A linear process-algebraic format for probabilistic systems with data

Joost-Pieter Katoen[†*], Jaco van de Pol*, Mariëlle Stoelinga* and Mark Timmer*

*Formal Methods and Tools, Faculty of EEMCS
University of Twente, The Netherlands
{vdpol, marielle, timmer}@cs.utwente.nl

[†]Software Modeling and Verification
RWTH Aachen University, Germany
katoen@cs.rwth-aachen.de

Abstract—This paper presents a novel linear process-algebraic format for probabilistic automata. The key ingredient is a symbolic transformation of probabilistic process algebra terms that incorporate data into this linear format while preserving strong probabilistic bisimulation. This generalises similar techniques for traditional process algebras with data, and — more importantly — treats data and data-dependent probabilistic choice in a fully symbolic manner, paving the way to the symbolic analysis of parameterised probabilistic systems.

Keywords-probabilistic process algebra, linearisation, data-dependent probabilistic choice, symbolic transformations

I. INTRODUCTION

Efficient model checking algorithms exist, supported by powerful software tools, for verifying qualitative and quantitative properties for a wide range of probabilistic models. These techniques are applied in areas like security, randomised distributed algorithms, systems biology, and dependability and performance analysis. Major deficiencies of probabilistic model checking are the *state explosion problem* and the restricted treatment of *data*.

As opposed to process calculi like μCRL [1] and E-LOTOS, which support rich data types, the treatment of data in modelling formalisms for probabilistic systems is mostly neglected. Instead, the focus has been on understanding random phenomena and the interplay between randomness and nondeterminism. Data is treated in a restricted manner: probabilistic process algebras typically allow a random choice over a fixed distribution, and input languages for model checkers such as the reactive module language of PRISM [2] or the probabilistic variant of Promela [3] only support basic data types, but neither support more advanced data structures or *parameterised* (i.e., state-dependent) random choice. To model realistic systems, however, convenient means for data modelling are indispensable.

Although parameterised probabilistic choice is semantically well-defined [4], the incorporation of data yields a significant increase of, or even an infinite, state space. Aggressive abstraction techniques for probabilistic models (e.g., [5], [6], [7], [8], [9]) obtain smaller models at the

model level, but the successful analysis of data requires *symbolic* reduction techniques. Such methods reduce stochastic models using syntactic transformations at the *language level*, minimising state spaces *prior to* their generation while preserving functional and quantitative properties. Other approaches that partially deal with data are probabilistic CEGAR ([10], [11]) and the probabilistic GCL [12].

Our aim is to develop symbolic minimisation techniques — operating at the syntax level — for data-dependent probabilistic systems. The starting point for our work is laid down in this paper. We define a probabilistic variant of the process-algebraic μCRL language [1], named prCRL, which treats data as first-class citizens. The language prCRL contains a carefully chosen minimal set of basic operators, on top of which syntactic sugar can be defined easily, and allows data-dependent probabilistic branching. To enable symbolic reductions, we provide a two-phase algorithm to transform prCRL terms into LPPEs: a probabilistic variant of *linear process equations* (LPEs) [13], which is a restricted form of process equations akin to the Greibach normal form for string grammars. We prove that our transformation is correct, in the sense that it preserves strong probabilistic bisimulation [14]. Similar linearisations have been provided for plain μCRL [15] and a real-time variant thereof [16].

To motivate the expected advantage of a probabilistic linear format, we draw an analogy with the purely functional case. There, LPEs have provided a uniform and simple format for a process algebra with data. As a consequence of this simplicity, the LPE format was essential for theory development and tool construction. It lead to elegant proof methods, like the use of invariants for process algebra [13], and the cones and foci method for proof checking process equivalence ([17], [18]). It also enabled the application of model checking techniques to process algebra, such as optimisations from static analysis [19] (including dead variable reduction [20]), data abstraction [21], distributed model checking [22], symbolic model checking (either with BDDs [23] or by constructing the product of an LPE and a parameterised μ-calculus formula ([24], [25])), and confluence reduction [26] (a form of partial-order reduction).

In all these cases, the LPE format enabled a smooth theoretical development with rigorous correctness proofs (often checked in PVS), and a unifying tool implementation,
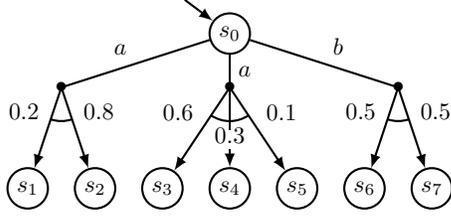
Figure 1. A probabilistic automaton.

enabling the cross-fertilisation of the various techniques by composing them as LPE-LPE transformations.

To demonstrate the whole process of going from prCRL to LPPE and applying reductions to this LPPE, we discuss a case study of a leader election protocol.

We refer the reader to [27] for the extended version of the current paper, which includes proofs of all theorems and propositions and more details about the case study.

II. PRELIMINARIES

Let S be a finite set, then $\mathscr{P}(S)$ denotes its *powerset*, i.e., the set of all its subsets, and $\mathrm{Distr}(S)$ denotes the set of all *probability distributions* over S, i.e., all functions $\mu \colon S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$. If $S' \subseteq S$, let $\mu(S')$ denote $\sum_{s \in S'} \mu(s)$. For the injective function $f \colon S \to T$, let $\mu_f \in \mathrm{Distr}(T)$ such that $\mu_f(f(s)) = \mu(s)$ for all $s \in S$. We use $\{*\}$ to denote a singleton set with a dummy element, and denote vectors and sets of vectors in bold.

A. Probabilistic automata

Probabilistic automata (PAs) are similar to labelled transition systems (LTSs), except that the transition function relates a state to a set of pairs of actions and distribution functions over successor states [28].

Definition 1. *A probabilistic automaton (PA) is a tuple $\mathcal{A} = \langle S, s^0, A, \Delta \rangle$, where*

- *S is a finite set of states, of which s^0 is initial;*
- *A is a finite set of actions;*
- *$\Delta \colon S \to \mathscr{P}(A \times \mathrm{Distr}(S))$ is a transition function.*

When $(a, \mu) \in \Delta(s)$, we write $s \xrightarrow{a} \mu$. This means that from state s the action a can be executed, after which the probability to go to $s' \in S$ equals $\mu(s')$.

Example 1. Figure 1 shows an example PA. Observe the nondeterministic choice between actions, after which the next state is determined probabilistically. Note that the same action can occur multiple times, each time with a different distribution to determine the next state. For this PA we have $s_0 \xrightarrow{a} \mu$, where $\mu(s_1) = 0.2$ and $\mu(s_2) = 0.8$, and $\mu(s_i) = 0$ for all other states s_i. Also, $s_0 \xrightarrow{a} \mu'$ and $s_0 \xrightarrow{b} \mu''$, where μ' and μ'' can be obtained similarly.

