UNIVERSITY OF OSLO
Department of Informatics

Towards an Object-Oriented Modeling Language for Probabilistic Open Distributed Systems

Research Report 397

Lucian Bentea

Olaf Owe

ISBN 82-7368-358-3
ISSN 0806-3036

22 June 2010

Contents

1 Introduction **2**

2 Preliminaries **3**
- 2.1 Probabilistic Rewrite Theories 4
- 2.2 Overview of Creol 5

3 Syntax and Semantics of PCreol **7**
- 3.1 Generating Pseudo-Random Numbers 8
- 3.2 Probabilistic Choice Operator 8
- 3.3 Random Assignment 12
- 3.4 Lossy Communication and Component Failures 13

4 Implementation **14**
- 4.1 Overview of PMaude and VeStA 14
- 4.2 A Stochastic Time Model for PCreol 15
- 4.3 Implementation of the Stochastic Time Model 16
- 4.4 Examples 18

5 Related Work **22**

6 Conclusions and Future Research **22**

A Probability Theory and Stochastic Processes **27**

B Further Questions **28**

Towards an Object-Oriented Modeling Language for Probabilistic Open Distributed Systems

Lucian Bentea, Olaf Owe
Dept. of Informatics – Univ. of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mails: {lucianb,olaf}@ifi.uio.no

June 22, 2010

Abstract

In this paper, we propose a probabilistic extension of the Creol modeling language, called PCreol, for which we give the operational semantics in Probabilistic Rewrite Logic. We give details on the implementation of a prototype PCreol interpreter, executable in Maude, on top of the existing one for Creol. We also achieved the integration of PCreol with the VeStA tool, which allows for statistical model checking and statistical quantitative analysis of PCreol programs. We give two example PCreol programs and show how VeStA can be used to study their properties. The paper is concluded with a number of future research directions. [1]

1 Introduction

Software systems of today are often distributed, consisting of independent and concurrently executing units which communicate over networks of different quality, and which are supposed to work in open and evolving environments. It is non-trivial to design, model and program such systems and in particular to analyze system properties and reliability. The high degree of nondeterminism makes model checking difficult. For such systems, non-functional properties expressing probabilistic behaviour are valuable.

At the modeling level there is a need for high-level programming constructs making interaction and process control more manageable. And there is a need for methodology and tools that can be used to investigate system properties and robustness. Creol, short for *Concurrent, Reflective, Object-Oriented Language*, is an executable modeling and programming language introduced in [17], meeting these challenges. It is tailored for modeling software systems made up of physically distributed components, each running on its own processor and communicating with one another through asynchronous method calls. A Creol system runs in an open environment in which components may appear or disappear, while some components may even change their functionality, *during* their execution. The language features conditional and unconditional *processor release points*, allowing each object to suspend its execution until a later time and execute another (enabled) process.

The main aim of our work is to extend Creol's syntax and semantics with probabilistic features, in order to be able to model realistic behaviour of open distributed systems, such as lossy communication network, independent processor speeds, components running random algorithms, or an open environment exhibiting probabilistic behaviour.

[1]Partially supported by the Nordunet3 project "COSoDIS – Contract-Oriented Software Development for Internet Services" (`http://www.ifi.uio.no/cosodis/`).

This extension is given the name PCreol, which is short for Probabilistic Creol. We introduce the following new syntactic constructs: a command for generating pseudo-random numbers, a probabilistic choice operator, random assignment, lossy inter-object communication and probabilistic object creation. We also give their operational semantics in terms of *probabilistic rewrite theories* [20].

Currently, model checking of Creol programs can only be achieved through Maude's LTL model checker and its breadth-first state exploration facilities. The disadvantage is that even average-sized Creol programs lead to state space explosion, making it infeasible to model check them through Maude. A possible alternative is to entirely give up on using Creol to implement the model and instead write its low-level specification directly in Maude. But distributed, object-oriented models in Maude also lead to state explosion problems. Therefore, it would be more beneficial to continue using Creol's high-level, object-oriented programming paradigm, in combination with a model checking tool that scales well with the size of the model. Following this direction, we have made an integration of PCreol with VeStA ([26], [27], [28]), which allows for *probabilistic reasoning* on PCreol programs, via statistical model checking against Continuous Stochastic Logic [4] formulae and statistical quantitative analysis against queries expressed in QuaTEx, short for Quantitative Temporal Expressions [2]. This integration is essentially achieved by refining the original Creol operational semantics, so that *representative* runs of a Creol program are more easily obtained through discrete-event simulations and the model checking problem of large models becomes feasible through statistical model checking. The refinement process includes adding an explicit notion of time to the global configuration of the program and scheduling objects to execute at random time instants. This resolves all nondeterminism in the interpreter, allowing VeStA to run discrete-event simulations and do statistical analysis of PCreol programs.

The research report is structured as follows. Section 2 contains a few preliminaries on probabilistic rewrite logic and gives a short overview of the syntax and semantics of the Creol programming language. We then proceed with the main contribution of this work in Section 3, namely extending the syntax and semantics of Creol with probabilities. Thus, we give the operational semantics of PCreol in probabilistic rewrite logic. Section 4 gives details on the actual Maude implementation of the PCreol interpreter, as well as a few examples of PCreol programs, showing how they can be statistically model checked and how quantitative information can be extracted from them using the VeStA tool. In Section 5 we discuss related projects and the features that make PCreol different from them, while Section 6 concludes the report with the identification of a number of topics for future research.

2 Preliminaries

There are several reasons why the semantic framework of probabilistic rewrite theories is one of the most suitable when defining the formal semantics of our proposed probabilistic extension to Creol. First of all, [16] introduces Creol's operational semantics using *rewrite logic*, which can be seen as a particular case of probabilistic rewrite logic. Therefore, the framework that we use allows us to formulate a natural probabilistic extension to Creol's operational semantics, in which all of Creol's operational rules are kept the same, without the need to redefine them. Also, [20] describes probabilistic rewrite theories as an unifying semantic framework for several models of probabilistic systems. They show that certain mappings exist and that it is possible to implement algorithms that convert probabilistic rewrite logic specifications into specifications expressed in different formalisms. In theory, this would allow the use of existing tools like PRISM [22] or PEPA [29] in order to model check PCreol programs. However, it also requires implementing an algorithm that converts the current Maude source code of the

Creol interpreter to match the input language of these tools. We consider this beyond the scope of our paper and suggest it as a topic of future research. The point that we want to make is that using probabilistic rewrite theories to define the operational semantics of PCreol saves a lot of time and effort, only requiring the implementation of newly added operational rules, while the source code for the rest of the interpreter stays the same. In this respect, the implementation of our probabilistic extension to Creol can be seen as a *patch* to the Creol interpreter, making it easy to keep in sync with the latest version of Creol. The remainder of this section introduces probabilistic rewrite theories and provides an overview of the Creol programming language, emphasizing on the features that we extend to the probabilistic setting in Section 3.

2.1 Probabilistic Rewrite Theories

A *membership equational theory* [23] is a pair (Σ, E), with Σ its *signature* comprised of a set K of *kinds*, S_k a set of *sorts* for each $k \in K$ and a set of *function symbols* of the form $f : k_1 \dots k_n \to k$, where $k, k_1, \dots, k_n \in K$; the set E contains *membership equational logic* sentences [23], i.e. *conditional* Σ-equations and Σ-memberships of the form

$$(\forall X)\ \ t = t' \quad \Leftarrow \quad \left[\bigwedge_{i=1}^{n}(u_i = e_i)\right] \wedge \left[\bigwedge_{i=1}^{m}(w_i : s_i)\right], \tag{1}$$

$$(\forall X)\ \ t : s \quad \Leftarrow \quad \left[\bigwedge_{i=1}^{n}(u_i = e_i)\right] \wedge \left[\bigwedge_{i=1}^{m}(w_i : s_i)\right], \tag{2}$$

correspondingly, where n, m are positive integers, the symbols t, t', $u_1, u_2, \dots, u_n$, $w_1, w_2, \dots, w_m$ represent terms, X denotes the *variables* in these terms and $s_1, s_2, \dots, s_m$ are sorts. The algebra of terms associated with the signature Σ is denoted by T_Σ.

A membership equational theory (Σ, E) together with a collection of *structural axioms* A generate an *initial algebra* $T_{\Sigma, E\cup A}$. Provided that E is terminating, confluent and sort-decreasing modulo A [7], let $\mathrm{Can}_{\Sigma,E/A}$ be the algebra of canonical forms, i.e. fully simplified terms, which is isomorphic to the initial algebra $T_{\Sigma,E\cup A}$. If t is a fully simplified term w.r.t. the set of equations E, let $[t]_A$ be its A-equivalence class. Given a collection of variables X, a mapping $[\theta]_A : X \to \mathrm{Can}_{\Sigma,E/A}$ is called an *E/A-canonical ground substitution* for X. We use the notation $[\theta]_A$ to emphasize that E/A-canonical ground substitutions are induced by ordinary substitutions $\theta : X \to T_\Sigma$, provided that $\theta(x)$ is fully simplified w.r.t. the set of equations E modulo A, for each variable $x \in X$. Let $\mathrm{CanGSubst}_{E/A}(X)$ be the set of all E/A-canonical ground substitutions associated with the set of variables X.

We now introduce a few notions from probability theory, which are needed later. A *σ-algebra* on a set $\Omega \neq \emptyset$ is a set $\mathcal{F} \subseteq 2^\Omega$ with $\emptyset \in \mathcal{F}$, which is closed under complement and under finite or countably infinite unions. The empty set and the power set 2^Ω are trivial examples of σ-algebras on Ω. Given a σ-algebra $\mathcal{F}$ on a set Ω, a *probability measure* on $\mathcal{F}$ is a function $P : \mathcal{F} \to [0,1]$ with the properties that $P(\Omega) = 1$ and $P\left(\cup_{i\in I} A_i\right) = \sum_{i\in I} P(A_i)$, for all finite or countably infinite collections $\{A_i\}_{i\in I} \subseteq \mathcal{F}$ of pairwise disjoint sets. Let $\mathrm{PFun}(\Omega, \mathcal{F})$ be the set of all probability measures defined on the σ-algebra $\mathcal{F}$ over Ω. A *probability space* is a triple $(\Omega, \mathcal{F}, P)$, where $\mathcal{F}$ is a σ-algebra on Ω and $P : \mathcal{F} \to [0,1]$ is a probability measure on $\mathcal{F}$; the set Ω is also known as the *sample space*, while the elements of $\mathcal{F}$ are called *events*.

