delta}{2}, \bar{x}+\frac{\delta}{2}]$, where $\bar{x}$ is the current value calculated for the query using the simulations, then MultiVeStA returns the results and ends it execution. If not, then MultiVeStA will send another batch of simulations until the confidence interval is finally reached.

In general, the complete process can be viewed in the following flowchart:

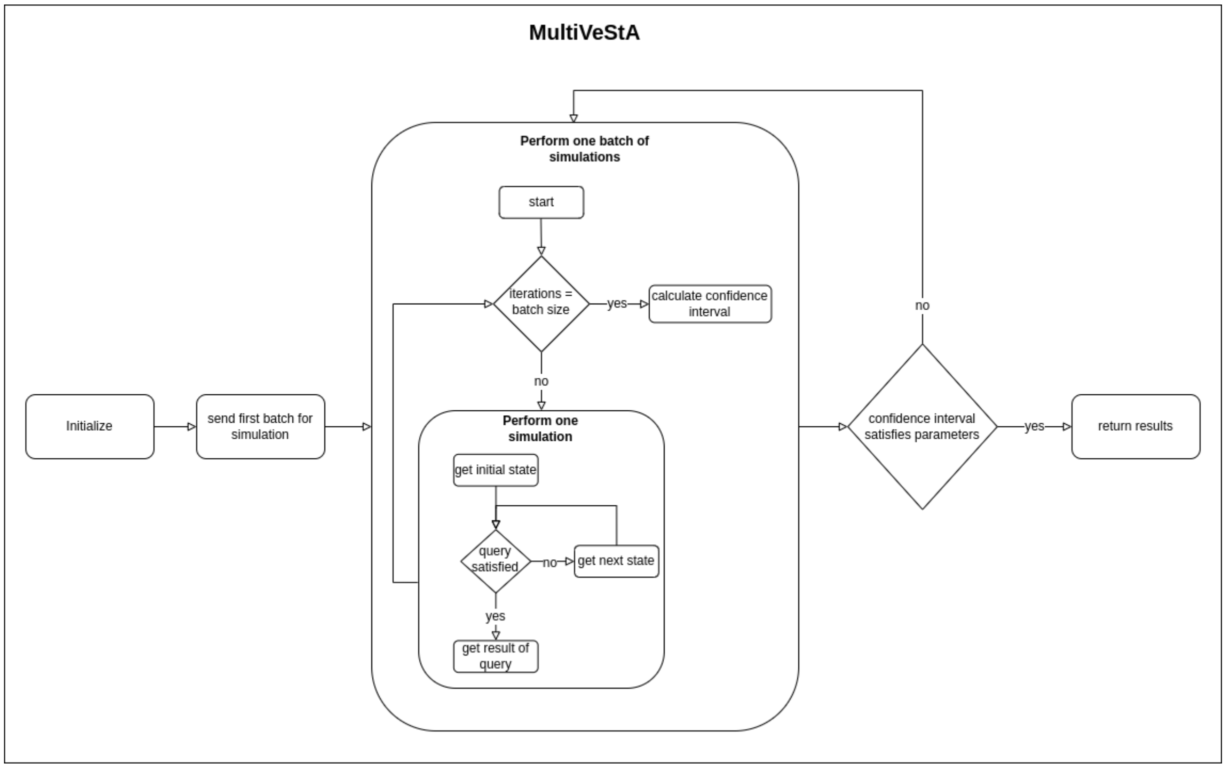


Figure 4.2: Flowchart of MultiVeStA's process

Having understood how the general process to perform simulations in MultiVeStA is done, it is necessary to define how to modify a PMaude specification to be able to run it with the tool. To do this, in the next section a general explanation will be given that illustrates how individual simulations work (sub-steps a,b and c from the step 1), and how this can be achieved including new elements into the PMaude models that is going to be verified.

4.2 MultiVeStA and Maude interaction

The way MultiVeStA sends and retrieves information from Maude, is through the Maude console. Initially, MultiVeStA loads the apmaude.maude and model.maude file through the Maude console. Then, MultiVeStA sends commands to the Maude console and Maude executes these commads. Finally, MultiVeStA reads these results from the Maude's console and use them to calculate the results of the simulations. This process is captured in the following sequence diagram:

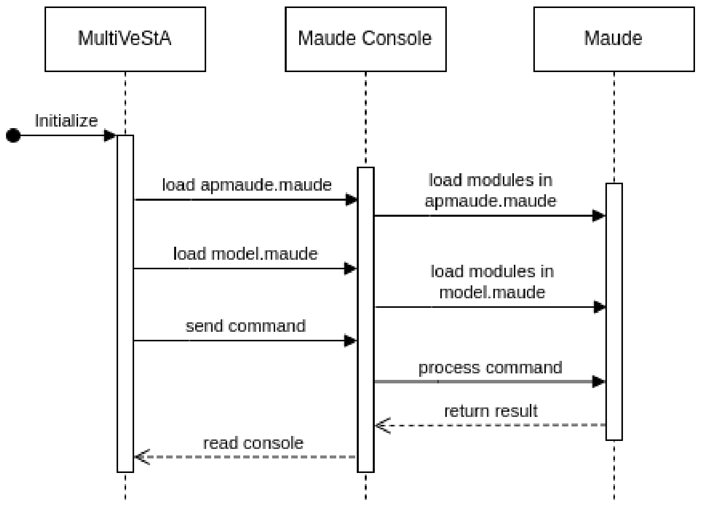


Figure 4.3: Sequence diagram of MultiVeStA and Maude interaction

In order to fully detail the process specified in Figure 4.3, it is necessary to define some preliminary operators and equations contained in apmaude.maude and model.maude. Hence, the following list contains the description of these structures and their respective definitions:

- **Scheduler:** the scheduler is a structure that contains the current number of steps or transitions in the simulation, represented as a float number, and a list of scheduler elements. In Maude it is specified as:

  ```
  op {_|_} : Float ScheduleList -> Scheduler .
  ```