B. Strong probabilistic bisimulation

Strong probabilistic bisimulation [14] is a probabilistic extension of the traditional notion of bisimulation introduced by Milner [29], equating any two processes that cannot be distinguished by an observer. Two states s, t of a PA $\mathcal{A}$ are strongly probabilistic bisimilar (denoted by $s \approx t$) if there exists an equivalence relation $R \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ such that $(s, t) \in R$, and for all $(p, q) \in R$ and $p \xrightarrow{a} \mu$ there is a transition $q \xrightarrow{a} \mu'$ such that $\mu \sim_R \mu'$. Here, $\mu \sim_R \mu'$ is defined as $\forall C \, . \, \mu(C) = \mu'(C)$, with C ranging over the equivalence classes of states modulo R.

C. Isomorphism

Two states s and t of a PA $\mathcal{A} = \langle S, s^0, A, \Delta \rangle$ are *isomorphic* (which we denote by $s \equiv t$) if there exists a bijection $f \colon S \to S$ such that $f(s) = t$ and $\forall s' \in S, \mu \in \mathrm{Distr}(S), a \in A \, . \, s' \xrightarrow{a} \mu \Leftrightarrow f(s') \xrightarrow{a} \mu_f$. Obviously, isomorphism implies strong probabilistic bisimulation.

III. A PROCESS ALGEBRA WITH PROBABILISTIC CHOICE

A. The language prCRL

We add a probabilistic choice operator to a restriction of full μCRL [1], obtaining a language called prCRL. We assume an external mechanism for the evaluation of expressions (e.g., equational logic), able to handle at least boolean expressions and real-valued expressions. Also, we assume that all closed expressions can be evaluated. Note that this restricts the expressiveness of the data language. Let Act be a countable set of actions.

Definition 2. *A process term in prCRL is any term that can be generated by the following grammar.*

$$p ::= Y(\boldsymbol{t}) \mid c \Rightarrow p \mid p + p \mid \sum_{x:D} p \mid a(\boldsymbol{t}) \sum_{x:D} f : p$$

Here, Y is a process name, c a boolean expression, $a \in$ Act a (parameterised) atomic action, f a real-valued expression yielding values in $[0, 1]$ (further restricted below), $\boldsymbol{t}$ a vector of expressions, and x a variable ranging over type D. We write $p = p'$ for syntactically identical process terms.

A process equation is an equation of the form $X(\boldsymbol{g} : \boldsymbol{G}) = p$, where $\boldsymbol{g}$ is a vector of global variables and $\boldsymbol{G}$ a vector of their types, and p is a process term in which all free variables are elements of $\boldsymbol{g}$; $X(\boldsymbol{g} : \boldsymbol{G})$ is called a process with process name X and right-hand side p. To obtain unique solutions, indirect (or direct) unguarded recursion is not allowed. Moreover, every construct $\sum_{x:D} f$ in a right-hand side p should comply to $\sum_{d \in D} f[x := d] = 1$ for every possible valuation of the variables in p (the summation now used in the mathematical sense). A prCRL specification is a set of process equations $X_i(\boldsymbol{g_i} : \boldsymbol{G_i}) = p_i$ such that all X_i are named differently, and for every process instantiation $Y(\boldsymbol{t})$ occurring in some p_i there exists a process equation $Y(\boldsymbol{g_i} : \boldsymbol{G_i}) = p_i$ such that $\boldsymbol{t}$ is of type $\boldsymbol{G_i}$.

Table I
SOS RULES FOR PRCRL.

$$\text{INST } \frac{p[\boldsymbol{g} := \boldsymbol{t}] \xrightarrow{\alpha} \mu}{Y(\boldsymbol{t}) \xrightarrow{\alpha} \mu} \text{ if } Y(\boldsymbol{g} : \boldsymbol{G}) = p \qquad\qquad \text{IMPLIES } \frac{p \xrightarrow{\alpha} \mu}{c \Rightarrow p \xrightarrow{\alpha} \mu} \text{ if } c \text{ holds}$$

$$\text{NCHOICE-L } \frac{p \xrightarrow{\alpha} \mu}{p + q \xrightarrow{\alpha} \mu} \qquad \text{NSUM } \frac{p[x := d] \xrightarrow{\alpha} \mu}{\sum_{x:D} p \xrightarrow{\alpha} \mu} \text{ for any } d \in D \qquad \text{NCHOICE-R } \frac{q \xrightarrow{\alpha} \mu}{p + q \xrightarrow{\alpha} \mu}$$

$$\text{PSUM } \frac{-}{a(\boldsymbol{t})\sum_{x:D} f : p \xrightarrow{a(t)} \mu} \text{ where } \forall d \in D . \ \mu(p[x := d]) = \sum_{\substack{d' \in D \\ p[x:=d]=p[x:=d']}} f[x := d']$$

The initial process *of a specification* P *is an instantiation* $Y(\boldsymbol{t})$ *such that there exists an equation* $Y(\boldsymbol{g} : \boldsymbol{G}) = p$ *in* P, $\boldsymbol{t}$ *is of type* $\boldsymbol{G}$, *and* $Y(\boldsymbol{t})$ *does not contain any free variables.*

In a process term, $Y(\boldsymbol{t})$ denotes *process instantiation* (allowing recursion). The term $c \Rightarrow p$ is equal to p if the *condition* c holds, and cannot do anything otherwise. The $+$ operator denotes *nondeterministic choice*, and $\sum_{x:D} p$ a (possibly infinite) *nondeterministic choice over data type* D. Finally, $a(\boldsymbol{t})\sum_{x:D} f : p$ performs the action $a(\boldsymbol{t})$ and then does a *probabilistic choice* over D. It uses the value $f[x := d]$ as the probability of choosing each $d \in D$. We do not consider process terms of the form $p \cdot p$ (where $\cdot$ denotes sequential composition), because this would significantly increase the difficulty of linearisation as it requires using a stack [16]. Moreover, most specifications used in practice can be written without this form.

The operational semantics of prCRL is given in terms of PAs. The states are closed process terms, the initial state is the initial process, the action set is Act, and the transition relation is the smallest relation satisfying the SOS rules in Table I. Here, $p[x := d]$ is used to denote the substitution of all occurrences of x in p by d. Similarly, $p[\boldsymbol{x} := \boldsymbol{t}]$ denotes the substitution of every $\boldsymbol{x}(i)$ in p by $\boldsymbol{t}(i)$. For brevity, we use α to denote an action name together with its parameters. A mapping to PAs is only provided for processes without any free variables; this is consistent with Definition 2.

Proposition 1. *The SOS-rule* PSUM *defines a probability distribution* μ.