A *rewrite theory* is a triple (Σ, E, R), with (Σ, E) a membership equational theory and R a collection of *conditional rewrite rules* of the form

$$t(X) \longrightarrow t'(X) \ \ \textbf{if} \ \ C(X), \tag{3}$$

where $t(X)$ and $t'(X)$ are terms of the same kind and $C(X)$ is a condition given by a conjunction of equations, memberships or rewrites referring to the variables in X. A *probabilistic rewrite theory* is a tuple $(\Sigma, E \cup A, R, \pi)$, where $(\Sigma, E \cup A, R)$ is a rewrite theory with rules of the form

$$t(X) \longrightarrow t'(X, Y) \;\; \textbf{if} \;\; C(X), \tag{4}$$

the set X contains the variables in t, Y represents the variables in t' that are not in t, $C(X)$ is a conjunction of Σ-equations and Σ-memberships with variables taken from X and

$$\pi : R \to \mathrm{PFun}(\mathrm{CanGSubst}_{E/A}(Y), \mathcal{F}_r)^{[\![C]\!]} \tag{5}$$

is a function which assigns to each rule $r \in R$ a mapping

$$\pi_r : [\![C]\!] \to \mathrm{PFun}(\mathrm{CanGSubst}_{E/A}(Y), \mathcal{F}_r), \tag{6}$$

where

$$[\![C]\!] = \{\, [\mu]_A \in \mathrm{CanGSubst}_{E/A}(X) \; ; \; E \cup A \vdash \mu(C) \,\}, \tag{7}$$

is the set of all E/A-canonical ground substitutions for X that satisfy the condition C and $\mathcal{F}_r$ is a σ-algebra on $\mathrm{CanGSubst}_{E/A}(X)$. A *probabilistic rewrite rule* $r \in R$ can then be given using the following syntax:

$$t(X) \longrightarrow t'(X, Y) \;\; \textbf{if} \;\; C(X) \;\; \textbf{with probability} \;\; Y := \pi_r(X). \tag{8}$$

If $\mathrm{CanGSubst}_{E/A}(X) = \emptyset$ due to Y being empty, we say that $\pi_r(X)$ defines a *trivial distribution*. This happens in the case of standard rewrite rules, without any probabilities assigned to them. By allowing trivial distributions, probabilistic rewrite theories can express both nondeterministic and probabilistic behaviour of a system.

2.2 Overview of Creol

The aim of this section is to provide a brief introduction to the syntax and operational semantics of the Creol object-oriented programming language, which we extend with probabilistic features in Section 3. For a detailed introduction to Creol we refer to [16]. A summary of the language syntax is given in Figure 1, as it also appears in [18]. The part of the syntax that we aim to extend is the syntactic category Stmt of Creol statements.

The operational semantics of Creol is given in rewrite logic and its implementation is executable through Maude [9]. The configuration of a Creol program consists of a multiset of objects, classes and messages, following the Actor model proposed in [3], allowing the specification of actors that execute local tasks and communicate through asynchronous message passing. We use empty syntax, i.e. whitespace, to denote the associative and commutative multiset concatenation operator. At each execution step, the Creol program makes a transition from one configuration to another, which results from all possible local transitions between its subconfigurations. Local transitions of the program are expressed as conditional rewrite rules of the form:

$$\text{subconfiguration}_1 \;\; \longrightarrow \;\; \text{subconfiguration}_2 \;\; \textbf{if} \;\; \text{condition}. \tag{9}$$

Creol *objects* are denoted by constructs of the form

$$\langle\; O : Ob \mid Cl,\, Pr,\, PrQ,\, Lvar,\, Att,\, EvQ,\, Lcnt \;\rangle, \tag{10}$$

where O is the object's identifier, Cl is its corresponding class, Pr contains its active process code and PrQ is a multiset of pending processes. Also, Att contains the object's

Syntactic categories	*Definitions*
$C, I, m \in \mathsf{Names}$	$IF ::= \texttt{interface}\ I\ [\texttt{inherits}\ \overline{I}]\ \texttt{begin}\ \{\texttt{with}\ I\ \overline{Sg}\}\ \texttt{end}$
$t \in \mathsf{Label}$	$CL ::= \texttt{class}\ C\ [\overline{x : I}]\ [\texttt{inherits}\ \overline{C}\ [(\overline{e})]]\ [\texttt{implements}\ \overline{I}][\texttt{contracts}\ \overline{I}]$
$g \in \mathsf{Guard}$	$\quad \texttt{begin}\ \{\texttt{var}\ \{x : I\ [:= e]\}\}\ \overline{M}\ \{\texttt{with}\ I\ \overline{M}\}\ \texttt{end}$
$p \in \mathsf{MtdCall}$	$M ::= Sg == [\texttt{var}\ \{x : I\ [:= e]\};]\ \overline{s}$
$s \in \mathsf{Stmt}$	$Sg ::= \texttt{op}\ m\ ([\texttt{in}\ \overline{x : I}][\texttt{out}\ \overline{x : I}])$
$x \in \mathsf{Var}$	$g ::= b \mid t? \mid g \wedge g \mid g \vee g$
$e \in \mathsf{Expr}$	$s ::= \texttt{begin}\ \overline{s}\ \texttt{end} \mid \overline{s} \,\Box\, \overline{s} \mid x := e \mid x := \texttt{new}\ C[(\overline{e})]$
$o \in \mathsf{ObjExpr}$	$\quad \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ \overline{s}\ [\texttt{else}\ \overline{s}]\ \texttt{end} \mid \texttt{while}\ b\ \texttt{do}\ \overline{s}\ \texttt{end}$
$b \in \mathsf{BoolExpr}$	$\quad \mid [t]![o.]m(\overline{e}) \mid t?(x) \mid \texttt{release} \mid \texttt{await}\ g \mid [\texttt{await}][o.]m(\overline{e}; \overline{x})$

Figure 1: The syntax of Creol. The terms denoted by $\overline{e}$, $\overline{x}$, and $\overline{s}$ represent lists over terms of the corresponding syntactic categories, the notation $\{\ldots\}$ represents lists over larger syntactical elements, and $[\ldots]$ denotes optional elements. Elements in a list are separated by a comma, while statements in a statement list are separated by semicolon.

variables and *Lvar* gives the values of these variables. Finally, *EvQ* is a multiset of unprocessed messages and the value of *Lcnt* is used to identify method calls. Similarly, a Creol *class* is defined using the syntax

$$\langle\ C : Cl \mid Par,\ Att,\ init,\ Mtds,\ Ocnt\ \rangle, \tag{11}$$

in which C is the class identifier, *Par* is the list of class parameters, *Att* is its list of attributes, *init* contains the Creol code for the *constructor* of class C and *Mtds* is the multiset of class methods including *run*, a special method that is automatically executed after the class constructor. Finally, *Ocnt* gives the current number of instances of class *C*. Messages sent between objects are the asynchronous *invocation* and the asynchronous *completion* messages, with the syntax *invoc*(o_1, o_2, m, in) and *comp*(o_1, out), correspondingly. The meaning of such a pair of invocation and completion messages is that object o_1 calls method m of object o_2, with arguments *in* and the result is stored in the *out* parameter of the completion message. Note that we omit the object o_1 sending or receiving a message, whenever it is understood from the context.

Among the basic Creol statements we mention those which we extend to the probabilistic setting: *nondeterministic choice* with the syntax $\mathrm{S}_1 \,\Box\, \mathrm{S}_2$, where S_1 and S_2 are statement lists, *object creation* using the *new* operator, as well as *asynchronous communication* with the syntax $t!o.m(\overline{e})$, where o is the object whose method m is called with parameter list $\overline{e}$, and t is a label that can be used to query for the return value of this method call, at some point in the *future* execution of the current object. The label t is also called a *future variable* in [1]. We now give the operational semantics for these Creol statements that we wish to extend. This is to observe what the operational semantics for these statements was before extending it with probabilistic features. We refer to [16] for a detailed and complete description of Creol's operational semantics in rewrite logic.

The semantics for the nondeterministic choice operator is given through the following conditional rewrite rule, using the commutativity and associativity properties of this operator,

$$\begin{array}{l}
\langle\ O : Ob\ \mid Pr : (\mathrm{S}_1 \,\Box\, \mathrm{S}_2)\,;\mathrm{S}_3,\ PrQ : \mathrm{Q},\ Lvar : \mathrm{L},\ Att : \mathrm{A}\ \rangle \\
\longrightarrow \\
\langle\ O : Ob\ \mid Pr : \mathrm{S}_1\,;\mathrm{S}_3,\ PrQ : \mathrm{Q},\ Lvar : \mathrm{L},\ Att : \mathrm{A}\ \rangle \\
\textbf{if}\ ready(\mathrm{S}_1,\ (\mathrm{A}\ ;\ \mathrm{L}),\ \mathrm{Q})
\end{array}$$

where O is an arbitrary object and *ready* is a predicate whose value tells whether the given process S_1 is ready for execution, in the context of the variable bindings $(\mathrm{A}\ ;\ \mathrm{L})$

and the object's pending processes multiset $\textsc{q}$. Notice that we omit irrelevant attributes, in the style of Full Maude [9]. Thus, the previous rewrite rule applies to instances O of any class, as the class attribute *Cl* is not included.

The *new* operator for object creation is given the following semantics

$$
\begin{array}{l}
\langle\ O : Ob \mid Pr :\ (\textsc{v} := new\ C(\textsc{e});\ \textsc{s}),\ Lvar :\ \textsc{l},\ Att :\ \textsc{a}\ \rangle \\
\langle\ C : Cl \mid Par :\ \textsc{v},\ Att :\ \textsc{a}',\ init :\ (\textsc{s}', \textsc{l}'),\ Ocnt :\ n\ \rangle \\
\longrightarrow \\
\langle\ O : Ob \mid Pr :\ (\textsc{v} := C\#n;\ \textsc{s}),\ Lvar :\ \textsc{l},\ Att :\ \textsc{a}\ \rangle \\
\langle\ C\#n :\ Ob \mid Cl :\ C,\ Pr :\ \textsc{v} := eval(\textsc{e},\ (\textsc{a}\ ;\ \textsc{l}));\ \textsc{s}';\ run, \\
PrQ : \varepsilon,\ Lvar :\ \textsc{l}',\ Att :\ self \mapsto C\#n;\ \textsc{v};\ \textsc{a}',\ Lcnt :\ 1\ \rangle \\
\langle\ C : Cl \mid Par :\ \textsc{v},\ Att :\ \textsc{a}',\ init :\ (\textsc{s}', \textsc{l}'),\ Ocnt :\ next(n)\ \rangle
\end{array}
$$

where the operation *new* $C(\textsc{e})$ creates a new object of class C with parameters given by $\textsc{e}$ and $eval(\textsc{e},\ (\textsc{a};\ \textsc{l}))$ is a function which evaluates the expression list $\textsc{e}$ in the context of the list of variable bindings $(\textsc{a};\ \textsc{l})$. Also, notice that the active process *Pr* of the newly created object $C\#n$ contains a call to its corresponding class constructor $\textsc{s}'$, immediately followed by a call to its *run* method. The *next* operation generates a fresh label, from a given old label n.

The operational semantics for asynchronous *invocation* messages is given by the rewrite rule

$$
\begin{array}{l}
\langle\ O : Ob \mid Pr :\ t!x.m(\textsc{e});\ \textsc{s},\ Lvar :\ \textsc{l},\ Att :\ \textsc{a},\ Lcnt :\ n\ \rangle \\
\longrightarrow \\
\langle\ O : Ob \mid Pr :\ t := n;\ \textsc{s},\ Lvar :\ \textsc{l},\ Att :\ \textsc{a},\ Lcnt :\ next(n)\ \rangle \\
invoc(eval(x,\ (\textsc{a};\ \textsc{l})),\ m,\ (O\ n\ eval(\textsc{e},\ (\textsc{a};\ \textsc{l}))))
\end{array}
$$

where n is the label value used to identify the future variable t. A separate rule takes the invocation message into the process queue *PrQ* of the called object. Similarly, the formal semantics for asynchronous *completion* messages is given through

$$
\begin{array}{l}
\langle\ O : Ob \mid Pr :\ return(\textsc{v});\ \textsc{s},\ Lvar :\ \textsc{l},\ Att :\ \textsc{a}\ \rangle \\
\longrightarrow \\
\langle\ O : Ob \mid Pr :\ \textsc{s},\ Lvar :\ \textsc{l},\ Att :\ \textsc{a}\ \rangle \\
comp(eval((caller\ label\ \textsc{v}),\ (\textsc{a};\ \textsc{l})))
\end{array}
$$

where *caller* is the object that made the call, *label* is the label value of the call, and $\textsc{v}$ contains the actual return values. A separate rule takes a completion message *comp(O n out)* into the event queue *EvQ* of the calling object O, thereby enabling guards on a label with value n.