  Each one of the scheduler elements corresponds to a pair that contains a message that identifies the next rewriting rule to be applied, and a float number that specifies in which step of the simulation the rule is going to be applied. They also contain a third value, which is a natural number that is used as a flag to drop or keep scheduler elements when inserted (this will be explained in more detail in the `insert` equation). In Maude it is defined as:

  ```
  op [_,_,_] : Float Msg Nat -> ScheduleElem [ ctor ] .
  op [_,_] : Float Msg -> ScheduleElem .

  var t1 : Float .
  var M1 : Msg .

  eq [ t1 , M1 ] = [ t1 , M1 , 0] .
  ```

The scheduler forms part of the system states. Thus, to be able to include it in the configuration, the sort `Scheduler` is a subsort of the sort `Configuration`.

```
subsort Scheduler < Configuration .
```

The scheduler is a very important part of the simulation process, since it will be in charged of determining the next transition or rewriting rule to be applied to the model's current configuration during a simulation. Associated to the scheduler, there is an operator named `insert`, that allows to insert new elements in the scheduler list. It is defined as:

```
op insert : Scheduler ScheduleElem -> Scheduler .
op insert : ScheduleList ScheduleElem -> ScheduleList .

vars t1 t2 gt : Float .
var SL : ScheduleList .
var e : ScheduleElem .
vars M1 M2 : Msg .
var p : Nat .

eq insert({ gt | SL },e) = { gt | insert(SL,e) } .
eq insert(SL , [ t2 , M2 , 1]) = SL .
eq insert( nilSL , [ t2 , M2 , 0]) = [ t2 , M2 , 0] .
eq insert([ t1 , M1 , p] ; SL , [ t2 , M2 , 0]) =
   if t1 < t2
    then [ t1 , M1 , p] ; insert(SL, [ t2 , M2 , 0])
    else ([ t2 , M2 , 0] ; [ t1 , M1 , p] ; SL)
   fi .
```

What this equation does is to take a scheduler, a scheduler element, and insert the element inside the list of the scheduler according to the float number in the scheduler elements, like a priority queue. In the end, the first element in the list is going to contain the message that specifies the next rule to be executed, since it has the smallest step. If the value of the flag is 1, then the element is not inserted, but if it is 0, then the element is inserted.

Moreover, the scheduler induces an implicit constraint over the model specified in model.maude: the selection of the next rule to be applied must be deterministic, in the sense that in every state of the simulation the next rule must be pre-selected, instead of the non-deterministic mechanism for choosing rules in Maude. For this reason, model.maude must have a rule or equation that chooses the next rule during the simulations.

- **Random Counter:** the random counter is a structure that contains a natural number, that will be used to select random numbers, similar to the `COUNTER` module explained in section 2.6. The reason for not using the COUNTER module in MultiVeStA and using a structure that saves the value of the counter, is because during simulation MultiVeStA resets the Maude console. Therefore, if the COUNTER module is used, the value of the counter to select random numbers will always be 0. With every application of a rule, the idea is to change the number of the random counter, so the generated random numbers are different in every simulation step. In Maude, the random counter is defined as:

```
op randomCounter : Nat -> Object [ ctor ] .
```

The random counter has sort Object since it is also part of the system states.

- **States of the Simulation:** as defined in section 2.4, states of a Maude specification are represented with a term. In this case, the states in the simulations have sort Configuration and are defined as:

```
var gt : Float .
var SL : ScheduleList .
var C : Configuration .
var n : Nat .

{gt | SL} {C randomCounter(n)}
```

where {gt | SL} is a term of sort Scheduler and C is a term of sort Configuration. The curly brackets {_} are an operator defined as:

```
op { _ } : Configuration -> Configuration [ ctor ] .
```

C represents the original states of the model without the required elements to run it with MultiVeStA, i.e the scheduler and the random counter. For instance, in the clock example from section 2.7, the term:

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

represents the configuration C. Finally, randomCounter(n), as explained before, is an object that contains a natural number that will be used to generate random numbers. Additionally, there is an operator named tick, defined as:

```
op tick : Configuration -> Configuration .
op mytick : Scheduler -> Configuration .

vars t1 gt : Float .
var C : Configuration .
var sc : Scheduler .
var p : Nat .
var SL : ScheduleList .
var M1 : Msg .

eq tick( sc C ) = mytick(sc) C .
eq mytick({ gt | [ t1 , M1 , p] ; SL }) = M1 { t1 | SL } .
eq mytick({ gt | nilSL }) = { gt | nilSL } .
```

its main purpose is to take out the first element from the scheduler list, add the message inside the scheduler element (i.e. `M1`) to the current state of the simulation, and replace `gt` with the step inside the scheduler element (i.e. `t1`).

- **Initial State:** as the name suggests it, this term corresponds to the initial state of the system. It is defined as:

```
op init : -> Configuration .
op initState : -> Configuration .
eq initState = closeWorldExceptScheduler(init) [ owise ]
```

For the initial state there are two operators involved: `init` and `initState`. `init` is a constant of sort `Configuration`, that will have to be defined by the user in the model.maude file with the equation:

```
eq init = {0.0 | nilSL} C0 randomCounter(0).
```

It initializes the scheduler with steps equal to 0.0 and an empty list of scheduler elements written as `nilSL`. The term `C0` is the configuration that represents the initial state of the model, and `randomCounter` starts with the value 0. `initState` corresponds to the application of the equation `closeWorldExceptScheduler` over the constant `init`. This equation is defined as:

```
op closeWorldExceptScheduler : Configuration -> Configuration .
eq closeWorldExceptScheduler(sc C) = sc { C } .
```

The equation is used to group all the objects from the configuration and isolate the scheduler. Thus, inside the curly brackets the objects will be the system state and the random counter: {`C randomCounter(n)`}.

- **Scheduler Rule:** Continuing with the observation made about the scheduler, a rule *SR* will be defined inside model.maude, that is in charge of selecting the next rule that is going to be applied to the current system state. For that reason, the *SR* rule must insert in the scheduler list the scheduler element that has the message that represents the rule chosen by the *SR* rule.

- **Rule Associated to a Message:** Each one of the scheduler elements contains a message `Mi` that is going to be used to identify the next rule that is going to be executed. Therefore, the associated rule must consider in its definition the presence of the message `Mi`. Based on this, the rule R_{M_i} can be defined as the rule associated with the message `Mi`, and specified as:

```
rl : Mi {gt | SL} {C randomCounter(n)} =>
        {gt | SL} {C' randomCounter(n)} .
```

The rule eliminates the message from the configuration and changes the term `C` to `C'`, indicating that the rule has been applied, and the state of the system has been changed.