Example 2. The following process equation models a system that continuously writes data elements of the finite type D randomly. After each write, it beeps with probability 0.1. Recall that $\{*\}$ denotes a singleton set with an anonymous element. We use it here since the probabilistic choice is trivial and the value of j is never used.

$$B() = \tau()\sum_{d:D} \frac{1}{|D|} : \text{send}(d) \sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.1 \text{ else } 0.9) :$$
$$(i = 1 \Rightarrow \text{beep}() \sum_{j:\{*\}} 1.0 : B()) + (i \neq 1 \Rightarrow B())$$

B. Syntactic sugar

For notational ease we define some syntactic sugar. Let X be a process name, a an action, p, q two process terms, c a condition, and $\boldsymbol{t}$ an expression vector. Then, we write X as an abbreviation of $X()$, and a for $a()$. Moreover,

$$p \triangleleft c \triangleright q \qquad \overset{\text{def}}{=} \qquad (c \Rightarrow p) + (\neg c \Rightarrow q)$$
$$a(\boldsymbol{t}) \cdot p \qquad \overset{\text{def}}{=} \qquad a(\boldsymbol{t})\sum_{x:\{*\}} 1.0 : p$$
$$a(\boldsymbol{t}) \, \boldsymbol{U}_{d:D} \, c \Rightarrow p \qquad \overset{\text{def}}{=} \qquad a(\boldsymbol{t})\sum_{d:D} f : p$$

where x does not occur in p and f is the function 'if c then $\frac{1}{|\{e \in D | c[d := e]\}|}$ else 0'. Note that $\boldsymbol{U}_{d:D} \, c \Rightarrow p$ is the uniform choice among a set, choosing only from its elements that fulfil a certain condition c.

For finite probabilistic sums that do not depend on data,

$$a(\boldsymbol{t})(u_1 : p_1 \oplus u_2 : p_2 \oplus \cdots \oplus u_n : p_n)$$

is used to abbreviate $a(\boldsymbol{t})\sum_{x:\{1,\dots,n\}} f : p$ with $f[x := i] = u_i$ and $p[x := i] = p_i$ for all $1 \leq i \leq n$.

Example 3. The process equation of Example 2 can now be represented as follows:

$$B = \tau\sum_{d:D} \frac{1}{|D|} : \text{send}(d)(0.1 : \text{beep} \cdot B \oplus 0.9 : B)$$

Example 4. Let X continuously send an arbitrary element of some type D that is contained in a finite set Set_D, according to a uniform distribution. It is represented by

$$X(s : \text{Set}_D) = \tau \, \boldsymbol{U}_{d:D} \text{contains}(s, d) \Rightarrow \text{send}(d) \cdot X(s),$$

where $\text{contains}(s, d)$ is assumed to hold when s contains d.

IV. A LINEAR FORMAT FOR PRCRL

A. The LPE and LPPE formats

In the non-probabilistic setting, LPEs are given by the following equation [16]:

$$X(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \cdot X(\boldsymbol{n_i}),$$

where $\boldsymbol{G}$ is a type for *state vectors* (containing the global variables), I a set of *summand indices*, and $\boldsymbol{D_i}$ a type for *local variable vectors* for summand i. The summations

represent nondeterministic choices; the outer between different summands, the inner between different possibilities for the local variables. Furthermore, each summand i has an *action* a_i and three expressions that may depend on the state $\boldsymbol{g}$ and the local variables $\boldsymbol{d_i}$: the *enabling condition* c_i, *action-parameter vector* $\boldsymbol{b_i}$, and *next-state vector* $\boldsymbol{n_i}$.

Example 5. Consider a system consisting of two buffers, B_1 and B_2. Buffer B_1 reads a message of type D from the environment, and sends it synchronously to B_2. Then, B_2 writes the message. The following LPE has exactly this behaviour when initialised with $a = 1$ and $b = 1$.

$$X(a : \{1, 2\}, b : \{1, 2\}, x : D, y : D) =$$

$$
\begin{array}{llll}
\sum_{d:D} & a = 1 & \Rightarrow \mathrm{read}(d) \cdot X(2, b, d, y) & (1) \\
+ & a = 2 \wedge b = 1 & \Rightarrow \mathrm{comm}(x) \cdot X(1, 2, x, x) & (2) \\
+ & b = 2 & \Rightarrow \mathrm{write}(y) \cdot X(a, 1, x, y) & (3)
\end{array}
$$

Note that the first summand models B_1's reading, the second the inter-buffer communication, and the third B_2's writing. The global variables a and b are used as program counters for B_1 and B_2, and x and y for their local memory.

As our intention is to develop a linear format for prCRL that can easily be mapped onto PAs, it should follow the concept of nondeterministically choosing an action and probabilistically determining the next state. Therefore, a natural adaptation is the format given by the following definition.

Definition 3. *An* LPPE *(linear probabilistic process equation) is a prCRL specification consisting of one process equation, of the following format:*

$$X(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i : X(\boldsymbol{n_i})$$

Compared to the LPE we added a probabilistic choice over an additional vector of local variables $\boldsymbol{e_i}$. The corresponding probability distribution expression f_i, as well as the next-state vector $\boldsymbol{n_i}$, can now also depend on $\boldsymbol{e_i}$. An initial process $X(\boldsymbol{v})$ is represented by its *initial vector* $\boldsymbol{v}$, and g^0 is used to refer to the initial value of global variable g.

B. Operational semantics

As the behaviour of an LPPE is uniquely determined by its global variables, the states of the underlying PA are precisely all vectors $\boldsymbol{g'} \in \boldsymbol{G}$ (with the initial vector as initial state). From the SOS rules it follows that for all $\boldsymbol{g'} \in \boldsymbol{G}$, there is a transition $\boldsymbol{g'} \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if for at least one summand i there is a choice of local variables $\boldsymbol{d_i'} \in \boldsymbol{D_i}$ such that

$$c_i(\boldsymbol{g'}, \boldsymbol{d_i'}) \wedge a_i(\boldsymbol{b_i}(\boldsymbol{g'}, \boldsymbol{d_i'})) = a(\boldsymbol{q}) \wedge \forall \boldsymbol{e_i'} \in \boldsymbol{E_i} .$$

$$\mu(\boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d_i'}, \boldsymbol{e_i'})) = \sum_{\substack{\boldsymbol{e_i''} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d_i'}, \boldsymbol{e_i'}) = \boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d_i'}, \boldsymbol{e_i''})}} f_i(\boldsymbol{g'}, \boldsymbol{d_i'}, \boldsymbol{e_i''}),$$

where for c_i and $\boldsymbol{b_i}$ the notation $(\boldsymbol{g'}, \boldsymbol{d_i'})$ is used to abbreviate $[\boldsymbol{g} := \boldsymbol{g'}, \boldsymbol{d_i} := \boldsymbol{d_i'}]$, and for $\boldsymbol{n_i}$ and f_i we use $(\boldsymbol{g'}, \boldsymbol{d_i'}, \boldsymbol{e_i'})$ to abbreviate $[\boldsymbol{g} := \boldsymbol{g'}, \boldsymbol{d_i} := \boldsymbol{d_i'}, \boldsymbol{e_i} := \boldsymbol{e_i'}]$.