In the following section, we generalize the previously mentioned rewrite rules to probabilistic rewrite rules, also adding new syntax and semantics for a *random assignment* operation.

3 Syntax and Semantics of PCreol

First of all, notice that the entire Creol operational semantics given in [16] can be directly expressed as a probabilistic rewrite theory, where conditional rules of the form

$$\text{subconfig}_1 \ \longrightarrow\ \text{subconfig}_2 \quad \textbf{if} \quad \text{condition},$$

in the original operational semantics, can be translated into probabilistic rewrite rules of the form

$$\text{subconfig}_1 \ \longrightarrow\ \text{subconfig}_2 \quad \textbf{if} \quad \text{condition} \quad \textbf{with probability}\ \ \pi_0, \tag{12}$$

where π_0 denotes a trivial distribution, as defined in Section 2.1. In other words, we are able to extend the Creol interpreter to the probabilistic setting just by adding new rewrite rules, without altering the existing ones, except when we decide to add probabilistic features.

This section provides an overview of the main probabilistic features that extend the syntax and semantics of Creol. Thus, in Section 3.1 we introduce the `random` keyword, which allows generating pseudo-random numbers in Creol.

3.1 Generating Pseudo-Random Numbers

The Creol compiler recognizes the syntax `random(i)`, where `i` is an arbitrary integer. However, the existing version of the Creol interpreter gives this command the following semantics, which is not very practical. The interpreter rewrites terms like `random(i)` to the Maude term with the same syntax, but where the `random` keyword is taken from the `RANDOM` Maude core module. Thus, the previous command returns the `i`-th number in the sequence of pseudo-random numbers generated by Maude's built-in pseudorandom number generator, with respect to the initial random seed given through the `-random-seed` command line parameter. This means that the integer parameter `i` must be incremented each time a new pseudo-random number is needed.

We choose to give different semantics to `random(i)`, namely that it generates a pseudo-random number in the unit interval following an uniform distribution, i.e. all values in $[0, 1)$ have an equal chance to be sampled. Until a new version of the Creol compiler adds support for a special command with this semantics, we use the construct `random(i)` to generate a new pseudo-random number at each call, regardless of the value of `i`.

3.2 Probabilistic Choice Operator

We first consider adding an infix *probabilistic choice* operator $\Box_p$ to the syntactic category of Creol statements. This operator has the syntax $\mathrm{S}_1 \Box_p \mathrm{S}_2$ where $p \in [0, 1]$ is a fixed real value in the unit interval and S_1, S_2 are two arbitrary *lists* of statements. The informal semantics of $\mathrm{S}_1 \Box_p \mathrm{S}_2$ is that, whenever it is encountered throughout the control flow, the list of statements S_1 is selected for execution with probability p, while S_2 is selected with probability $1 - p$. However each list of statements is executed provided that it is *ready*, i.e. if its corresponding process may be waken up, which is checked through the *ready* predicate. The following result emphasizes the differences between this operator and the nondeterministic choice operator $\Box$ in Creol.

Proposition 1. *The probabilistic choice operator $\Box_p$ is not associative. Also, $\Box_p$ is commutative if and only if $p = 0.5$.*

Proof. Let $p, q \in [0, 1]$ be two real numbers and consider S_1, S_2 and S_3, three fixed and arbitrary lists of Creol statements. In order to prove that $\Box_p$ is not associative, it suffices to show that the probability $p_1 \in [0, 1]$ of selecting S_1 in $\mathrm{S}_1 \Box_p (\mathrm{S}_2 \Box_q \mathrm{S}_3)$ is different from the probability $p_2 \in [0, 1]$ of selecting the same list of statements S_1 in $(\mathrm{S}_1 \Box_p \mathrm{S}_2) \Box_q \mathrm{S}_3$. Since the two probabilistic choices $\Box_p$ and $\Box_q$ are independent events, it follows from the axioms o probability theory that $p_1 = p$, while $p_2 = pq$. For any value $q \neq 1$, the two values p_1 and p_2 become different, hence $\Box_p$ is not associative.

To prove the second part of the proposition, notice that the probability of selecting S_1 in $\mathrm{S}_1 \Box_p \mathrm{S}_2$ is p, while the probability of selecting S_1 in $\mathrm{S}_2 \Box_p \mathrm{S}_1$ is $1 - p$. Therefore $\Box_p$ is commutative if and only if $p = 1 - p = 0.5$.

$\Box$

Denote by BERNOULLI(p) an operation that samples from the Bernoulli discrete probability distribution with parameter p, to return the value `true` with probability p and `false` with probability $1 - p$. If both statements in the probabilistic choice are ready for execution, the formal semantics for the $\Box_p$ operator is given by the following probabilistic conditional rewrite rule:

$\langle\ o : Ob \mid Pr : (\mathrm{S}_1 \Box_p \mathrm{S}_2) ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
$\longrightarrow$
if B **then**
  $\langle\ o : Ob \mid Pr : \mathrm{S}_1 ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
else
  $\langle\ o : Ob \mid Pr : \mathrm{S}_2 ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
fi
if $ready(\mathrm{S}_1, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$ **and** $ready(\mathrm{S}_2, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$
with probability $B :=$ BERNOULLI(p)

When only one of the statements is ready for execution, this statement is automatically selected and the suspended one is dropped. This is achieved by simplification with respect to the conditional equations

$\langle\ o : Ob \mid Pr : (\mathrm{S}_1 \Box_p \mathrm{S}_2) ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
$=$
$\langle\ o : Ob \mid Pr : \mathrm{S}_1 ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
if $ready(\mathrm{S}_1, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$ **and not**($ready(\mathrm{S}_2, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$)

for the case when only the first statement is ready, and

$\langle\ o : Ob \mid Pr : (\mathrm{S}_1 \Box_p \mathrm{S}_2) ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
$=$
$\langle\ o : Ob \mid Pr : \mathrm{S}_2 ; \mathrm{S}_3,\ Lvar : \mathrm{L},\ Att : \mathrm{A},\ EvQ : \mathrm{Q}\ \rangle$
if not($ready(\mathrm{S}_1, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$) **and** $ready(\mathrm{S}_2, (\mathrm{A} ; \mathrm{L}), \mathrm{Q})$

when only the second statement is ready. The case when neither one of the statements is ready for execution is not handled, neither through conditional rewrite rules nor via conditional equations. Hence, a probabilistic choice operation is only made as soon as at least one of the statements becomes ready for execution.

In the following, we consider generalizing the probabilistic choice operator, motivated by the need to naturally express random selection of a statement list from a set of statement lists. For example, in order to randomly choose between four assignments $x := 3$, $x := 5$, $x := 7$ and $x := 11$, each with an equal chance of being selected, the binary probabilistic choice operator can be used as follows:

$$x := 3 \quad \Box_{1/4} \quad (x := 5 \quad \Box_{1/3} \quad (x := 7 \quad \Box_{1/2} \quad x := 11)). \tag{13}$$

However, this does not naturally express the fact that the four assignments are selected for execution with the same probability. Instead, the probabilities 1/4, 1/3 and 1/2 in (13) need to be *derived* from the uniform distribution:

$$\begin{pmatrix} 3 & 5 & 7 & 11 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{pmatrix}. \tag{14}$$

A more natural solution is to consider a mixfix *uniform probabilistic choice* operator $\Box_u$ that takes as input a variable number of statement lists and selects either one of them for execution, each with equal probability. Thus, the fair selection statement (13) can more easily be expressed using the $\Box_u$ operator as:

$$x := 3 \quad \Box_u \quad x := 5 \quad \Box_u \quad x := 7 \quad \Box_u \quad x := 11. \tag{15}$$

Restricting to the case when all statements lists are ready for execution, the formal semantics for the $\Box_u$ operator is given by the following conditional rewrite rule

$$\begin{array}{l}\langle\ o : \mathit{Ob} \mid \mathit{Pr} : (\textsc{s}_1 \Box_u \textsc{s}_2 \Box_u \ \ldots\ \Box_u \textsc{s}_n)\,;\textsc{s},\ \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ \longrightarrow \\ \langle\ o : \mathit{Ob} \mid \mathit{Pr} : \mathit{pickUniform}(\textsc{s}_1\,;\textsc{s}_2\,;\ldots\,;\textsc{s}_n)\,;\textsc{s},\ \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ \\ \textbf{if } \mathit{allReady}(\textsc{s}_1\,;\textsc{s}_2\,;\ldots\,;\textsc{s}_n,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})\end{array}$$

where $n \geq 1$ is a fixed, arbitrary positive integer and *pickUniform*(S) denotes an operation that picks one of the statement lists in S, uniformly at random. For brevity, we omit here the operational semantics of *pickUniform*, but mention that it can also be given through a probabilistic rewrite rule. The *allReady* predicate in the rule's condition is a variant of the *ready* predicate and it is true provided that *all* given lists of statements $\textsc{s}_1,\ \textsc{s}_2,\ \ldots,\ \textsc{s}_n$ are ready for execution. In the case when not all statements are ready for execution, we simplify the uniform probabilistic choice statement via the following conditional equation

$$\begin{array}{l}\langle\ o : \mathit{Ob} \mid \mathit{Pr} : (\textsc{s}_1 \Box_u \textsc{s}_2 \Box_u \ \ldots\ \Box_u \textsc{s}_n)\,;\textsc{s},\ \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ = \\ \langle\ o : \mathit{Ob} \mid \mathit{Pr} : \mathit{extractReady}(\textsc{s}_1\ ;\ \textsc{s}_2\ ;\ \ldots\ ;\ \textsc{s}_n)\,;\textsc{s},\ \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ \\ \textbf{if not}(\mathit{allReady}(\textsc{s}_1\,;\textsc{s}_2\,;\ldots\,;\textsc{s}_n,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q}))\ \textbf{and}\ \mathit{ready}(\textsc{s}_i,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})\end{array}$$

where the integer $i \in \{1, 2, \ldots, n\}$ is arbitrary and *extractReady*(SL) is an operation that extracts the $k \geq 1$ statement lists $\textsc{s}_{i_1},\ \textsc{s}_{i_2},\ \ldots,\ \textsc{s}_{i_k}$ that are ready for execution in the given list SL and returns the uniform probabilistic choice statement:

$$\textsc{s}_{i_1} \Box_u \textsc{s}_{i_2} \Box_u \ \ldots\ \Box_u \textsc{s}_{i_k}. \tag{16}$$

Notice that the condition $\mathit{ready}(\textsc{s}_i,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})$ ensures that at least one of the statement lists is ready for execution, in the initial uniform probabilistic choice statement. In the case when there are no statements ready for execution, the process *Pr* is suspended.