- **Val operator:** This operator is used to define observations over the states of the system, allowing to extract valuable information that can be then evaluated in the MultiQuaTEx query using the `s.rval()` function, explained in section 2.7. It is defined as:

  ```
  op val : Nat Configuration -> Float .
  op val : String Configuration -> Float .
  ```

  Where the natural number and the string serve as identifiers, and the configuration represents the state where the observation is going to be made. For example, recall the state observation `s.rval("isBrk")` in the clock example from section 2.7. For MultiVeStA to be able to get the value for `s.rval("isBrk")` when evaluating the MultiQuaTEx query, the user must define in model.maude the corresponding equation with the `val` operator and the same identifier, in this case `"isBrk"`. The resulting equation in the model.maude files is:

  ```
  var C : Configuration .
  var gt : Float .
  var SL : ScheduleList .
  var cl : Oid .
  var n : Nat .

  --- returns 1.0 if the clock is broken, 0.0 if not
  op isBroken : Configuration -> Float .
  eq isBroken(< cl : Clock | state: broken, attrSet > C) = 1.0 .
  eq isBroken(< cl : Clock | state: working, attrSet > C) = 0.0 .

  --- Eval if the clock is broken
  eq val("isBrk", {gt | SL} {C randomCounter(n)}) = isBroken(C) .
  ```

  It is also important to mention, that ther are two predefined state observations in MultiVeStA, which are `s.rval("steps")` and `s.rval("time")`. The former returns the number of ticks done in the simulation, and the latter returns the value `gt` of the scheduler {`gt | SL`}.

Having defined the preliminary structures used in the simulations, the complete process of a single simulation will be described in detail as a series of steps:

0. The user defines the initial state as:

   ```
   eq init = {0.0 | nilSL} C0 randomCounter(0) .
   ```

   and the state observations:

   ```
   eq val("obs1", {gt | SL} {C randomCounter(n)}) = ...
   eq val("obs2", {gt | SL} {C randomCounter(n)}) = ...
   ...
   eq val("obsn", {gt | SL} {C randomCounter(n)}) = ...
   ```

   inside model.maude.

1. MultiVeStA loads the apmaude.maude and the model.maude files to Maude, using the Maude console.

2. MultiVeStA sends the command `rew initState`.

   (a) Maude reduces the `initState` term:

   ```
   initState → {0.0 | nilSL} {C0  randomCounter(0)} .
   ```

   (b) The rule SR chooses the next rule to be executed, represented with the message `Mi`, by inserting an element in the scheduler list. The random counter is also incremented:

   ```
   {0.0 | nilSL }              SR→    insert({0.0 | nilSL },[ gt + 1.0 , Mi])
   {C0 randomCounter(0)}               {C0 randomCounter(0+1)}
   ```

   (c) Maude reduces the terms:

   ```
   insert({0.0 | nilSL },[ 0.0 + 1.0 , Mi]) → {0.0 | [1.0 , Mi] }
   {C0 randomCounter(0+1)}                      {C0 randomCounter(1)}
   ```

3. MultiVeStA reads the Maude console and saves the current configuration. The saved configuration will be defined as `Conf`.

4. MultiVeStA evaluates the query in query.multiquatex, using the user defined equations for the `val` operators, to calculate the `s.rval()` state observations. To do this, for every `s.rval("obs")` in the query, MultiVeStA sends to Maude the command `red val("obs", Conf)`, and obtains the float number returned by the equation. Using these results of the `val` equations, then MultiVeStA is able to evaluate the query.

5. If the query is satisfied, then MultiVeStA gets the float number returned by the query. If not, then continue with step 6.

6. MultiVeStA sends the command `rew tick(Conf)`.
   (a) Maude reduces the term `rew tick(Conf)`:

   $$\begin{array}{lcl}\texttt{tick(\{gt | [gt' , Mi]\}} & \rightarrow & \texttt{Mi \{gt' | nilSL \}} \\ \quad\texttt{\{C randomCounter(n)\})} & & \texttt{\{C randomCounter(n)\}}\end{array}$$

   (b) Maude rewrites the term with the rule associated with the message `Mi`:

   $$\begin{array}{lcl}\texttt{Mi \{gt' | nilSL\}} & \xrightarrow{R_{M_i}} & \texttt{\{gt' | nilSL \}} \\ \texttt{\{C randomCounter(n)\}} & & \texttt{\{C' randomCounter(n)\}}\end{array}$$

   In this case, the term `C'` represents the next state of the model after applying R_{M_i} over `C`.

   (c) The rule SR chooses the next rule to be executed, represented with the message `Mj`:

   $$\begin{array}{lcl}\texttt{\{gt' | nilSL \}} & \xrightarrow{SR} & \texttt{insert(\{gt' | nilSL\},[gt' + 1.0 , Mj])} \\ \texttt{\{C' randomCounter(n)\}} & & \texttt{\{C' randomCounter(n+1)\}}\end{array}$$

   (d) Maude reduces the terms:

   $$\begin{array}{lcl}\texttt{insert(\{gt' | nilSL\},[gt' + 1.0 , Mj])} & \rightarrow & \texttt{\{gt' | [gt' + 1.0 , Mj]\}} \\ \texttt{\{C' randomCounter(n+1)\}} & & \texttt{\{C' randomCounter(n+1)\}}\end{array}$$

7. Repeat step 3 and 4.
8. If the query is satisfied, then get the float number returned by the query. If not, then repeat steps 6 trough 8 over the previous configuration `Conf`, until the formula is satisfied.