Example 6. Consider a system that continuously sends a random element of a finite type D. It is represented by

$$X = \tau \sum_{d:D} \tfrac{1}{|D|} : \mathrm{send}(d) \cdot X,$$

and is easily seen to be isomorphic to the following LPPE when initialised with $pc = 1$. The initial value of d can be chosen arbitrarily, as it will be overwritten before used.

$$X(pc : \{1, 2\}, d : D) =$$

$$pc = 1 \Rightarrow \tau \sum_{d:D} \tfrac{1}{|D|} : X(2, d)$$

$$+ \ pc = 2 \Rightarrow \mathrm{send}(d) \sum_{y:\{*\}} 1.0 : X(1, d^0)$$

Obviously, the earlier defined syntactic sugar could also be used on LPPEs, writing $\mathrm{send}(d) \cdot X(1, d^0)$ in the second summand. However, as linearisation will be defined only on the basic operators, we will often keep writing the full form.

V. LINEARISATION

Linearisation of a prCRL specification is performed in two steps: (1) Every right-hand side becomes a summation of process terms, each of which contains exactly one action; this is the *intermediate regular form* (IRF). This step is performed by Algorithm 1, which uses Algorithms 2 and 3. (2) An LPPE is created based on the IRF, using Algorithm 4. The algorithms are shown in detail on the following pages. We first illustrate both steps based on two examples.

Example 7. Consider the specification $X = a \cdot b \cdot c \cdot X$. We transform X into the strongly bisimilar (in this case even isomorphic) IRF $\{X_1 = a \cdot X_2, X_2 = b \cdot X_3, X_3 = c \cdot X_1\}$ (with initial process X_1). Now, an isomorphic LPPE is constructed by introducing a program counter pc that keeps track of the subprocess that is currently active, as below. It is easy to see that $Y(1)$ generates the same state space as X.

$$
\begin{aligned}
Y(pc : \{1, 2, 3\}) &= pc = 1 \Rightarrow a \cdot Y(2) \\
&+ pc = 2 \Rightarrow b \cdot Y(3) \\
&+ pc = 3 \Rightarrow c \cdot Y(1)
\end{aligned}
$$

Example 8. Now consider the following specification, consisting of two process equations with parameters. Let $B(d')$ be the initial process for some $d' \in D$.

$$B(d : D) =$$

$$\tau \sum_{e:E} \tfrac{1}{|E|} : \mathrm{send}(d + e) \sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.1 \text{ else } 0.9) :$$

$$((i = 1 \Rightarrow \mathrm{crash} \sum_{j:\{*\}} 1.0 : B(d)) + (i \neq 1 \Rightarrow C(d + 1)))$$

$$C(f : D) =$$

$$\mathrm{write}(f^2) \sum_{k:\{*\}} 1.0 : \sum_{g:D} \mathrm{write}(f + g) \sum_{l:\{*\}} 1.0 : B(f + g)$$

Again we introduce a new process for each subprocess. For brevity we use $(\boldsymbol{p})$ for $(d:D, f:D, e:E, i:\{1,2\}, g:D)$. The initial process is $X_1(d', f^0, e^0, i^0, g^0)$, where f^0, e^0, i^0, and g^0 can be chosen arbitrarily.

$$X_1(\boldsymbol{p}) = \tau \sum_{e:E} \frac{1}{|E|} : X_2(d, f^0, e, i^0, g^0)$$

$$X_2(\boldsymbol{p}) = \text{send}(d+e) \sum_{i:\{1,2\}} (\text{if } i=1 \text{ then } 0.1 \text{ else } 0.9) : X_3(d, f^0, e^0, i, g^0)$$

$$X_3(\boldsymbol{p}) = (i=1 \Rightarrow \text{crash} \sum_{j:\{*\}} 1.0 : X_1(d, f^0, e^0, i^0, g^0))$$
$$+ (i \neq 1 \Rightarrow \text{write}((d+1)^2) \sum_{k:\{*\}} 1.0 : X_4(d', d+1, e^0, i^0, g^0))$$

$$X_4(\boldsymbol{p}) = \sum_{g:D} \text{write}(f+g) \sum_{l:\{*\}} 1.0 : X_1(f+g, f^0, e^0, i^0, g^0)$$

Note that we added global variables to remember the values of variables that were bound by a nondeterministic or probabilistic summation. As the index variables j, k and l are never used, they are not remembered. We also reset variables that are not syntactically used in their scope to keep the state space minimal.

Again, the LPPE is obtained by introducing a program counter. The initial vector is $(1, d', f^0, e^0, i^0, g^0)$, where f^0, e^0, i^0, and g^0 can again be chosen arbitrarily.

$$X(pc:\{1,2,3,4\}, d:D, f:D, e:E, i:\{1,2\}, g:D) =$$
$$pc = 1 \Rightarrow \tau \sum_{e:E} \frac{1}{|E|} : X(2, d, f^0, e, i^0, g^0)$$
$$+ pc = 2 \Rightarrow \text{send}(d+e) \sum_{i:\{1,2\}} (\text{if } i=1 \text{ then } 0.1 \text{ else } 0.9) : X(3, d, f^0, e^0, i, g^0)$$
$$+ pc = 3 \wedge i = 1 \Rightarrow \text{crash} \sum_{j:\{*\}} 1.0 : X(1, d, f^0, e^0, i^0, g^0)$$
$$+ pc = 3 \wedge i \neq 1 \Rightarrow$$
$$\text{write}((d+1)^2) \sum_{k:\{*\}} 1.0 : X(4, d', d+1, e^0, i^0, g^0)$$
$$+ \sum_{g:D} pc = 4 \Rightarrow \text{write}(f+g) \sum_{l:\{*\}} 1.0 : X(1, f+g, f^0, e^0, i^0, g^0)$$

A. Transforming from prCRL to IRF

We now formally define the IRF, and then discuss the transformation from prCRL to IRF in more detail.

Definition 4. *A process term is in* IRF *if it adheres to the following grammar:*

$$p ::= c \Rightarrow p \mid p + p \mid \sum_{x:D} p \mid a(\boldsymbol{t}) \sum_{x:D} f : Y(\boldsymbol{t})$$

Note that in IRF every probabilistic sum goes to a process instantiation, and that process instantiations do not occur in any other way. A process equation is in IRF if its right-hand side is in IRF, and a specification is in IRF if all its

process equations are in IRF and all its processes have the same global variables. For every specification P with initial process $X(\boldsymbol{v})$ there exists a specification P' in IRF with initial process $X'(\boldsymbol{v}')$ such that $X(\boldsymbol{v}) \approx X'(\boldsymbol{v}')$ (as we provide an algorithm to find it). However, it is not hard to see that P' is not unique. Also, not necessarily $X(\boldsymbol{v}) \equiv X'(\boldsymbol{v}')$ (as we will show in Example 10).