The derivation of (13) becomes even more awkward in the case of non-uniform distributions, i.e. when the statement lists have different chances of being selected. To avoid this derivation, we introduce a mixfix *generalized probabilistic choice* operator that takes as input a list of values in $[0, 1]$ summing up to a value in $[0, 1]$, as well as the statement lists whose probabilities of being selected are given by these values. Thus, the binary probabilistic choice operator is generalized to $n \geq 3$ statements, with the syntax

$$\textsc{s}_1 \Box_{p_1} \textsc{s}_2 \Box_{p_2} \ \ldots\ \textsc{s}_{n-1} \Box_{p_{n-1}} \textsc{s}_n \tag{17}$$

for values $p_1, p_2, \ldots, p_{n-1} \in [0, 1]$ such that $\sum_{i=1}^{n-1} p_i \leq 1$. The informal semantics for this operator is a natural generalization both to the binary case $n = 2$ and to the case of uniform random selection, when $p_1 = p_2 = \ldots = p_{n-1} = 1/n$. Thus, whenever an expression of the form (17) is encountered in the control flow, the statement list $\textsc{s}_i$ is selected with probability p_i, for each $i \in \{1, 2, \ldots, n - 1\}$, provided that it is ready for execution. The last statement list $\textsc{s}_n$ is selected for execution with probability $1 - \sum_{i=1}^{n-1} p_i$. Note that using parentheses in (17) to put together two statement lists may cause the binary probabilistic choice operator to be used instead. Therefore, we recommend that an expression involving the generalized probabilistic choice operator should contain no parentheses.

As an example, consider the problem of expressing the random selection from the four assignments considered before $x := 3$, $x := 5$, $x := 7$ and $x := 11$, where this time the probabilities for assigning each of the given values to the variable x are given by the following non-uniform distribution:

$$\begin{pmatrix} 3 & 5 & 7 & 11 \\ 1/6 & 1/3 & 1/6 & 1/3 \end{pmatrix}. \tag{18}$$

In this case, the generalized probabilistic choice operator can be used as follows:

$$x := 3 \quad \Box_{1/6} \quad x := 5 \quad \Box_{1/3} \quad x := 7 \quad \Box_{1/6} \quad x := 11. \tag{19}$$

Again restricting to the case when all statements lists are ready for execution, the operational semantics for this operator is given by the following probabilistic conditional rewrite rule

$\langle\ o : Ob \mid Pr : (\textsc{s}_1 \Box_{p_1} \textsc{s}_2 \Box_{p_2} \ \dots\ \textsc{s}_{n-1} \Box_{p_{n-1}} \textsc{s}_n) \,;\, \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \rangle$
$\longrightarrow$
$\langle\ o : Ob \mid Pr : \mathit{pickRandom}(\textsc{s}_1 \,;\, \textsc{s}_2 \,;\, \dots \,;\, \textsc{s}_n,\ p_1 \,;\, p_2 \,;\, \dots \,;\, p_n,\ U) \,;\, \textsc{s},$
$\quad Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \rangle$

if $\mathit{allReady}(\textsc{s}_1 \,;\, \textsc{s}_2 \,;\, \dots \,;\, \textsc{s}_n,\ (\textsc{a} \,;\, \textsc{l}),\ \textsc{q})$
with probability $U := \text{UNIFORM}(0,\ 1)$

where $n \geq 1$ is a positive integer, $p_n := 1 - \sum_{i=1}^{n-1} p_i$ and $\textsc{uniform}(a,\ b)$ denotes an operation that samples from the uniform probability distribution over the interval $[a, b)$. The *pickRandom* operation takes as parameters a list of n statement lists $\textsc{s}_1 \,;\, \textsc{s}_2 \,;\, \dots \,;\, \textsc{s}_n$, a list of probabilities, i.e. a *distribution* $p_1 \,;\, p_2 \,;\, \dots \,;\, p_n$, as well as a numerical value $U \in [0, 1)$ and returns the statement $\textsc{s}_j$ with the property that $U \in I_j$, where $I_j \subseteq [0, 1)$ is the interval defined through

$$I_j = \left[\sum_{i=1}^{j-1} p_i,\ \sum_{i=1}^{j} p_i\right), \tag{20}$$

for all $j \in \{1, 2, \dots, n\}$. The intuition is that the *pickRandom* operation first divides the unit interval into subintervals of length p_i of the form (20), corresponding to the probability of selecting $\textsc{s}_i$, for each $i \in \{1, 2, \dots, n\}$. Then it returns the statement list corresponding to the subinterval containing the given value U.

Similar to the uniform probabilistic choice operator $\Box_u$, we simplify the generalized probabilistic choice statement, in the case when at least one statement is not ready for execution, with respect to the following conditional equation

$\langle\ o : Ob \mid Pr : (\textsc{s}_1 \Box_{p_1} \textsc{s}_2 \Box_{p_2} \ \dots\ \textsc{s}_{n-1} \Box_{p_{n-1}} \textsc{s}_n) \,;\, \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \rangle$
$=$
$\langle\ o : Ob \mid Pr : \mathit{extractReady}(\textsc{s}_1 \,;\, \textsc{s}_2 \,;\, \dots \,;\, \textsc{s}_n,\ p_1 \,;\, p_2 \,;\, \dots \,;\, p_n) \,;\, \textsc{s},$
$\quad Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \rangle$

if not$(\mathit{allReady}(\textsc{s}_1 \,;\, \textsc{s}_2 \,;\, \dots \,;\, \textsc{s}_n,\ (\textsc{a} \,;\, \textsc{l}),\ \textsc{q}))$ **and** $\mathit{ready}(\textsc{s}_i,\ (\textsc{a} \,;\, \textsc{l}),\ \textsc{q})$

where $i \in \{1, 2, \dots, n\}$ is an arbitrary integer and $p_n := 1 - \sum_{i=1}^{n-1} p_i$. The *extractReady* operation in this equation is a generalization of the one used for uniform probabilistic choice, with the following semantics. It first extracts the $k \geq 1$ statement lists $\textsc{s}_{i_1}$, $\textsc{s}_{i_2}$, $\dots$, $\textsc{s}_{i_k}$ that are ready for execution and returns the generalized probabilistic choice statement

$$\textsc{s}_{i_1} \Box_{p'_{i_1}} \textsc{s}_{i_2} \Box_{p'_{i_2}} \ \dots\ \Box_{p'_{i_{k-1}}} \textsc{s}_{i_k} \tag{21}$$

where

$$p'_{i_r} = \left(\sum_{j=1}^{k} p_{i_j}\right)^{-1} p_{i_r}, \tag{22}$$

for each $r \in \{1, 2, \ldots, k\}$, so that $\sum_{j=1}^{k} p'_{i_j} = 1$ and the axioms of probability theory hold. As in the case of uniform probabilistic choice, the generalized probabilistic choice operation is only made as soon as at least one of the statements becomes ready for execution.

3.3 Random Assignment

We add an *uniform random assignment* operator with the syntax

$$x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n]), \tag{23}$$

that randomly selects an expression from the list $E = [e_1,\, e_2,\, \ldots,\, e_n]$ and assigns it to the specified variable x, where each expression in E has an equal chance of being selected. Thus, the fair selection statement (15) using the uniform probabilistic choice operator $\Box_u$ can be more easily expressed as:

$$x := \text{random}([3,\, 5,\, 7,\, 11]). \tag{24}$$

In order to define the formal semantics of the uniform random assignment operator, we make use of the *pickUniform* operator defined in the previous section:

$$\begin{array}{l}\langle\ o : \mathit{Ob} \mid \mathit{Pr} : (x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n], [p_1,\, p_2,\, \ldots,\, p_n])\ ;\ \textsc{s}, \\ \quad \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ \longrightarrow \\ \langle\ o : \mathit{Ob} \mid \mathit{Pr} : \mathit{pickUniform}(x := e_1\ ;\ x := e_2\ ;\ \ldots\ ;\ x := e_n))\ ;\ \textsc{s}, \\ \quad \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle\end{array}$$

This operator may also be generalized, by considering arbitrary, possibly non-uniform distributions over the list E. The syntax for this *generalized random assignment* is

$$x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n], [p_1,\, p_2,\, \ldots,\, p_n]), \tag{25}$$

where $p_i \in [0, 1]$ denotes the probability of assigning e_i to x, for each $i \in \{1, 2, \ldots, n\}$. Also, by the axioms of probability theory, it must also be the case that $\sum_{i=1}^{n} p_i = 1$. Similar to uniform random assignment, we can use the generalized random assignment in order to express the non-uniform random assignment (19) in a more compact way:

$$x := \text{random}([3,\, 5,\, 7,\, 11], [1/6,\, 1/3,\, 1/6,\, 1/3]). \tag{26}$$

Also, its formal semantics is similar to that of the uniform random assignment operator:

$$\begin{array}{l}\langle\ o : \mathit{Ob} \mid \mathit{Pr} : (x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n], [p_1,\, p_2,\, \ldots,\, p_n])\ ;\ \textsc{s}, \\ \quad \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle \\ \longrightarrow \\ \langle\ o : \mathit{Ob} \mid \mathit{Pr} : \mathit{pickRandom}(x := e_1\ ;\ x := e_2\ ;\ \ldots\ ;\ x := e_n,\ p_1\ ;\ p_2\ ;\ \ldots\ ;\ p_n,\ U)\ ;\ \textsc{s}, \\ \quad \mathit{Lvar} : \textsc{l},\ \mathit{Att} : \textsc{a},\ \mathit{EvQ} : \textsc{q}\ \rangle\end{array}$$

with probability $U := \text{UNIFORM}(0,\ 1)$

Note. At first glance, it may seem that the random assignment operator is a particular case of generalized probabilistic choice, in which the statements are assignments of the form $x := e$, for each $e \in E$ and whose distribution is given by the values $p_1, p_2, \ldots, p_n$. Indeed, the uniform random assignment (23) can be expressed as

$$x := e_1 \quad \Box_u \quad x := e_2 \quad \Box_u \quad \ldots \quad \Box_u \quad x := e_n \tag{27}$$

while the more general (25) can be written:

$$x := e_1 \quad \Box_{p_1} \quad x := e_2 \quad \Box_{p_2} \quad \ldots \quad x := e_{n-1} \quad \Box_{p_{n-1}} \quad x := e_n. \tag{28}$$

However, there is an important difference between the two operators, namely that the random assignment can take as parameter a list E whose value is determined at *execution-time*, while the probabilistic choice can only be given lists of expressions *during implementation*. It also becomes more and more suitable to use the shorter random assignment syntax as the number of expressions in the list E grows larger.