The complete process can be visualized in a sequence diagram, that extends the previous one presented in figure 4.3:

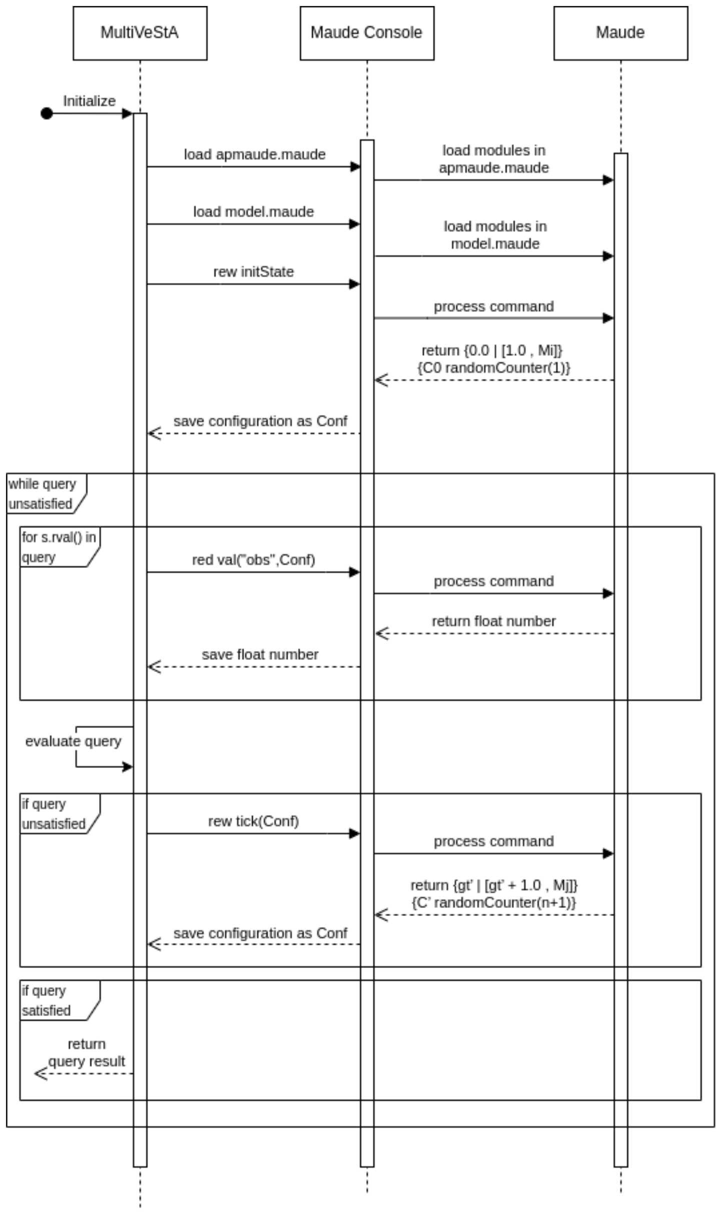


Figure 4.4: Extension of sequence diagram in Figure 4.3

Based on this introduction to MultiVeStA and chapter 3, the next section will present the integration between the rewriting logic approach to probabilistic Event-B and MultiVeStA.

4.3 Using MultiVeStA to verify $\mathscr{R}_{\mathscr{M}}$

To verify properties of probabilistic Event-B models encoded as probabilistic rewrite theories $\mathscr{R}_{\mathscr{M}}$, it is necessary to list the required components to complete this process. First, the mandatory theory to specify the encoded model that wants to be checked, should be included, i.e. the rewrite theory $\mathscr{R}$ with the modules `SAMPLER`, `EB-CORE`, `EBCONTEXT` and `EBMACHINE`. Then, the model.maude file should specify the system module `MODEL`, which includes $\mathscr{R}$. The interaction between the model and MultiVeStA is specified in Figure 4.5.

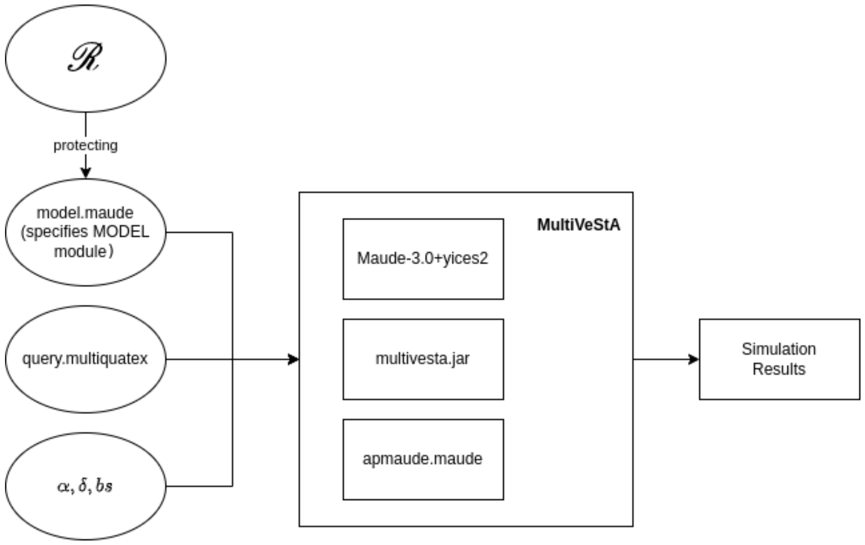


Figure 4.5: Interaction between the encoded model and MultiVeStA

Even though the simulation process is the same, as described in Figures 4.2 and 4.4, it is important to consider and identify the differences that arise when the simulation of the encoded models is done. Hence, the rest of this section will specify and describe how the same series of steps explained in section 4.2, function with the specific case of probabilistic rewrite theories $\mathscr{R}_{\mathscr{M}}$. Initially, the following list contains the required definitions to understand the simulation process:

- **States of the Simulation in $\mathscr{R}_{\mathscr{M}}$:** Given that the general definition for a state of a model that is going to be verified with MultiVeStA is:

  ```
  {gt | SL} {C randomCounter(n)}
  ```

  Then, replacing the term `C` with the term that represents states in $\mathscr{R}_{\mathscr{M}}$, the resulting configuration is:

  `{gt | SL} {`$\mathfrak{C}$ $\mathfrak{M}_i$ $\mathfrak{E}$ `randomCounter(n)}`

- **Initial State in $\mathscr{R}_{\mathscr{M}}$:** the initial state of the simulation is specified with the operator `init` and it is defined as:

  `eq init = {0.0 | nilSL}` $\mathfrak{C}$ $\mathfrak{M}_0$ $\mathfrak{E}$ `randomCounter(0) .`