Clearly, every specification P representing a finite PA can be transformed to an IRF describing an isomorphic PA: just define a data type S with an element s_i for every state of the PA underlying P, and create a process $X(s:S)$ consisting of a summation of terms of the form $s = s_i \Rightarrow a(\boldsymbol{t})(p_1 : s_1 \oplus p_2 : s_2 \ldots \oplus p_n : s_n)$ (one for each transition $s_i \xrightarrow{a(\boldsymbol{t})} \mu$, where $\mu(s_1) = p_1, \mu(s_2) = p_2, \ldots, \mu(s_n) = p_n$). However, this transformation completely defeats its purpose, as the whole idea behind the LPPE is to apply reductions *before* having to compute all states of the original specification.

Overview of the transformation to IRF.

Algorithm 1 transforms a specification P with initial process $X_1(\boldsymbol{v})$ to a specification P' with initial process $X_1'(\boldsymbol{v}')$ such that $X_1(\boldsymbol{v}) \approx X_1'(\boldsymbol{v}')$ and P' is in IRF. It requires that all global and local variables of P have unique names (which is easily achieved by α-conversion). Three important variables are used: (1) *done* is a set of process equations that are already in IRF; (2) *toTransform* is a set of process equations that still have to be transformed to IRF;

Algorithm 1: Transforming a specification to IRF

Input:
- A prCRL specification $P = \{X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1, \ldots, X_n(\boldsymbol{g_n} : \boldsymbol{G_n}) = p_n\}$ with unique variable names, and an initial vector $\boldsymbol{v}$ for X_1. (We use g_i^j to denote the jth element of $\boldsymbol{g_i}$.)

Output:
- A prCRL specification $\{X_1'(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p_1', \ldots, X_k'(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p_k'\}$ in IRF, and an initial vector $\boldsymbol{v}'$ such that $X_1'(\boldsymbol{v}') \approx X_1(\boldsymbol{v})$.

Initialisation

1   newPars $:= [(g_2^1 : G_2^1), (g_2^2 : G_2^2), \ldots, (g_3^1 : G_3^1), (g_3^2 : G_3^2), \ldots,$
$(g_n^1 : G_n^1), (g_n^2 : G_n^2), \ldots] + \boldsymbol{n}$
where $\boldsymbol{n} = [(v, D) \mid \exists i . p_i$ binds a variable v of type D via a nondeterministic or probabilistic sum and syntactically uses v within its scope]

2   pars $:= [(g_1^1 : G_1^1), (g_1^2 : G_1^2), \ldots] + $ newPars

3   $\boldsymbol{v}' := \boldsymbol{v} + [D^0 \mid (v, D) \leftarrow \text{newPars}, D^0 \text{ is any constant of type } D]$

4   done $:= \emptyset$

5   toTransform $:= \{X_1'(\text{pars}) = p_1\}$

6   bindings $:= \{X_1'(\text{pars}) = p_1\}$

Construction

7   **while** toTransform $\neq \emptyset$ **do**

8   Choose an arbitrary equation $(X_i'(\text{pars}) = p_i) \in$ toTransform

9     $(p_i', \text{newProcs}) := \text{transform}(p_i, \text{pars}, \text{bindings}, P, \boldsymbol{v}')$

10   done $:=$ done $\cup \{X_i'(\text{pars}) = p_i'\}$

11   bindings $:=$ bindings $\cup$ newProcs

12   toTransform $:= (\text{toTransform} \cup \text{newProcs}) \setminus \{X_i'(\text{pars}) = p_i\}$

13   **return** (done, $\boldsymbol{v}'$)

Algorithm 2: Transforming process terms to IRF

Input:
- A process term p, a list *pars* of typed global variables, a set *bindings* of process terms in P that have already been mapped to a new process, a specification P, and a new initial vector $\boldsymbol{v}'$.

Output:
- The IRF for p and the process equations to add to *toTransform*.

$\text{transform}(p, \text{pars}, \text{bindings}, P, \boldsymbol{v}') =$

1   **case** $p = a(\boldsymbol{t})\sum_{x:D} f : q$
2      $(q', \text{actualPars}) := \text{normalForm}(q, \text{pars}, P, \boldsymbol{v}')$
3   **if** $\exists j \,.\, (X'_j(\text{pars}) = q') \in \text{bindings}$ **then**
4   **return** $\left(a(\boldsymbol{t})\sum_{x:D} f : X'_j(\text{actualPars}), \emptyset\right)$
5   **else**
6        $k := |\text{bindings}| + 1$
7   **return** $\left(a(\boldsymbol{t})\sum_{x:D} f : X'_k(\text{actualPars}), \{(X'_k(\text{pars}) = q')\}\right)$
8   **case** $p = c \Rightarrow q$
9      $(\text{newRHS}, \text{newProcs}) := \text{transform}(q, \text{pars}, \text{bindings}, P, \boldsymbol{v}')$
10   **return** $(c \Rightarrow \text{newRHS}, \text{newProcs})$
11   **case** $p = q_1 + q_2$
12     $(\text{newRHS}_1, \text{newProcs}_1) := \text{transform}(q_1, \text{pars}, \text{bindings}, P, \boldsymbol{v}')$
13     $(\text{newRHS}_2, \text{newProcs}_2) := \text{transform}(q_2, \text{pars}, \text{bindings} \cup$
                            $\text{newProcs}_1, P, \boldsymbol{v}')$
14   **return** $(\text{newRHS}_1 + \text{newRHS}_2, \text{newProcs}_1 \cup \text{newProcs}_2)$
15   **case** $p = Y(\boldsymbol{t})$
16     $(\text{newRHS}, \text{newProcs}) := \text{transform}(\text{RHS}(Y), \text{pars}, \text{bindings}, P, \boldsymbol{v}')$
17   newRHS' = newRHS, with all free variables substituted by the
                    value provided for them by $\boldsymbol{t}$
18   **return** $(\text{newRHS'}, \text{newProcs})$
19   **case** $p = \sum_{x:D} q$
20     $(\text{newRHS}, \text{newProcs}) := \text{transform}(q, \text{pars}, \text{bindings}, P, \boldsymbol{v}')$
21   **return** $(\sum_{x:D} \text{newRHS}, \text{newProcs})$

Algorithm 3: Normalising process terms

Input:
- A process term p, a list *pars* of typed global variables, a prCRL specification P, and a new initial vector $\boldsymbol{v}'$.

Output:
- The normal form p' of p, and the actual parameters needed to supply to a process which has right-hand side p' to make its behaviour strongly probabilistic bisimilar to p.