3.4 Lossy Communication and Component Failures

We now pose the problem of modifying the operational semantics of Creol to specify *lossy network communication*. In order to achieve this, the semantics for the *invocation* and *completion* messages need to be extended, so that these messages are sent from one component to another, with the possibility of being lost in the process. Thus, the operational semantics for *lossy communication* is obtained by redefining the rewrite rules for lossless communication and has two cases, depending on the nature of the message. *Lossy invocation* with $\alpha \in [0,1]$ probability of successful message delivery has the following semantics

```
$\langle\ o : Ob \mid Pr : t!x.m(\textsc{e});\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ Lab : n\ \rangle$
$\longrightarrow$
**if** $B$ **then**
  $\langle\ o : Ob \mid Pr : t := n;\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ Lab : next(n)\ \rangle$
  $invoc(eval(\textsc{x},\ (\textsc{a};\ \textsc{l})),\ m,\ (o\ n\ eval(\textsc{e},\ (\textsc{a};\ \textsc{l}))))$
**else**
  $\langle\ o : Ob \mid Pr : \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ Lab : n\ \rangle$
**fi**
**with probability** $B := \text{BERNOULLI}(\alpha)$
```

where n is a new label used to identify the future variable t. Also, *lossy completion* with $\beta \in [0,1]$ probability of successful message delivery is given by the probabilistic rewrite rule:

```
$\langle\ o : Ob \mid Pr : return(\textsc{v});\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle$
$\longrightarrow$
**if** $B$ **then**
  $\langle\ o : Ob \mid Pr : \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle$
  $comp(eval((caller\ label\ \textsc{v}),\ (\textsc{a};\ \textsc{l})))$
**else**
  $\langle\ o : Ob \mid Pr : \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle$
**fi**
**with probability** $B := \text{BERNOULLI}(\beta)$
```

Note: As a generalisation, we consider the case when the probabilities $\alpha, \beta \in [0,1]$ depend on the class of the object making the invocation or sending the completion message. In other words, the syntax for class declarations should be extended to include these parameters, so that all instances of a class C with parameters α_C, β_C are able to successfully deliver invocation messages with probability α_C and they have β_C probability of success when replying with a completion message. A possible use of such a generalisation is in relating the probabilities α and β to the geographical location of the caller and of the callee in an invocation or completion message, considering that this position information is stored in the list of attributes of each of the two objects' classes. For example, in wireless sensor networks, we may argue that the probability of message loss is directly proportional to the distance between the two sensors attempting to communicate.

We may also model *probabilistic object creation*, i.e. the fact that sometimes creating an instance of a class may fail, by adding probabilities to the semantics of the *new* operator. However, this is mostly syntactic sugar since the same probabilistic behaviour can be obtained through a probabilistic choice between a *skip* statement and the statement containing the *new* operator. Denoting by $\gamma \in [0, 1]$ the probability of successfully creating new instances of *any* class, the modified operational semantics of the *new* operator is given through:

$$
\begin{array}{l}
\langle\ O : Ob \mid Pr : (\textsc{v} := new\ C(\textsc{e}); \textsc{s}),\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle \\
\langle\ C : Cl \mid Par : \textsc{v},\ Att : \textsc{a}',\ init : (\textsc{s}', \textsc{l}'),\ Ocnt : n\ \rangle \\
\longrightarrow \\
\textbf{if } B \textbf{ then} \\
\quad \langle\ O : Ob \mid Pr : (\textsc{v} := C\#n; \textsc{s}),\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle \\
\quad \langle\ C\#n : Ob \mid Cl : C,\ Pr : (\textsc{v} := eval(\textsc{e}, (\textsc{a}; \textsc{l})); \textsc{s}'; run, \\
\quad PrQ : \varepsilon,\ Lvar : \textsc{l}',\ Att : self \mapsto C\#n; \textsc{v}; \textsc{a}',\ Lcnt : 1\ \rangle \\
\quad \langle\ C : Cl \mid Par : \textsc{v},\ Att : \textsc{a}',\ init : (\textsc{s}', \textsc{l}'),\ Ocnt : next(n)\ \rangle \\
\textbf{else} \\
\quad \langle\ O : Ob \mid Pr : \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a}\ \rangle \\
\quad \langle\ C : Cl \mid Par : \textsc{v},\ Att : \textsc{a}',\ init : (\textsc{s}', \textsc{l}'),\ Ocnt : n\ \rangle \\
\textbf{fi} \\
\textbf{with probability } B := \text{BERNOULLI}(\gamma)
\end{array}
$$

Note: We only considered the case when γ is a global parameter, no matter the class given as parameter to the *new* operator. However, γ can be made to depend on the given class, by extending the syntax for class declarations to include this parameter, in a similar way as discussed for the case of lossy communication.

4 Implementation

We implemented a prototype PCreol interpreter on top of the current one for Creol, to test part of the features in the operational semantics described in Section 3. The features that we decided to implement and test were the pseudo-random number generation, binary probabilistic choice, lossy communication (invocation, completion) and probabilistic object creation. However, this section is mainly devoted to describing the Maude implementation of the stochastic time model that we use, allowing the VeStA tool to statistically model check and statistically analyze quantitative properties of PCreol programs.

We start with an overview of PMaude and VeStA, describe the implementation of the stochastic time model and finish with examples of PCreol programs, showing how VeStA can be used to analyze them.

4.1 Overview of PMaude and VeStA

The paper [21] introduces PMaude as an interpreter for finitary probabilistic rewrite theories. However, even if having its own homepage [25], where its full source code listing can be found, it seems that the PMaude interpreter is not officially maintained since around 2003 and is not compatible with the latest versions of Maude and Full Maude. This is the reason why we could not provide an implementation for the operational semantics of PCreol, based on PMaude, which was our initial plan. PMaude is reintroduced in [2] as a specification language for general (not just finitary) probabilistic rewrite theories, which are usually not directly executable in Maude. This language allows to express probabilistic, as well as nondeterministic behaviour and its probabilistic conditional rewrite rules have the general form given by (8):

$$t(X) \longrightarrow t'(X, Y) \ \ \textbf{if} \ \ C(X) \ \ \textbf{with probability} \ \ Y := \pi(X).$$

These rules are nondeterministic, as the variables Y in their right-hand side do not appear in the left-hand side, rendering them *nonexecutable* in Maude. However, Maude can be used to *simulate* a PMaude specification, provided that all variables Y, in rules such as the previous one, are replaced with actual values *sampled* from the probability distribution π.

The same paper [2] introduces an Actor PMaude module, that can be used to create executable PMaude specifications, which are free from any source of nondeterminism. This is achieved by considering the current state of the system as a multiset of *actors* and *messages*, in which time is made explicit through a global floating point value. When creating an executable PMaude specification from a nondeterministic one, all rewrite rules in the original specification must be *scheduled* to execute at random moments of time, in the Actor PMaude version, with the interval between two consecutive executions following an exponential probability distribution. Recall that the exponential distribution has cumulative distribution function $P(x) = 1 - \mathrm{e}^{-\lambda x}$, where $\lambda \in \mathbb{R}$ is called the *rate* parameter. We denote by $\mathrm{Exp}(\lambda)$ the exponential probability distribution with rate parameter λ and use the notation $X \sim \mathrm{Exp}(\lambda)$ in order to specify that the random variable X is sampled from $\mathrm{Exp}(\lambda)$.

The VeStA tool ([26], [27], [28]) can be used to generate execution traces from executable PMaude modules based on Actor PMaude, in which all nondeterminism has been resolved. It allows to *statistically model check* these modules against probabilistic temporal formulae expressed in CSL (Continuous Stochastic Logic) [4], giving an alternative to Maude's search and model checking commands. VeStA also allows the quantitative analysis of executable PMaude modules, via quantitative temporal expressions given in the QuaTEx logic [2]. These expressions relate the current state of the system to a numerical quantity, through a formula expressed in Maude. The average value of such an expression is estimated, within a certain confidence interval. Although taking inspiration from Probabilistic Computation Tree Logic (PCTL) [11], QuaTEx is more expressive than the latter.

4.2 A Stochastic Time Model for PCreol

The following paragraphs give precise mathematical meaning to the mechanism that we use to schedule the execution of PCreol objects, also implemented in the Actor PMaude module. We start with a question that forms the basis of our stochastic time model, namely *what defines an interleaving?* To be more precise, consider that we have an object-based model, in which the global configuration of the system is a multiset of objects and messages. As in Creol, consider that each object has a sequence of statements to execute and that it runs on its own microprocessor. In order to simplify notation and put more emphasis on the concept of interleaving, assume that the set $\{O_i\}_{i \in I}$ of all objects in the system's configuration always stays the same, i.e. no objects are created or destroyed at execution time. Propositions 2 and 3 that follow can be generalized to handle the case when the set $\{O_i\}_{i \in I}$ changes.

The execution traces of the concurrent system are represented by interleavings of one-statement executions performed by different objects. A natural way of defining interleavings is by means of a sequence $\{e_n\}_{n \geq 0}$ in which the term $e_n \in I$ represents the index of the object executing at step n in the system's evolution, for each $n \in \mathbb{N}$, and I is a nonempty index set such that $\{O_i\}_{i \in I}$ is the set of all objects in the global configuration. The following result provides an alternative to the previous definition:

Proposition 2. *Interleavings can be specified through sequences of increasingly ordered instants of time when each individual object O_i executes its statements, for each $i \in I$.*

Proof. Consider a function $f : I \to \mathbb{R}^{\mathbb{N}}$ that maps each object index $i \in I$ into the sequence of time values

$$f(i) = \left\{t_1^i < t_2^i < \ldots < t_j^i < \ldots\right\} \subseteq \mathbb{R} \tag{29}$$

containing the time instants when object O_i executes its statements. We require that $f(i) \cap f(i') = \emptyset$, for all $i, i' \in I$ with $i \neq i'$, i.e. no two objects execute at the same instant of time. Let us now determine the union

$$f(I) = \bigcup_{i \in I} f(i) = \left\{ t_1^{i_1} < t_2^{i_2} < \ldots < t_n^{i_n} < \ldots \right\}, \tag{30}$$

where $i_k \in I$ and $t_k^{i_k}$ denotes a term in the sequence $f(i_k)$, for all $k \geq 1$. The sequence $\{i_n\}_{n \geq 0}$ then defines an interleaving, in the usual sense. □

Standard Model checking goes through all possible interleavings of a concurrent program, causing state space explosion. One possible solution to this problem is to obtain a series of random interleavings by discrete-event simulation, i.e. by generating sequences $f(i)$ of random time values, for each $i \in I$, and using the proof of Proposition 2 to construct the associated random interleavings. These can then be used in statistical model checking and statistical quantitative analysis algorithms, as implemented in VeStA.

The next result provides a sufficient condition for the state space corresponding to a concurrent object-based system to be *fairly* checked using statistical model checking. The condition is that the holding times of all objects need to be exponentially distributed with the same rate parameter. Denote by $\mathrm{Exp}(\lambda)^{\mathbb{N}}$ the set of all sequences $\{X_j\}_{j \geq 0}$ of independent and identically distributed random variables such that $X_j \sim \mathrm{Exp}(\lambda)$ for all $j \geq 0$, i.e. all terms in the sequence follow an exponential distribution with rate $\lambda \in \mathbb{R}$.

Proposition 3. *Let $\lambda > 0$ be a fixed positive real number and assume that the holding times of each object O_i follow an exponential distribution with rate parameter λ. Then all objects have an equal chance to execute at all times.*

Proof. Consider a function $g : I \to \mathrm{Exp}(\lambda)^{\mathbb{N}}$ mapping each index $i \in I$ into the sequence of random, exponentially distributed holding times $g(i) = \{X_j^i\}_{j \geq 1} \in \mathrm{Exp}(\lambda)^{\mathbb{N}}$ of object O_i. For each $i \in I$, $g(i)$ uniquely defines the sequence $T(g(i)) := f(i)$ of time values in (29) by setting $t_j^i = \sum_{k=1}^{j} X_j^i$, for each $i, j \geq 0$. Therefore, the function g defines a random interleaving $T(g(I)) := \bigcup_{i \in I} T(g(i))$ as given by equation (30). What we need to prove is that in the sequence of random indices $\{i_n\}_{n \geq 0}$ corresponding to $T(g(I))$, each term follows a discrete uniform distribution over the index set I, i.e. $\Pr(i_n = i) = 1/|I|$ for all $i \in I$ and all $n \geq 0$, where $|A|$ denotes the number of elements of the set A.