- **Messages in $\mathscr{R}_{\mathscr{M}}$:** The messages that identify the next rule to be applied, are defined for the translated models $\mathscr{R}_{\mathscr{M}}$ with the operator:

  ```
  op _<-_ : Oid Content -> Msg .
  ```

  where the `Oid` is the object identifier of the machine $\mathfrak{M}$, and the second parameter of sort `Content` is used to identify the rule that is going to be applied. For example for the brake system example explained in section 2.3 and chapter 3, the message that identifies the push pedal rule in the model would be:

  ```
  op RulePushPedal : -> Content [ ctor ] .
  'BrakeSystem <- RulePushPedal
  ```

- **Scheduler Rule in $\mathscr{R}_{\mathscr{M}}$:** In the case of translated models $\mathscr{R}_{\mathscr{M}}$ by the encoding, the SR rule explained in the previous section corresponds to the *chooseEvt* rule. To integrate the *chooseEvt* rule with the scheduler, an auxiliary equation is used to identify the selected rule and insert it to the scheduler. The definition of this rule is:

  ```
  var gt : Float .
  var n : Nat .
  var SL : ScheduleList .
  var MNAME : Oid .
  var ruleQid : Qid .

  op scheduleEvent : Configuration -> Configuration .
  eq scheduleEvent( { gt | SL }
                    {$\mathfrak{C}$ $\mathfrak{M}_i$
                    < events  : Events  | state: ( ev(ruleQid, execute) ) >
                    randomCounter(n)} )
                   =
                    insert({ gt | SL },
                           [ gt + 1.0 , (MNAME <- qidToContent(ruleQid)),0])
                    {$\mathfrak{C}$ $\mathfrak{M}_i$
                    < events  : Events  | state: ( ev(ruleQid, execute) ) >
                    randomCounter(n)} .
  ```

  where `MNAME` is object identifier of the machine $\mathfrak{M}_i$, `ruleQid` is the quoted identifier of the selected rule, and `qidToContent` is an equation that maps the `Qid` of every rule, to a term of sort `Content`.

- **Rule R_{M_i} in $\mathscr{R}_{\mathscr{M}}$:** the $execEvt_{e_i}$ rule in $\mathscr{R}_{\mathscr{M}}$, corresponds to the R_{M_i} rule. To adapt each one of the rules $execEvt_{e_i}$ to resemble the behavior of R_{M_i}, it is necessary to include in the definition of $execEvt_{e_i}$, the message that represents the event. For example, if the rule $execEvt_{e_i}$ is defined as:

  ```
  rl [eventEi]: { gt | SL } $\mathfrak{C}$ $\mathfrak{M}_i$ $\mathfrak{E}$ randomCounter(n)
  ```

```
                 =>
                 { gt | SL } ℭ 𝔐_j 𝔈 randomCounter(n)
```

then, the adapted rule is:

```
rl [eventEi]: (MNAME <- eventEi) { gt | SL } ℭ 𝔐_i 𝔈 randomCounter(n)
              =>
              { gt | SL } ℭ 𝔐_j 𝔈 randomCounter(n)
```

where MNAME is the Oid of the machine, and eventEi is a term of sort content that identifies the rule.

With these definitions, it is now possible to redefine the simulation steps explained in the previous section:

0. The user defines the initial state as:

   ```
   eq init = {0.0 | nilSL} ℭ 𝔐_0 𝔈 randomCounter(0) .
   ```

   and the state observations:

   ```
   eq val("obs1", {gt | SL} {ℭ 𝔐 𝔈 randomCounter(n)}) = ...
   eq val("obs2", {gt | SL} {ℭ 𝔐 𝔈 randomCounter(n)}) = ...
   ...
   eq val("obsn", {gt | SL} {ℭ 𝔐 𝔈 randomCounter(n)}) = ...
   ```

   inside model.maude.

1. MultiVeStA loads the apmaude.maude, the files that contain the modules SAMPLER, EB-CORE, EBCONTEXT and EBMACHINE, and the model.maude file which includes the previous modules.

2. MultiVeStA sends the command rew initState.

   (a) Maude reduces the initState term:

   $$\texttt{initState} \rightarrow \texttt{\{0.0 | nilSL\} \{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}\ \texttt{randomCounter(0)\} .}$$

   (b) The rule *chooseEvt* chooses the next rule to be executed. The random counter is also incremented:

   $$\begin{array}{lcl} \texttt{\{0.0 | nilSL \}} & \xrightarrow{chooseEvt} & \texttt{scheduleEvent(\{0.0 | nilSL \}} \\ \texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}\ \texttt{randomCounter(0)\}} & & \texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}'\ \texttt{randomCounter(0+1)\})} \end{array}$$

   where $\mathfrak{E}'$ = <events:Events|state:(ev(ruleQid,execute))

   (c) Maude reduces the terms:

$$\texttt{scheduleEvent(\{0.0 | nilSL \}} \rightarrow \texttt{insert(\{0.0 | nilSL \},[0.0+1.0,}MG\texttt{,0])}$$
$$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}'\ \texttt{randomCounter(0+1)\})} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}'\ \texttt{randomCounter(1)\}}$$

$$\texttt{insert(\{0.0 | nilSL \},[0.0+1.0,}MG\texttt{,0])} \rightarrow \texttt{\{0.0 | [1.0 , }MG\texttt{] \}}$$
$$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}'\ \texttt{randomCounter(1)\}} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_0\ \mathfrak{E}'\ \texttt{randomCounter(1)\}}$$

Where $MG =$ `(MNAME<-qidToContent(ruleQid))`

3. Do steps 3, 4 and 5 from the previous section.
4. MultiVeStA sends the command `rew tick(Conf)`.
   (a) Maude reduces the term `rew tick(Conf)`:

   $$\texttt{tick(\{gt | [gt' , }MG\texttt{]\}} \rightarrow MG\ \texttt{\{gt' | nilSL \}}$$
   $$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_i\ \mathfrak{E}'\ \texttt{randomCounter(n)\})} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_i\ \mathfrak{E}'\ \texttt{randomCounter(n)\}}$$

   (b) Maude rewrites the term with the rule associated with the message `Mi`:

   $$MG\ \texttt{\{gt' | nilSL \}} \xrightarrow{execEvt_{e_i}} \texttt{\{gt' | nilSL \}}$$
   $$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_i\ \mathfrak{E}'\ \texttt{randomCounter(n)\}} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_j\ \mathfrak{E}\ \texttt{randomCounter(n)\}}$$

   Where $\mathfrak{M}_j$ is the next system state after applying the rule $execEvt_{e_i}$. Note that $\mathfrak{E} = \langle events : Events \mid state : ev(e_1, unknown)...ev(e_n, unknown)\rangle$

   (c) The rule *chooseEvt* chooses the next rule to be executed:

   $$\texttt{\{gt' | nilSL \}} \xrightarrow{chooseEvt} \texttt{scheduleEvent(\{gt' | nilSL \}}$$
   $$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_j\ \mathfrak{E}\ \texttt{randomCounter(n)\}} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_j\ \mathfrak{E}'\ \texttt{randomCounter(n+1)\})}$$

   where $\mathfrak{E}' =$ `<events:Events|state:(ev(ruleQid,execute))`.