$\text{normalForm}(p, \text{pars}, P, \boldsymbol{v}') =$

1   **case** $p = Y(\boldsymbol{t})$
2      $p' := \text{RHS}(Y)$
3      $\text{actualPars} := [n(v) \mid (v, D) \leftarrow \text{pars}]$
       where

$$n(v) = \begin{cases} v^0 & \text{if } v \text{ is no global variable of } Y \text{ in } P, \\ & (v^0 \text{ can be found by inspecting pars and } \boldsymbol{v}') \\ \boldsymbol{t}(i) & \text{if } v \text{ is the } i^{\text{th}} \text{ global variable of } Y \text{ in } P \end{cases}$$

4   **return** $(p', \text{actualPars})$
5   **case** otherwise
6   **return** $(p, [n'(v) \mid (v, D) \leftarrow \text{pars}])$
       where $n'(v) = v$ if v occurs syntactically in p,
             otherwise it is v's initial value v^0

---

(3) *bindings* is a set of process equations $X'_i(\text{pars}) = p_i$ such that $X'_i(\text{pars})$ is the process in *done* $\cup$ *toTransform* representing the process term p_i of the original specification.

Initially, *done* is empty and we bind the right-hand side of the initial process to X'_1 (and add this equation to *toTransform*). Also, *pars* becomes the list of all variables occurring in P as global variables or in a summation (and syntactically used after being bound). The new initial vector is constructed by appending dummy values to the original initial vector for all newly added parameters. (We use Haskell-like list comprehension to denote this.) Then, basically we repeatedly take a process equation $X'_i(\text{pars}) = p_i$ from *toTransform*, transform p_i to a strongly probabilistic bisimilar IRF p'_i using Algorithm 2, add the process $X'_i(\text{pars}) = p'_i$ to *done*, and remove $X'_i(\text{pars}) = p_i$ from *toTransform*. The transformation may have introduced new processes, which are added to *toTransform*, and *bindings* is updated accordingly.

Transforming single process terms to IRF.

Algorithm 2 transforms process terms to IRF recursively by means of a case distinction over the structure of the terms.

The base case is a probabilistic choice $a(\boldsymbol{t})\sum_{x:D} f : q$. The corresponding IRF is $a(\boldsymbol{t})\sum_{x:D} f : X'_i(\text{actualPars})$, where X'_i is either the process name already mapped to q (as stored in *bindings*), or a new process name when there did not yet exist such a process. More precisely, instead of q we use its *normal form* (computed by Algorithm 3); when q is a process instantiation $Y(\boldsymbol{t})$, its normal form is the right-hand side of Y, otherwise it is just q. When q is *not* a process instantiation, the actual parameters for X'_i are just the global variables (possibly resetting variables that are not used in q). When $q = Y(v_1, v_2, \ldots, v_n)$, all global variables are reset, except the ones corresponding to the original global variables of Y; for them $v_1, v_2, \ldots, v_n$ are used. Newly created processes are added to *toTransform*.

For a summation $q_1 + q_2$, the IRF is $q'_1 + q'_2$ (with q'_i an IRF of q_i). For the condition $c \Rightarrow q_1$ it is $c \Rightarrow q'_1$, and for $\sum_{x:D} q_1$ it is $\sum_{x:D} q'_1$. Finally, the IRF for $Y(\boldsymbol{t})$ is the IRF for the right-hand side of Y, where the global variables of Y occurring in this term have been substituted by the expressions given by $\boldsymbol{t}$.

Example 9. We linearise two example specifications: $P_1 = \{X_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2, X_2 = a \cdot b \cdot c \cdot X_1\}$, and $P_2 = \{X_3(d : D) = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)\}$ (with initial processes X_1 and $X_3(d')$). Tables II and III show *done*, *toTransform* and *bindings* at line 7 of Algorithm 1 for every iteration. As both *done* and *bindings* only grow, we just list their additions. For layout purposes, we omit the parameters $(d : D, e : D)$ of every X''_i in Table III. The initial processes are X'_1 and $X''_1(d', e^0)$ for some $e^0 \in D$.

Theorem 1. *Let* $P = \{X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1, \ldots, X_n(\boldsymbol{g_n} : \boldsymbol{G_n}) = p_n\}$ *be a prCRL specification with initial vector* $\boldsymbol{v}$ *for* X_1. *Given these inputs Algorithm 1 terminates, and the specification* $P' = \{X'_1(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p'_1, \ldots, X'_k(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p'_k\}$ *and initial vector* $\boldsymbol{v}'$ *it returns are such that* $X'_1(\boldsymbol{v}')$ *in* P' *is strongly probabilistic bisimilar to* $X_1(\boldsymbol{v})$ *in* P. *Also,* P' *is in IRF.*

The following example shows that Algorithm 1 does not always compute an isomorphic specification.

Table II
TRANSFORMING $\{X_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2, X_2 = a \cdot b \cdot c \cdot X_1\}$ WITH INITIAL PROCESS X_1 TO IRF.

	done	toTransform	bindings
0	$\emptyset$	$X'_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2$	$X'_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2$
1	$X'_1 = a \cdot X'_2 + c \cdot X'_3$	$X'_2 = b \cdot c \cdot X_1, X'_3 = a \cdot b \cdot c \cdot X_1$	$X'_2 = b \cdot c \cdot X_1, X'_3 = a \cdot b \cdot c \cdot X_1$
2	$X'_2 = b \cdot X'_4$	$X'_3 = a \cdot b \cdot c \cdot X_1, X'_4 = c \cdot X_1$	$X'_4 = c \cdot X_1$
3	$X'_3 = a \cdot X'_2$	$X'_4 = c \cdot X_1$	
4	$X'_4 = c \cdot X'_1$	$\emptyset$	

Table III
TRANSFORMING $\{X_3(d : D) = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)\}$ WITH INITIAL PROCESS $X_3(d')$ TO IRF.

	done	toTransform	bindings
0	$\emptyset$	$X''_1 = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)$	$X''_1 = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)$
1	$X''_1 = \sum_{e:D} a(d + e) \cdot X''_2(d', e)$	$X''_2 = c(e) \cdot X_3(5)$	$X''_2 = c(e) \cdot X_3(5)$
2	$X''_2 = c(e) \cdot X''_1(5, e^0)$	$\emptyset$	

Example 10. Let $X = \sum_{d:D} a(d) \cdot b(f(d)) \cdot X$, with $f(d) = 0$ for all $d \in D$. Then, our procedure will yield the specification $\{X'_1(d : D) = \sum_{d:D} a(d) \cdot X'_2(d),$ $X'_2(d : D) = b(f(d)) \cdot X'_1(d^0)\}$ with initial process $X'_1(d^0)$ for an arbitrary $d^0 \in D$. Note that the number of states of $X'_1(d^0)$ is $|D| + 1$ for any $d^0 \in D$. However, the state space of X only consists of the two states X and $b(0) \cdot X$.