Let $n \in \mathbb{N}$ be fixed and arbitrary. Considering that the system is currently at time $t_n^{i_n}$, the holding time T_i of object O_i from this moment on is exponentially distributed with parameter λ, for all $i \in I$. This follows from the memorylessness property of the exponential distribution, i.e. if $X \sim \mathrm{Exp}(\lambda)$ then

$$\Pr(X > x + \alpha \mid X > x) = \Pr(X > \alpha), \tag{31}$$

for all $x, \alpha \in \mathbb{R}$. To prove that all objects have an equal chance to execute after time $t_n^{i_n}$, notice that

$$\Pr(i_{n+1} = i_*) = \Pr(T_{i_*} = \min\{T_i \; ; \; i \in I\}) = \frac{1}{|I|}, \tag{32}$$

for all $i_* \in I$, from the properties of the exponential probability distribution and using the fact that the holding times $\{T_i\}_{i \in I}$ are independent random variables. □

4.3 Implementation of the Stochastic Time Model

We briefly give the implementation details for the stochastic time model described in Section 4.2. Firstly, we need to make `Float` a subsort of the configuration, in order to be able to explicitly specify *time*. Then we define *execution marks* and make them a subsort of the configuration:

```
subsort ExecMark < Configuration .
op execute(_) : Oid -> ExecMark .
```

We also define *scheduled execution marks* and make them a subsort of the configuration:

```
subsort ScheduledExecMark < Configuration .
op [_,_] : Float ExecMark -> ScheduledExecMark .
```

The scheduled and unscheduled execution marks form the main ingredient of our stochastic time model, making it possible to quantify and resolve all nondeterminism in the Creol interpreter. We then add a *tick* operation that makes the system evolve by unwrapping the scheduled execution marks into unscheduled ones and rendering exactly one object active

```
op tick : Config -> Config .
```

where `Config` is a sort whose terms are obtained from terms of sort `Configuration` by adding a pair of curly brackets:

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

This is to ensure compatibility with the VeStA tool. The formal semantics of the *tick* operation is the same as in the Actor PMaude model, selecting the next object for execution in chronological order:

```
op tickAux : Float ExecMark Configuration -> Config .

var CF : Configuration .
vars T1 : Float .
vars E E1 : ExecMark .

eq tick( { [T, E] CF } ) = tickAux(T, E, CF) .
eq tick( { CF } ) = { CF } [owise] .

ceq tickAux(T, E, [T1 , E1] CF) = tickAux(T1, E1, [T , E ] CF) if T1 < T .
eq tickAux(T, E, CF T1) = { E CF T } [owise] .
```

Also to ensure compatibility with VeStA, we add an initial state term, giving the state in which a PCreol program is initially found:

```
op initState : -> Config .
```

The `initState` term needs to be defined for each particular PCreol program. Finally, in order to resolve all nondeterminism, we adjust the implementation of the original Creol interpreter by adding execution marks in the right-hand sides of all rewrite rules. For example, the rewrite rule giving the operational semantics for the *skip* statement is changed from

```
rl
  < O : C | Att: S, Pr: { L | skip ; SL }, PrQ: W, Lcnt: F >
  =>
  < O : C | Att: S, Pr: { L | SL }, PrQ: W, Lcnt: F > .
```

to

```
rl
  < O : C | Att: S, Pr: { L | skip ; SL }, PrQ: W, Lcnt: F >
  execute(O) T
  =>
  < O : C | Att: S, Pr: { L | SL }, PrQ: W, Lcnt: F >
  [T + sampleExpWithRate(1.0), execute(O)] T .
```

by adding an execution mark and also the global time to the right-hand side of the rule, as well as adding a scheduled execution mark to its left-hand side in order to make the new subconfiguration active at a later time, after a random interval of time has passed, following an exponential probability distribution with rate parameter 1. This random value is generated using the `sampleExpWithRate` operation in Maude. The other rewrite rules are adjusted in a similar manner. Note that, in the current implementation the rates corresponding to the holding times of all scheduled execution marks are equal to 1.

In order to fully integrate the new implementation of the Creol interpreter with the VeStA tool, we add an operation giving the current time of the global configuration:

```
op getTime : Config -> Float .
eq getTime( { T CF:Configuration } ) = T .
```

More importantly, we define the predicates and valuations that map the current configuration of the PCreol program into a Boolean value and a floating point value, respectively. Thus, consider a set of atomic propositions $AP = \{ \mathrm{sat}_0, \mathrm{sat}_1, \ldots, \mathrm{sat}_n \}$ and a labeling function $L : S \to 2^{AP}$ mapping each state $s \in S$ of the Creol program into the subset of atomic propositions $L(s) \subseteq AP$ that are true in set s. This labeling function is used in the statistical model checking process of VeStA and is implemented through an operation

```
op sat : Nat Config -> Bool .
```

which allows us to determine $L(s)$ by constructing the set $\{ \mathrm{sat}_i \mid$ `sat(i, s) == true` $\}$. Also, consider $V_1, V_2, \ldots, V_k : S \to \mathbb{R}$ a set of $k \geq 1$ valuation functions. These are implemented through an operation

```
op val : Nat Config -> Float .
```

which gives the value of $V_i(s)$ as `val(i, s)`, for all $1 \leq i \leq k$.

After compiling a Creol program into its corresponding Maude specification, which is executable using the Creol interpreter, the predicates, as well as the valuation functions need to be added to the compiled code and explicitly defined by the user in order to determine various properties of the Creol program, through statistical model checking and statistical quantitative analysis with VeStA. The following section provides examples of how this can be achieved.

4.4 Examples

In the following, we consider an example PCreol program which starts by creating an instance of the `Server` class. In its turn, the `Server` object creates an instance of the `Client` class and then calls its *run* method, which sends a number of 3 invocation messages to the `Client` object, asking for the value of its attribute `x`. This attribute is initialized to a random value in the constructor of the `Client` class. We also count the number of successful invocation-completion pairs in the variable `successes`. The full listing of this example is given below:

```
interface Client
begin
with Any
  op getPos(out res: Float)
end

class Client implements Client
begin
  var x : Float
  op init == x := random(0)
  with Any
    op getPos(out res: Float) == res := x
end

class Server
begin
  var c : Client
  var x : Float
  var i : Int
  var successes : Int

  op init == c := new Client

  op run ==
    if c /= null then
      i := 0;
      successes := 0;
      while i < 3 do
        x := -1;
        c.getPos(;x);
        if x /= -1 then
          successes := successes + 1
        end;
        i := i + 1
      end
    end

end
```

After compiling this PCreol program into its corresponding Maude specification, we are ready to add the initial state term, as well as the predicates and the valuation functions. We define the initial state term to be

```
eq initState = main(classes, "Server", emp) .
```

meaning that the execution of the PCreol program starts with creating an object of class `Server`. We consider a single predicate that becomes `true` provided that the PCreol program terminates. This happens as soon as there are no more scheduled execution marks in the current global configuration. For the initial state, the termination predicate is considered to be `false`. We give the Maude implementation for this predicate:

```
var conf : Configuration .

eq sat(0, { conf [T:Float, execute(O:Oid)] }) = false .
ceq sat(0, { conf }) = false if { conf } == initState .
eq sat(0, { conf }) = true [owise] .
```

The single valuation function that we add to the compiled PCreol program is one which gives the value of the successes attribute of the Server object, stored in the current program configuration:

```
eq val(0, {< O:Oid : "Server" | Att: "successes" |-> int(I:Int),
S:Subst, Pr: P:Process, PrQ: M:MProc, Lcnt: N:Nat > conf}) = float(I:Int) .

eq val(0, {< O:Oid : "Server" | Att: "successes" |-> null,
S:Subst, Pr: P:Process, PrQ: M:MProc, Lcnt: N:Nat > conf}) = 0.0 [owise] .
```

where conf is an arbitrary term of sort Configuration. We are now ready for statistical model checking and statistical quantitative analysis of the PCreol program, using VESTA. We consider model checking the program against the CSL formula $\mathcal{P}_{\geq 0.9}[\Diamond_{<80}\, T]$, where T is the previously defined termination predicate. The meaning of this formula is that the predicate T eventually becomes true in the first 80 units of time, with probability greater than or equal to 0.9. Indeed, this property holds as VESTA returns the result:

```
Result: true
Running time: 20.999 seconds
States sampled: 1917
```

The QUATEX query that we consider asks for the expected value of the previously defined valuation function, i.e. for the value of the successes attribute as soon as the program terminates:

```
value() = if { s.sat(0) } then { s.rval(0) } else # value() fi;
eval E[ value() ];
```

In the case when the probabilities α, β and γ are all equal to 1.0, for lossy communication and probabilistic object creation, i.e. when the program executes under ideal conditions, the value returned by VESTA is 3, meaning that the Server object always manages to receive a completion message from the Client, for each of its invocations. Otherwise, this value would be strictly less than 3.

Let us now consider a more interesting example, also modeling the interaction between a client and a server. This time the first created object is an instance of class Main. This instance, in turn creates a Server and a Client object. The Client class is parameterized by an integer parameter value and a Server class parameter s, representing the server with which the client is going to communicate. The Server class has an attribute v, storing its current value.

As soon as the Client object starts running, it executes a while loop of 10 iterations. At each iteration the client makes a call to the server's add_or_sub method with its argument equal to the value parameter of the Client class. The server responds either by adding or by subtracting its current value with the value parameter sent by the client. A probabilistic choice is made between the two alternatives. In this example we assigned a probability of 0.8 for selecting addition and a probability of $1 - 0.8 = 0.2$ for selecting subtraction. In the following, we give a complete listing of the PCreol program:

```
interface Server
begin
with Any
  op add_or_sub(in value: Int)
end
```

```
interface Client
begin
end

class Server implements Server
begin
  var v : Float
  op init == v := 0
  with Any
    op add_or_sub(in value: Int) ==
      v := v + value [0.8] v := v - value
end

class Client(value : Int, s: Server) implements Client
begin
  var i : Int
  op run ==
    i := 0;
    while i < 10 do
      s.add_or_sub(value;);
      i := i + 1
    end
end

class Main
begin
  var s : Server
  var c : Client

  op init ==
    s := new Server;
    c := new Client(1, s)
end
```

We use a similar QuaTEx query as in the previous example, asking for the expected value of the `Server` object, as soon as the program terminates. VeStA gives the following answer:

```
Result: 6.14
Running time: 82.344 seconds
States sampled: 15500
```

There is a mathematical interpretation for this example, namely that the value stored by the `Server` object when the program terminates is equal to the value of a simple random walk on the integer number line $\mathbb{Z}$, starting at 0 (the value that the `Server` object is initialized with in its constructor) and taking 10 unit steps, where each step is either taken to the right with probability 0.8 (the addition) or to the left with probability 0.2 (the subtraction).

```
Result: true
Running time: 77.931 seconds
States sampled: 16318
```

We also consider another predicate $G : S \to \{\texttt{false}, \texttt{true}\}$ on the set of states of the program that returns `true` provided that the server's value is above -1 and `false` otherwise

```
ceq sat(1, { conf }) = true if val(0, { conf }) > -1.0 .
eq sat(1, { conf }) = false [owise] .
```

where `conf` is an arbitrary term of sort `Configuration`. We used VeStA to statistically model check our program against the CSL formula $\mathcal{P}_{\geq 0.9}[\Diamond\ G]$, with the meaning that the server's value eventually becomes greater than -1 with probability 0.9. Notice that the $\Diamond$ operator that we use in this case is unbounded, i.e. it does not have any time constraints, opposite to the previous example where we restricted the search to the first 80 time units. The result of the statistical model checking is

```
Result: true
Running time: 77.931 seconds
States sampled: 16318
```

which also agrees with the intuition that giving the positive steps in the random walk a probability of 0.8, greater than that of the negative steps, makes it more likely that the value of the random walk after 10 steps is found in the positive part of the interval $[-10, 10]$ than in its negative part.