   (d) Maude reduces the terms:

   $$\texttt{scheduleEvent(\{gt' | nilSL \}} \rightarrow \texttt{\{gt' | [gt' + 1.0 , }MG\texttt{]\}}$$
   $$\texttt{\{}\mathfrak{C}\ \mathfrak{M}_j\ \mathfrak{E}'\ \texttt{randomCounter(n+1)\})} \qquad \texttt{\{}\mathfrak{C}\ \mathfrak{M}_j\ \mathfrak{E}'\ \texttt{randomCounter(n+1)\})}$$

5. Repeat step 3.
6. If the query is satisfied, then get the float number returned by the query. If not, then repeat steps 4 trough 6 over the previous configuration `Conf`, until the formula is satisfied.

Following this process, it is then possible to do statistical model checking over $\mathfrak{R}_M$ rewrite theories, and evaluate MultiQuaTEx queries. In the next section, some case studies will be presented to illustrate the interplay between the encoder and MultiVeStA.

Chapter 5

Case Studies

The purpose of these case studies is to test the correct integration of the encoder with the MultiVeStA tool. In total there are 3 case studies: two probabilistic programs using dices and coins, a bounded re-transmission protocol, and a program that simulates a modal operator in positive negative logic. The repository that contains the code of this integration can be found in the following repository: `https://github.com/dfosorio/EventB2Maude-MultiVeStA`. Furthermore, this repository contains the code of the probabilistic Event-B models, their translated versions in PMaude, and the results of the simulations. A short guide on how to use the software can also be found in this repository, for any reader that might want to test or use the tool.

5.1 Dice Programs

The dice programs, based on the Knuth & Yao paper [27], were originally discovered in the PRISM model checker web page [28]. The idea is then to translate the model from the PRISM language to probabilistic Event-B, and afterwards translate the probabilistic Event-B model into PMaude using the encoder. The resulting PMaude specification will then be verified using MultiVeStA. If the results from the MultiVeStA simulations are the same as the PRISM model checker, then we can assume that the encoding and verification process were performed correctly. The probabilistic model for the first dice program is the following:

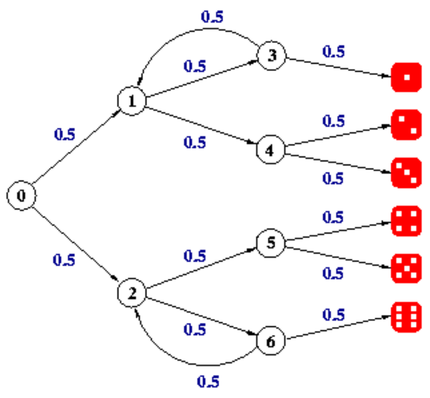


Figure 5.1: States and transitions of the dice model, taken from [28]

Having as a reference the Figure 5.1, the starting state of the model is the state 0. A fair coin is tossed to decide the next transition to execute. If the coin lands on heads, then the upper branch is taken. Conversely, when the coin lands on tails, the lower branch is taken. This process is repeated in each on of the sates 1 to 6, until reaching a terminating state, i.e. reaching a state where one of the dice faces is chosen. To represent this model in probabilistic Event-B, the process is straight forward: to represent all the states of the model a deferred set with all is created in the context.

```
CONTEXT ctxDiceProgram1
SETS
    STATES : { s0, s1, s2, s3, s4, s5, s6, s7 }
CONSTANTS
END
```

where the state s_7 represents all the possible terminating states. In the machine of the model, two variables will be considered: one that stores the current state of the system and other that contains the current value of the dice.

```
MACHINE DiceProgram1
  SEES ctxDiceProgram1

  VARIABLES
    st
    dice
  INVARIANTS
    st  : STATES
    dice : Nat
```

Each one of the transitions or coin tosses is represented with an event. For example, the first two transitions that start in state 0 can be represented with the event:

```
EVENT State0Trans
WEIGHT 1
WHERE
    st = s0
THEN
    st := {s1 @ 0.5 , s2 @ 0.5 }
END
```

As seen in the event, the fifty-fifty probability of the coin toss is represented with a probabilistic assignment. For transitions that precede a terminating state, they are represented by changing the state to s_7, and assigning to the dice the respective value. For example, the transitions that branch out of the fourth state are represented in Event-B as:

```
EVENT State4Trans
WEIGHT 1
WHERE
    st = s4
THEN
    st := s7
    dice := {2 @ 0.5 , 3 @ 0.5 }
END
```

Finally, to initilize the system the values s_0 and 0 are assigned to the variables *st* and *dice* respectively.

```
INITIALISATION
    st := s0
    dice := 0
```

The property that wants to be verified for this program is the probability of reaching a terminating state where the value of $dice = k$, where $k = 1, 2, ..., 6$. if the model is correct, then the probability is $P(dice = k) = \frac{1}{6}$ for all k.