B. Transforming from IRF to LPPE

Based on a specification P' in IRF, Algorithm 4 constructs an LPPE X. The global variables of X are a newly introduced program counter pc and all global variables of P'. To construct the summands for X, the algorithm ranges over the process equations in P'. For each equa-

---

Algorithm 4: Constructing an LPPE from an IRF

Input:
- A prCRL specification $P' = \{X'_1(\boldsymbol{g} : \boldsymbol{G}) = p'_1, \ldots,$ $X'_k(\boldsymbol{g} : \boldsymbol{G}) = p'_k\}$ in IRF (without variable pc).

Output:
- An LPPE $X(pc : \{1, \ldots, k\}, \boldsymbol{g} : \boldsymbol{G})$ such that $X'_1(\boldsymbol{v}) \equiv X(1, \boldsymbol{v})$ for all $\boldsymbol{v} \in \boldsymbol{G}$.

Construction

1. $S = \emptyset$
2. **forall** $(X'_i(\boldsymbol{g} : \boldsymbol{G}) = p'_i) \in P'$ **do**
3. $\quad S := S \cup \text{makeSummands}(p'_i, i)$
4. **return** $X(pc : \{1, \ldots, k\}, \boldsymbol{g} : \boldsymbol{G}) = \sum_{s \in S} s$

where
$\quad$ makeSummands$(p, i) =$
5. $\quad$ **case** $p = a(\boldsymbol{t})\sum_{x:D} f : X'_j(e_1, \ldots, e_k)$
6. $\quad\quad$ **return** $\{pc = i \Rightarrow a(\boldsymbol{t})\sum_{x:D} f : X(j, e_1, \ldots, e_k)\}$
7. $\quad$ **case** $p = c \Rightarrow q$
8. $\quad\quad$ **return** $\{c \Rightarrow q' \mid q' \in \text{makeSummands}(q, i)\}$
9. $\quad$ **case** $p = q_1 + q_2$
10. $\quad\quad$ **return** makeSummands$(q_1, i) \cup$ makeSummands(q_2, i)
11. $\quad$ **case** $p = \sum_{x:D} q$
12. $\quad\quad$ **return** $\{\sum_{x:D} q' \mid q' \in \text{makeSummands}(q, i)\}$

---

tion $X'_i(\boldsymbol{g} : \boldsymbol{G}) = a(\boldsymbol{t})\sum_{x:D} f : X'_j(e_1, \ldots, e_k)$, a summand $pc = i \Rightarrow a(\boldsymbol{t})\sum_{x:D} f : X(j, e_1, \ldots, e_k)$ is constructed. For an equation $X'_i(\boldsymbol{g} : \boldsymbol{G}) = q_1 + q_2$ the union of the summands produced by $X'_i(\boldsymbol{g} : \boldsymbol{G}) = q_1$ and $X'_i(\boldsymbol{g} : \boldsymbol{G}) = q_2$ is taken. For $X'_i(\boldsymbol{g} : \boldsymbol{G}) = c \Rightarrow q$ the condition c is prefixed before the summands produced by $X'_i(\boldsymbol{g} : \boldsymbol{G}) = q$; the nondeterministic sum is handled similarly.

To be precise, an actual LPPE is only obtained after a few manipulation of the summands obtained this way. The nondeterministic sums should still be moved to the front, and separate nondeterministic sums and separate conditions should be merged (using vectors and conjunctions, respectively). This does not change behaviour because of the assumed uniqueness of variable names.

Theorem 2. *Let* $P' = \{X'_1(\boldsymbol{g} : \boldsymbol{G}) = p'_1, \ldots, X'_k(\boldsymbol{g} : \boldsymbol{G}) = p'_k\}$ *be a specification in IRF, and* $X(pc : \{1, \ldots, k\}, \boldsymbol{g} : \boldsymbol{G})$ *the LPPE obtained by applying Algorithm 4 to* P'. *Then,* $X'_1(\boldsymbol{v}) \equiv X(1, \boldsymbol{v})$ *for every* $\boldsymbol{v} \in \boldsymbol{G}$. *Also,* X *is an LPPE (after, within each summand, moving the nondeterministic sums to the beginning and merging separate nondeterministic sums and separate conditions).*

Proposition 2. *The time complexity of linearising a specification* P *is in* $O(n^3)$, *where* $n = \sum_{(X_i(\boldsymbol{g_i}:\boldsymbol{G_i})=p_i) \in P} size(\boldsymbol{g_i}) + size(p_i)$. *The LPPE size is in* $O(n^2)$.

Although the transformation to LPPE increases the size of the specification, it facilitates optimisations to reduce the state space (which is worst-case in $O(2^n)$).

Example 11. Looking at the IRFs obtained in Example 9, it follows that $X'_1 \equiv X(1)$ where $X(pc : \{1, 2, 3, 4\}) = (pc = 1 \Rightarrow a \cdot X(2)) + (pc = 1 \Rightarrow c \cdot X(3)) + (pc = 2 \Rightarrow b \cdot X(4)) + (pc = 3 \Rightarrow a \cdot X(2)) + (pc = 4 \Rightarrow c \cdot X(1))$.

Also, $X''_1(d', e^0) \equiv X(1, d', e^0)$ where $X(pc : \{1, 2\}, d : D, e : D) = (\sum_{e:D} pc = 1 \Rightarrow a(d + e) \cdot X(2, d', e)) + (pc = 2 \Rightarrow c(e) \cdot X(1, 5, e^0))$.

Table IV
SOS RULES FOR PARALLEL PRCRL.

$$\text{PAR-L } \frac{p \xrightarrow{\alpha} \mu}{p \,\|\, q \xrightarrow{\alpha} \mu'} \text{ where } \forall p' \,.\, \mu'(p' \,\|\, q) = \mu(p') \qquad\qquad \text{PAR-R } \frac{q \xrightarrow{\alpha} \mu}{p \,\|\, q \xrightarrow{\alpha} \mu'} \text{ where } \forall q' \,.\, \mu'(p \,\|\, q') = \mu(q')$$

$$\text{PAR-COM } \frac{p \xrightarrow{a(\boldsymbol{t})} \mu \qquad q \xrightarrow{b(\boldsymbol{t})} \mu'}{p \,\|\, q \xrightarrow{c(\boldsymbol{t})} \mu''} \text{ if } \gamma(a,b) = c, \text{ where } \forall p', q' \,.\, \mu''(p' \,\|\, q') = \mu(p') \cdot \mu'(q')$$

$$\text{HIDE-T } \frac{p \xrightarrow{a(\boldsymbol{t})} \mu}{\tau_H(p) \xrightarrow{\tau} \tau_H(\mu)} \text{ if } a \in H \qquad\qquad \text{HIDE-F } \frac{p \xrightarrow{a(\boldsymbol{t})} \mu}{\tau_H(p) \xrightarrow{a(\boldsymbol{t})} \tau_H(\mu)} \text{ if } a \notin H$$

$$\text{RENAME } \frac{p \xrightarrow{a(\boldsymbol{t})} \mu}{\rho_R(p) \xrightarrow{R(a)(\boldsymbol{t})} \rho_R(\mu)} \qquad\qquad \text{ENCAP-F } \frac{p \xrightarrow{a(\boldsymbol{t})} \mu}{\partial_E(p) \xrightarrow{a(\boldsymbol{t})} \partial_E(\mu)} \text{ if } a \notin E$$

VI. PARALLEL COMPOSITION

Using prCRL processes as basic building blocks, we support the modular construction of large systems by introducing top-level parallelism, encapsulation, hiding, and renaming. The resulting language is called *parallel prCRL*.