5 Related Work

The generalized probabilistic choice operator, as well as the random assignment in PCreol take inspiration from similar constructions in a probabilistic version of ProMeLa [14], called ProbMeLa [6]. For instance, our generalized probabilistic choice can be expressed using the **pif** ... **ifp** construction in ProbMeLa. However, PCreol is an object-oriented programming language with additional features like inheritance, future variables and others, which ProbMeLa is lacking. The operational semantics for ProbMeLa is also given in terms of a Markov Decision Process, while we give the operational semantics for PCreol using Probabilistic Rewrite Theories. This represents a more general unifying semantical framework, containing Markov Decision Processes as a subclass [20].

PRISM [22] is another similar programming language that lacks object-orientation features, but comes with powerful probabilistic model checking tools that PCreol is missing, as VeStA is still a prototype probabilistic model checker. The only real attempt to use the advanced probabilistic model checker of PRISM in combination with Maude specifications was made in [12]. However, this never led to an actual implementation that we could use with PCreol. On the other hand, the language of PRISM is less expressive, only allowing the specification of automata-like models, while PCreol allows for more general probabilistic rewrite logic specifications, even if the model checking problem may become undecidable.

6 Conclusions and Future Research

The main contribution of this research report is to introduce PCreol, the first *object-oriented* programming language, based on Creol, which allows the specification of probabilistic open distributed systems. We used the semantic framework of probabilistic rewrite logic to define its operational semantics. Also, we integrated PCreol with the VeStA tool for statistical model checking and statistical quantitative analysis, which allows to check different properties of PCreol programs, as well as to extract particular quantitative information from them. The integration of PCreol with VeStA is achieved by extending the Creol interpreter with the implementation of a stochastic time model taking inspiration from the Actor PMaude model introduced in [2]. While this extension is mainly aimed at PCreol, it can also be used to integrate Creol with VeStA, allowing to use VeStA for statistical model checking instead of Maude's LTL model checking tool. The latter is prone to large running times, due to the state explosion problem associated with highly nondeterministic Creol programs.

An important direction for further research is the exact (numerical) probabilistic model checking of probabilistic distributed object-oriented systems modeled with PCreol. This includes using proof systems based on probabilistic extensions of Hoare logic, as introduced in [11]. Also, the integration of PCreol with a numerical (exact) probabilistic model checker would be of great benefit, even if the only currently available probabilistic model checker for systems modeled in probabilistic rewrite logic is VeStA, which is statistical and not exact. On the other hand, VeStA provides a very general model checking algorithm and alternatives to this algorithm, tailored for particular kinds of models, are worth investigating, as in [19]. Using such alternative algorithms, we may be able to significantly increase the efficiency of VeStA and decrease the running time of model checking PCreol programs.

The stochastic time model that we use allows to specify that different objects have different processor speeds, or that explicit time intervals are associated with `await` statements, during which an object suspends its execution. We plan to include these features in the next version of the PCreol interpreter. Other research directions include further generalizations of the syntax and semantics of PCreol, which we describe in what follows.

A class may not be a perfect implementation of its interface, i.e. it may only meet the assume/guarantee specifications of the interface to a certain extent. This happens when there are probabilistic variables in the interface, modeling random environment factors, e.g. the intensity of light coming from the Sun when modeling a solar panel. Following [10], the set of variables of an interface can be separated into *deterministic* and *probabilistic*. Interfaces containing at least one probabilistic variable are called *probabilistic interfaces*. The classes implementing such probabilistic interfaces, or *probabilistic contracts* as they are called in [10], are said to *satisfy* the contracts to a certain extent, given by a reliability value. The degree of satisfiability is an important concept since no class could claim to fully meet the requirements of its contract, as long as this contract includes random environment variables. The paper [10] creates a general theory of probabilistic contracts, but does not apply this to the context of distributed object-oriented programming. Also, they do not consider randomness in the communication between components, which is part of real life open distributed systems. These may also prove to be interesting research topics.

At the same time, components may also behave randomly, e.g. they may become faulty with time or random factors may affect their internal evolution, like random algorithms in their methods, noise or logical errors in their hardware. In other words, components may behave randomly even on their own, when executing self-calls. A component may therefore not be a perfect instance of its class implementation. To model this kind of random behaviour in the components, we may consider probabilistic extensions at different abstraction levels, as follows. At the *class level*, the assume/guarantee specifications can be extended to include probabilities and they may have a syntax similar to `asum(p)` φ and `guar(q)` ψ, where:

- p is the probability that the class instance *believes* the assumption φ to be true, referring to the environment, before the instance starts running,
- q is the probability with which the class instance guarantees that the predicate ψ becomes true, after it finishes running.

There is a close relation between the quality of the message transport channels, i.e.

the communication network, and the probabilities p and q. These probabilities are also influenced by the possibly faulty behaviour of the component at a lower specification level, as described in what follows.

At the *methods level*, a class may contain methods implementing probabilistic algorithms, e.g. probabilistic primality testing, genetic algorithms, swarm optimization algorithms, etc. in which case the post-conditions of such a method should be extended with probabilities. The pre-conditions should also be extended to the probabilistic setting, since a method can take as input the random output of another method implementing a probabilistic algorithm. To be able to estimate the reliability of such a system, probabilistic extensions to Hoare logic need to be considered, as described in the PhD thesis [11], Chapter 6.

At the *instructions level*, the `prove` assertion command may also be extended with the syntax `prove(r)` α, where r is the probability that the assertion α is correctly checked by the class instance. In this manner, we model the fact that each object's logical unit might be prone to hardware errors.

Acknowledgements

We wish to thank Cristian Prisacariu for fruitful discussions throughout the writing of this research report. We also are indebted to Rudolf Schlatte for constant support on developing the current version of the PCreol interpreter.

References

[1] Ábrahāma, E., Grabe, I., Grünerd, A., Steffen, M., Behavioral interface description of an object-oriented language with futures and promises, presented at *The 19th Nordic Workshop on Programming Theory (NWPT 2007)*, Journal of Logic and Algebraic Programming, Volume 78, Issue 7, pp. 491–518, 2009.

[2] G. Agha, J. Meseguer, K. Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems, in the *Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005)*, Electronic Notes in Theoretical Computer Science, Volume 153, Issue 2, pp. 213–239, 2006.

[3] G. Agha, *Actors: A Model of Concurrent Computations in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[4] Aziz, A., Sanwal, K., Singhal, V., Brayton, R., Verifying Continuous Time Markov Chains, *Computer Aided Verification*, LNCS 1102, pp. 269–276, 1996.

[5] Baier, C., Katoen, J.-P., *Principles of Model Checking*, The MIT Press, 2008.

[6] Baier, C., Ciesinski, F., Größer, M., ProbMeLa: A modeling language for communicating probabilistic systems, in the *Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '04)*, pp. 57–66. IEEE Computer Society Press, Los Alamitos, 2006.

[7] Bouhoula A., Jouannaud, J.-P., Meseguer, J., Specification and proof in membership equational logic, *Theoretical Computer Science*, Volume 236, Issue 1-2, pp. 35–132, 2000.

[8] Brewster, R., Graph Homomorphism Tutorial, *Field's Institute Covering Arrays Workshop*, 2006. Also available online at: `http://www.site.uottawa.ca/~lucia/CA06/TutorialBrewster.pdf`

[9] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., The Maude 2.0 System, in the *Proceedings of Rewriting Techniques and Applications 2003*, LNCS 2706, Springer-Verlag, pp. 76–87, 2003.

[10] Delahaye, B., Caillaud, B., Legay, A., Compositional Reasoning on (Probabilistic) Contracts, in the *Proceedings of the EMSOFT 2009 International Conference on Embedded Software*, Grenoble, France, 2009.

[11] den Hartogs, J., *Probabilistic Extensions of Semantical Models*, PhD thesis, under the supervision of Jaco de Bakker, Vrije Universiteit, Amsterdam, 2002. Available for download from `http://www.win.tue.nl/~jhartog/thesis/`.

[12] Hassen, J.B., Tahar, S., On the numerical verification of probabilistic rewriting systems, in the *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006*, Munich, Germany, pp. 1223–1224, 2006.

[13] Hell, P., Nešetřil, J., *Graphs and Homomorphisms*, Oxford University Press, Oxford, 2004.

[14] Holzmann, G.J., The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2004.

[15] Johnsen, E.B., Owe, O., Torjusen, A. B., Validating Behavioural Component Interfaces in Rewriting Logic, *Fundamenta Informaticae*, Volume 82, IOS Press, pp. 341–359, 2008.

[16] Johnsen, E.B., Owe, O., An asynchronous communication model for distributed concurrent objects, *Software and Systems Modeling*, Volume 6, Issue 1, Springer, pp. 39–58, 2007.

[17] Johnsen, E.B., Owe, O., Arnestad, M., Combining Active and Reactive Behavior in Concurrent Objects, in D. Langmyhr (Ed.): *Proceedings of the Norwegian Informatics Conference (NIK '03)*, Tapir Academic Publisher, pp. 193–204, 2003.

[18] Johnsen, E.B., Blanchette, J.C., Kyas, M, Owe, O., Intra-Object versus Inter-Object: Concurrency and Reasoning in Creol, in the *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2008)*, Electronic Notes in Theoretical Computer Science, Volume 243, pp. 89–103, 2009.

[19] Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N., A Probabilistic Formal Analysis Approach to Cross Layer Optimization in Distributed Embedded Systems, *Formal Methods for Open Object-Based Distributed Systems*, Volume 4468/2007, pp. 285–300, 2007.

[20] Kumar, N., Sen, K., Meseguer, J., Agha, G., Probabilistic Rewrite Theories: Unifying Models, Logics and Tools, *Technical report UIUCDCS-R-2003-2347*, University of Illinois at Urbana-Champaign, 2003.

[21] Kumar, N., Sen, K., Meseguer, J., Agha, G., A Rewriting Based Model for Probabilistic Distributed Object Systems, in the *Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2003)*, Paris, France, LNCS 2884, Springer-Verlag, pp. 32–46, 2003.