Using the encoder, the PMaude specification is obtained by repeating the process in Chapter 3. The file that contains this specification can be found in the repository with the filename `DiceProgram1.maude`. In order to define the user observations `rval`, an equation needs to be added that obtains the value of the dice variable. Therefore, for each one of the possible values of the dice (1 to 6) an equation is added that returns true (represented as 1.0 since `rval` return float numbers) if the dice has that specific value. For example, to verify that the dice has the value 1, the following equation is added to the PMaude specification:

```
  --- User Defined Observations
eq val("obs1",{Conf < MNAME:Machine |
                     variables:('st |-> st , 'dice |-> dice) >}
```

```
                    {gt | SL})
                    = toFloat(((dice) =b (val(elt(1))))) .
```

Hence, when the `s.rval("obs1")` is called in the MultiQuaTEx file, it will return 1.0 if the value of the dice is 1, and 0.0 otherwise. Moreover, for every encoded model, the encoder will also generate inside the specification 2 equations that can be used to define `rval` observations, regarding the terminating state of the model:

```
eq val("isMax", {Conf} {gt | SL}) = if (gt >= MAX-STEPS)
                                    then 1.0
                                    else 0.0 fi .

 eq val("deadlock", {Conf < events : Events | state: (LEv) >}
                    {gt | SL}) = if (not-unknown(LEv) and not(one-firable(LEv)))
                                 then 1.0
                                 else 0.0 fi .
```

The first equation returns 1.0 (true) if the maximum number of transitions have been exeuted. Each transition corresponds to the execution of the *chooseEvt* rule, and the number that defines the maximum number of transitions is specified as a constant named as `MAX-STEPS` in the specification. By default it is set to 10000, but it can be modified directly in the PMaude specification. Inside the specification, this number is used to bound the number of transitions of the PMaude model, and terminating its execution when the number is reached.

The second equation returns 1.0 if the model has reached a deadlock, i.e. none of the guards of the translated events have been satisfied. To do this, the equation `not-unknown` (it verifies that all the states have a state different from `unknown`) and `one-firable` (it verifies if there is at least one event with satisfied guards) are used. It is important to understand correctly these two equations, since they will be used to define the terminating condition of most of the PMaude specifications that result from the translation.

For the verification of the model, the MultiQuaTEx formula that is used to verify the model is:

```
PDice1(n) = if ((s.rval("isMax") == 1.0) || (s.rval("deadlock") == 1.0))
            then s.rval(n) else # PDice1(n) fi ;

eval E[ PDice1("obs1") ] ;
eval E[ PDice1("obs2") ] ;
eval E[ PDice1("obs3") ] ;
eval E[ PDice1("obs4") ] ;
eval E[ PDice1("obs5") ] ;
eval E[ PDice1("obs6") ] ;
```

When a terminating state is reached, i.e. the simulation reached the maximum number of transitions or the simulation got in a deadlock state, then the formula returns the the value of the observation `s.rval(n)` where `n` is a parameter of the formula, that is going to be replaced in each one of the evaluation calls with `"obs1"`,`"obs2"`... or `"obs6"`. Using this formula, and the parameters $\alpha = 0.01, \delta = 0.01$ and $bs = 28$, the results of the simulation using MultiVeStA where:

Property	ObtainedValue	Variance	CI
PDice1(obs1)	0.169194084235345	0.140571212336999	0.00999772484533201
PDice1(obs2)	0.165127336320909	0.137864066309953	0.00999898248710578
PDice1(obs3)	0.166322650733297	0.138663192561158	0.00999737034957979
PDice1(obs4)	0.164898271712973	0.137710597576251	0.00999724079076279
PDice1(obs5)	0.166095890410959	0.138511810325858	0.00999571303811207
PDice1(obs6)	0.167870619946092	0.139693840237627	0.00999651791545913

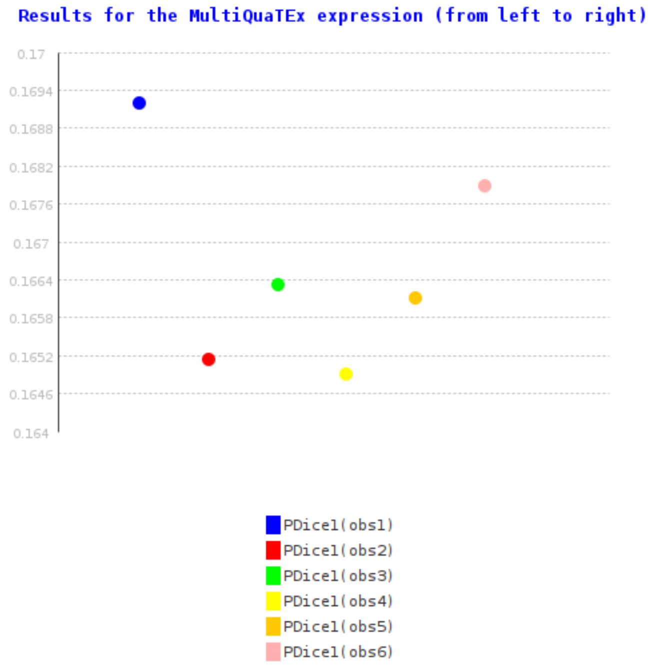


Figure 5.2: Graph of the simulation results of the first dice program

The total time it took to run the complete simulation was 2521.757 seconds (42 minutes). From this simulation, it is safe to say that the obtained results are close to the expected real value of $\frac{1}{6}$, and the verification of the first dice program is concluded.

The second dice program corresponds to the same scenario, but considering two dice throws, i.e. the same transition system but simulated twice. This time, the property that wants to be verified is the probability of obtaining k as the sum of the results of both dice, where $k = 2, 3, 4, ..., 12$. The expected probability for each one of the possible sums of two dice is:

k	Probability
2	1/36
3	1/18
4	3/36
5	1/9
6	5/36
7	1/6
8	5/36
9	1/9
10	3/36
11	1/18
12	1/36

To specify the model, one approach would be to add new variables *state2*, *dice2*, and duplicate the events specified in the previous dice program to change the new variables. Event though implementing the Event-B model in this way would work, for this model the optimized version proposed in [27] will be used. This version consists of the same principle: multiple states and coin tosses as transitions, until reaching a terminating state, i.e. the sum of two dices. The graph that represents this model is:

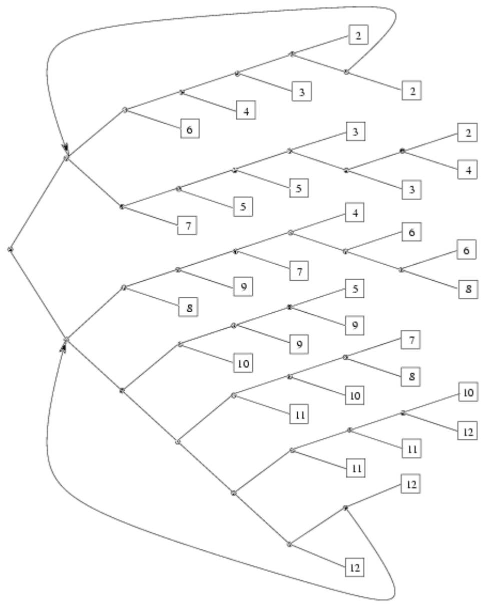


Figure 5.3: States and transitions of the two dice model, taken from [28]

Each one of the nodes in Figure 5.3, represents a state in the probabilistic Event-B model. Therefore, the new context is going to specify a deffered set with 33 states, and an extra 34th state to specify a terminating state.

```
CONTEXT ctxDiceProgram2
SETS
    STATES : { s0, s1, ... , s33, s34 }
CONSTANTS
END
```

For the variables, and the initialization of the machine, the Event-B specification is identical. Finally, for the events the principle is the same: For each one the transitions use a probabilistic assignment to symbolize the fifty-fifty chance in the system transitions. The resulting probabilistic Event-B model and the encoded version in PMaude, can be consulted in the `DiceProgram2.b` and `DiceProgram2.maude` files respectively. The equations that define the user observations are also defined in the same manner:

```
eq val("obs1", {Conf < MNAME : Machine |
                     variables: ('st |-> st , 'diceSum |-> diceSum) >}
                     {gt | SL}  ) = toFloat(((diceSum) =b (val(elt(2))))) .
```

In this case `"obs1"` references the case where $k = 2$, `"obs2"` references the case $k = 3$, and so on until `"obs11"`. The MultiQuaTEx formula used to verify the model is also very similar to the previous one:

```
PDice2(n) = if ((s.rval("isMax") == 1.0) || (s.rval("deadlock") == 1.0))
            then s.rval(n) else # PDice2(n) fi ;

eval E[ PDice2("obs1") ] ;
eval E[ PDice2("obs2") ] ;
eval E[ PDice2("obs3") ] ;
eval E[ PDice2("obs4") ] ;
eval E[ PDice2("obs5") ] ;
eval E[ PDice2("obs6") ] ;
eval E[ PDice2("obs7") ] ;
eval E[ PDice2("obs8") ] ;
eval E[ PDice2("obs9") ] ;
eval E[ PDice2("obs10") ] ;
eval E[ PDice2("obs11") ] ;
```

Using this formula, and the parameters $\alpha = 0.01, \delta = 0.01$ and $bs = 100$, the results of the simulation using MultiVeStA where:

Property	ObtainedValue	Variance	CI
PDice2(obs1)	0.03	0.0291037312475958	0.00995116917577923
PDice2(obs2)	0.0510077519379845	0.0484097138712153	0.00997972279089514
PDice2(obs3)	0.0853365384615385	0.0780579664517895	0.00997985024064457
PDice2(obs4)	0.114200743494424	0.101162694374703	0.00999035979240436
PDice2(obs5)	0.138710691823899	0.119473792835145	0.00998550057004364
PDice2(obs6)	0.164328767123288	0.137328585846037	0.00999266014168248
PDice2(obs7)	0.135176848874598	0.116907827497064	0.00998823325270086
PDice2(obs8)	0.110306513409962	0.0981427467677965	0.0099897800415653
PDice2(obs9)	0.0811616161616162	0.0745781747981354	0.00999816628115168
PDice2(obs10)	0.0551798561151079	0.0521387905863524	0.00997746359029961
PDice2(obs11)	0.0282191780821918	0.0274266131408506	0.0099855441351178

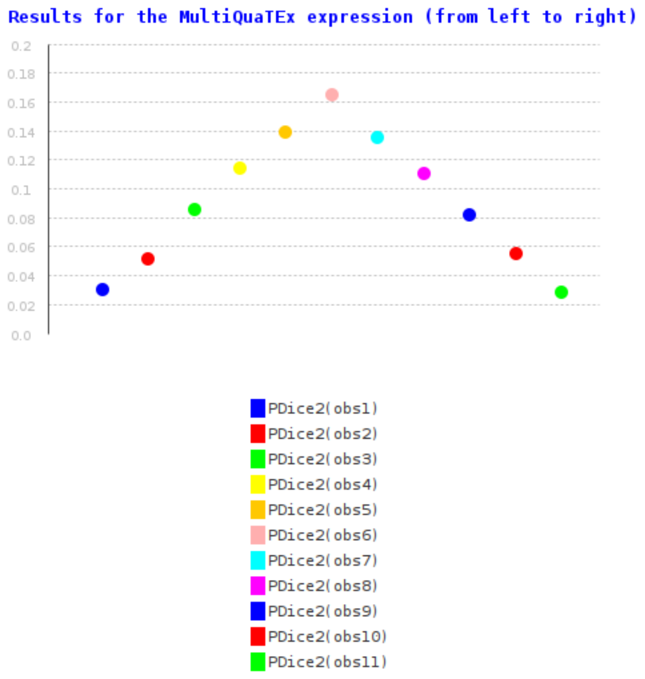


Figure 5.4: Graph of the simulation results of the second dice program

In total, the time it took to complete the simulation was 4583.059 seconds (76 minutes), and the results of the simulation are very close to the expected probability for each one of the possible values of the sum of the 2 dice.

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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] C. Olarte, C. Rocha, and D. Osorio, "A rewriting logic semantics and statistical analysis for probabilistic event-b," 2022.

[18] C. Olarte, D. Osorio, and C. Rocha, "Eventb2maude." `https://github.com/carlosolarte/EventB2Maude`, 2022.

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

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

[21] 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.

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

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

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

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

[26] A. Vandin, “Multivesta.” `https://github.com/andrea-vandin/MultiVeStA/wiki/Integration-with-PMaude-specification`, 2022.

[27] D. Knuth, “The complexity of nonuniform random number generation,” *Algorithms and Complexity, New Directions and Results*, pp. 357–428, 1976.

[28] prismmodelchecker.org, “Dice programs.” `https://www.prismmodelchecker.org/casestudies/dice.php`.