[22] Kwiatkowska, M, Norman, G., Parker, D., PRISM: Probabilistic Model Checking for Performance and Reliability Analysis, *ACM SIGMETRICS Performance Evaluation Review*, Volume 36, Issue 4, pp. 40–45, 2009.

[23] Meseguer, J., Membership algebra as a logical framework for equational specification, in F. Parisi-Presicce (Ed.): *Proceedings of WADT '97*, LNCS 1376, Springer, pp. 18–61, 1998.

[24] Parrow, J., Björn, V., The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes, in the *Proceedings of LICS '98*, 1998.

[25] The PMaude interpreter homepage, `http://maude.cs.uiuc.edu/pmaude/pmaude.html`

[26] Sen, K., Viswanathan, M., Agha, G., Statistical Model Checking of Black-Box Probabilistic Systems, *16th Conference on Computer Aided Verification (CAV '04)*, LNCS 3114, pp. 399–401, 2004.

[27] Sen, K., Viswanathan, M., Agha, G., On Statistical Model Checking of Stochastic Systems, *Computer Aided Verification*, LNCS 3576, pp. 266–280, 2005.

[28] Sen, K., Viswanathan, M., Agha, G., VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems, *Second International Conference on the Quantitative Evaluation of Systems (QEST '05)*, 2005.

[29] Tribastone, M., Duguid, A., Gilmore, S., The PEPA Eclipse Plug-in, *ACM SIGMETRICS Performance Evaluation Review*, Volume 36, Issue 4, pp. 28–33, 2009.

Appendix

A Probability Theory and Stochastic Processes

Given a set Ω, a *σ-algebra* on Ω is a set $\mathcal{F} \subseteq 2^{\Omega}$ containing the empty set and which is closed under complementation and finite or countably infinite union, where 2^{Ω} denotes the power set of Ω. The pair $(\Omega, \mathcal{F})$ is also called a *measurable space* and the elements of $\mathcal{F}$ are known as *measurable sets*. Given two measurable spaces $(\Omega, \mathcal{F})$ and $(\Psi, \mathcal{X})$, a function $f : \Omega \to \Psi$ is said to be *measurable* if $f^{-1}(X) \in \mathcal{F}$ for any $X \in \mathcal{X}$, i.e. if the preimage of any measurable set under f is measurable.

Let $\mathcal{F}$ be a σ-algebra over Ω. A function $\mu : \mathcal{F} \to [-\infty, +\infty]$ is called a *measure* provided that: *i)* $\mu(\emptyset) = 0$, *ii)* $\mu(F) \geq 0$ for all $F \in \mathcal{F}$ and *iii)* $\mu\left(\cup_{i\in I} A_i\right) = \sum_{i\in I} \mu(F_i)$, for all finite or countably infinite collections $\{F_i\}_{i\in I} \subseteq \mathcal{F}$ of pairwise disjoint sets. The triple $(\Omega, \mathcal{F}, \mu)$ is then called a *measure space*. If $\mu : \mathcal{F} \to [0, \infty]$, $(\Psi, \mathcal{X})$ is another measurable space and $f : \Omega \to \Psi$ is a measurable function, we define the *push-forward* of μ to be the measure $\mu \circ f^{-1} : \mathcal{X} \to [0, \infty]$.

A *probability space* is a measure space with the measure function satisfying $\mu(\Omega) = 1$; in this case it suffices for the range of μ to be the unit interval $[0, 1]$ instead of the extended real number line $[-\infty, \infty]$. The measure $\mu : \mathcal{F} \to [0, 1]$ of a probability space is called a *probability measure* and we denote it by P instead of μ. Also, the set Ω is called the *sample space* and the elements of $\mathcal{F}$ are known as *events*. Two events $F_1, F_2 \in \mathcal{F}$ are said to be *independent* if $P(F_1 \cap F_2) = P(F_1)P(F_2)$.

A *random variable* is a measurable function f from a probability space $(\Omega, \mathcal{F}, P)$ to a measurable space $(\Psi, \mathcal{X})$, also known as the *observation space* of the variable. The *probability distribution* of f is the push-forward measure $P \circ f^{-1} : \mathcal{X} \to [0, 1]$ and allows us to measure the probability that the value of f falls inside a set $X \in \mathcal{X}$. For example, assuming that for some $\psi \in \Psi$, the singleton $\{\psi\}$ is in $\mathcal{X}$, the probability that f takes the value ψ is given by $P \circ f^{-1}(\psi) = P(\{\omega \in \Omega\,;\, f(\omega) = \psi\})$, which we also denote by $P(f = \psi)$. The probability distribution of f is said to be *discrete* provided that $\sum_{\psi\in\Psi} P(f = \psi) = 1$, in which case f is called a *discrete random variable*. Thus, it suffices to know the probabilities of f taking each particular value $\psi \in \Psi$ in order to know its distribution. This suggests introducing the *probability mass function* of f as the function denoted by $\mathsf{pmf}_f : \Psi \to [0, 1]$ and defined through $\mathsf{pmf}_f(\psi) := P(f = \psi)$, for all $\psi \in \Psi$.

A *stochastic process* is a collection of random variables $\{X_t : \Omega \to \Psi\}_{t\in T}$ indexed by a set T of *time instants*. A *discrete time stochastic process* is one such that the set T is discrete, i.e. T is either finite or countably infinite. A *discrete time Markov process* is a discrete time stochastic process $\{X_i : \Omega \to S\}_{i\geq 0}$, with S a finite or countably infinite set of *states*, satisfying the *Markov property*

$$P(\ X_{i+1} = s_{i+1}\ \mid\ X_i = s_i,\ \ldots,\ X_0 = s_0\ ) = P(\ X_{i+1} = s_{i+1}\ \mid\ X_i = s_i\ ), \tag{33}$$

for all $i \geq 0$ and all $s_0, s_1, \ldots, s_i \in S$. This property says that the value of the process at time $i + 1$ only depends on its value at time i and not on its other previous values, i.e. we may say that a Markov process is *memoryless*. The Markov process $\{X_i\}_{i\geq 0}$ is said to be *time homogeneous* provided that

$$P(\ X_{i+1} = s'\ \mid\ X_i = s\ ) = P(\ X_i = s'\ \mid\ X_{i-1} = s\ ), \tag{34}$$

for all $s, s' \in S$ and all $i \geq 0$; otherwise it is called *time non-homogeneous*. The dynamics of time-homogeneous discrete time Markov processes can be captured through a stochastic matrix containing the transition probabilities for each pair of states. Thus, provided that the set of states is $S = \{s_i\}_{i\in I}$, a *transition matrix* for the Markov process

$\{X_i\}_{i\geq 0}$ is a stochastic matrix $(p_{uv})_{u,v\in I}$ such that

$$P(\ X_{i+1} = s_v \ \mid \ X_i = s_u\ ) = p_{uv}, \tag{35}$$

for all $u, v \in I$ and all $i \geq 0$.

Let A be a nonempty set, whose elements $a \in A$ are called *actions* and consider a function $\alpha : S \to 2^A$ that maps each state $s \in S$ of a process to a subset of *admissible actions* $\alpha(s) \subseteq A$ which can be taken when the process is in state s. A *discrete time Markov decision process* (MDP) can be seen as a collection of discrete time Markov processes $\{\, M_A\}_{A\in\mathcal{A}}$ with $M_A = \left\{\, X_i^A : \Omega \to S\,\right\}_{i\geq 0}$ and where the index set $\mathcal{A}$ contains all sequences of admissible actions, i.e. sequences of the form $A = \{\, a_i \in \alpha(X_i^A)\,\}_{i\geq 0}$. The elements of $\mathcal{A}$ are also called *policies*.

B Further Questions

1. **Question:** The method suggested in [15] approximates the black-box behaviour of an individual component through its corresponding communication history. Are there other ways of describing black-box behaviour?

   Answer: Possibly, the following are a few possibilities:

   Hidden Markov Models: Instead of using the communication history of a component, rather determine the Hidden Markov Model describing the component's behaviour, i.e. find the transition probabilities of the component going from one state to another. This alternative may give more, quantitative insight into the internal behaviour of a component than just the qualitative information stored in its communication history. Instead of rewriting logic, it may be possible to use linear algebra to study properties of open distributed systems, through their associated Hidden Markov Model.

   In this Markovian setting, it may also be possible to use Probabilistic Rewrite Theories and Probabilistic Model Checking, along with their Maude implementations, known as PMaude ([25]).

   Constraint Satisfaction Problems: The assume-guarantee specifications may be modeled as Constraint Satisfaction Problems (CSPs). In case the variables (the input to some component) belong to a finite set, search methods like backtracking, local search, etc. can be used. When the input is taken from an infinite set, methods like the simplex algorithm are used to find a solution to the constraint satisfaction problem, also known as a feasibility problem.

   It is worth noting that any CSP can also be seen as a graph homomorphism problem, as described e.g. in [8]. Thus, the input received from a component is modeled as a colored digraph G and the assumption conditions for the component receiving the input are modeled as a graph H. The fact that the input satisfies the assumptions is then equivalent to $G \longrightarrow H$, meaning that there exists a homomorphism (an edge preserving map) from G to H.

   In [8], necessary conditions are also given for a homomorphism to exist between two graphs, in terms of graph invariants (like the chromatic number, the odd girth, etc.). These conditions can be used to quickly check if the two graphs are not homomorphic, to prevent further in-depth checking. In [13] it is proved that the graph homomorphism problem is polynomial time solvable only when H is either bipartite or contains a loop, otherwise it is NP-complete. This gives further information on the complexity of checking whether an open distributed system of software components is able to function properly.

2. **Question:** Are there other ways of describing the communication between components?

   Answer: There might be other ways, for example:

   Graph Rewriting: All the invocation and completion messages sent from one component to another, at a particular moment of time, can be captured in a directed graph. The execution of the entire open distributed system can thus be modeled using graph transformations, also known as graph rewriting. If the environment is considered to be random, further insight into the system's behaviour may be gained using random graph rewriting and the general theory of random graphs.

   Fusion Calculus: The Fusion Calculus introduced in [24] may be used to model communication between two components as a *fusion*. This raises the question whether the communication inside an open distributed system can be made symmetric. Does the definition of bisimulation in Fusion Calculus give us more insight into the behaviour of open distributed systems?

   It may also be worth investigating how the π-calculus can model the interaction between components in a distributed setting.

3. **Question:** How can a probabilistic communication network be modeled?

   Answer: Perhaps the simplest model is to consider probabilistic remote method invocations and replies. Thus, a *random delay* can be associated with each message and notations like $o \xrightarrow{\mathcal{D}} o'$ would denote the fact that the message invocation delay is distributed according to the probability distribution $\mathcal{D}$. Similarly, the notation $o \xleftarrow{\mathcal{D}} o'$ means that the reply from o' to o has a random delay following the probability distribution $\mathcal{D}$. By allowing the random delay to take infinite values, we are able to model the fact that some messages are lost along the communication channels. Also, the random shuffling of the messages when being sent from o to o', through the communication channel, is induced by the decreasing ordering of their associated random delays. To model the fact that a channel connecting o and o' disappears entirely, we may assign infinite values to the delays of any message exchanged between these components.

   The probability distribution of the delay in sending a reply message from o' to o may provide some information on what the reasonable time-out values of o should be. Time-outs and race conditions in open distributed systems have been investigated in [16], Section 9.