Definition 5. *A* process term *in parallel prCRL is any term that can be generated according to the following grammar.*

$$q ::= p \mid q \,\|\, q \mid \partial_E(q) \mid \tau_H(q) \mid \rho_R(q)$$

Here, p is a prCRL process, $E, H \subseteq \text{Act}$, and $R\colon \text{Act} \to \text{Act}$. A parallel prCRL process equation *is of the form $X(\boldsymbol{g} : \boldsymbol{G}) = q$, and a* parallel prCRL specification *is a set of such equations. These equations and specifications are under the same restrictions as their prCRL counterparts.*

In a parallel prCRL process term, $q_1 \,\|\, q_2$ is parallel composition. Furthermore, $\partial_E(q)$ encapsulates the actions in E, $\tau_H(q)$ hides the actions in H (renaming them to τ and removing their parameters), and $\rho_R(q)$ renames actions using R. Parallel processes by default interleave all their actions. However, we assume a partial function $\gamma\colon \text{Act} \times \text{Act} \to \text{Act}$ that specifies which actions may communicate; more precisely, $\gamma(a,b) = c$ denotes that a and b may communicate, resulting in the action c. The SOS rules for parallel prCRL are shown in Table IV. For any probability distribution μ, we denote by $\tau_H(\mu)$ the probability distribution μ' such that $\forall p \,.\, \mu'(\tau_H(p)) = \mu(p)$. Similarly, we use $\rho_R(\mu)$ and $\partial_E(\mu)$.

A. Linearisation of parallel processes

The LPPE format allows processes to be put in parallel very easily. Although the LPPE size is worst-case exponential in the number of parallel processes (when all summands have different actions and all these actions can communicate), in practice we see only linear growth (since often only a few actions communicate). Given the LPPEs

$$X(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i \colon X(\boldsymbol{n_i}),$$

$$Y(\boldsymbol{g'} : \boldsymbol{G'}) = \sum_{i \in I'} \sum_{\boldsymbol{d'_i} : \boldsymbol{D'_i}} c'_i \Rightarrow a'_i(b'_i) \sum_{\boldsymbol{e'_i} : \boldsymbol{E'_i}} f'_i \colon Y(\boldsymbol{n'_i}),$$

where all global and local variables are assumed to be unique, the product $Z(\boldsymbol{g} : \boldsymbol{G}, \boldsymbol{g'} : \boldsymbol{G'}) = X(\boldsymbol{g}) \,\|\, Y(\boldsymbol{g'})$ is constructed as follows, based on the construction presented by Usenko for traditional LPEs [16]. Note that the first set of summands represents X doing a transition independent from Y, and that the second set of summands represents Y doing a transition independent from X. The third set corresponds to their communications.

$$\begin{aligned}
Z(\boldsymbol{g} : \boldsymbol{G}, \boldsymbol{g'} : \boldsymbol{G'}) = &\sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i \colon Z(\boldsymbol{n_i}, \boldsymbol{g'}) \\
+ &\sum_{i \in I'} \sum_{\boldsymbol{d'_i} : \boldsymbol{D'_i}} c'_i \Rightarrow a'_i(b'_i) \sum_{\boldsymbol{e'_i} : \boldsymbol{E'_i}} f'_i \colon Z(\boldsymbol{g}, \boldsymbol{n'_i}) \\
+ &\sum_{(k,l) \in I\gamma I'} \sum_{(\boldsymbol{d_k}, \boldsymbol{d'_l}) : \boldsymbol{D_k} \times \boldsymbol{D'_l}} c_k \wedge c'_l \wedge \boldsymbol{b_k} = \boldsymbol{b'_l} \Rightarrow \\
&\gamma(a_k, a'_l)(\boldsymbol{b_k}) \sum_{(e_k, e'_l) : \boldsymbol{E_k} \times \boldsymbol{E'_l}} f_k \cdot f'_l \colon Z(\boldsymbol{n_k}, \boldsymbol{n'_l})
\end{aligned}$$

In this definition, $I\gamma I'$ is the set of all combinations of summands $(k,l) \in I \times I'$ such that the action a_k of summand k and the action a'_l of summand l can communicate. Formally, $I\gamma I' = \{(k,l) \in I \times I' \mid (a_k, a'_l) \in \text{domain}(\gamma)\}$.

Proposition 3. *For all $\boldsymbol{v} \in \boldsymbol{G}, \boldsymbol{v'} \in \boldsymbol{G'}$, it holds that $Z(\boldsymbol{v}, \boldsymbol{v'}) \equiv X(\boldsymbol{v}) \,\|\, Y(\boldsymbol{v'})$.*

B. Linearisation of hiding, encapsulation and renaming

For hiding, renaming, and encapsulation, linearisation is quite straightforward. For the LPPE

$$X(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i \colon X(\boldsymbol{n_i}),$$

let the LPPEs $U(\boldsymbol{g})$, $V(\boldsymbol{g})$, and $W(\boldsymbol{g})$, for $\tau_H(X(\boldsymbol{g}))$, $\rho_R(X(\boldsymbol{g}))$, and $\partial_E(X(\boldsymbol{g}))$, respectively, be given by

$$U(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a'_i(b'_i) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i \colon U(\boldsymbol{n_i}),$$

$$V(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i''(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i : V(\boldsymbol{n_i}),$$

$$W(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I'} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i : W(\boldsymbol{n_i}),$$

where

$$a_i' = \begin{cases} \tau & \text{if } a_i \in H \\ a_i & \text{otherwise} \end{cases}, \quad \boldsymbol{b_i}' = \begin{cases} [] & \text{if } a_i \in H \\ \boldsymbol{b_i} & \text{otherwise} \end{cases}$$

$$a_i'' = R(a_i) \qquad , \quad I' = \{i \in I \mid a_i \notin E\}.$$

Proposition 4. *For all* $\boldsymbol{v} \in \boldsymbol{G}$, $U(\boldsymbol{v}) \equiv \tau_H(X(\boldsymbol{v}))$, $V(\boldsymbol{v}) \equiv \rho_R(X(\boldsymbol{v}))$, *and* $W(\boldsymbol{v}) \equiv \partial_E(X(\boldsymbol{v}))$.

VII. IMPLEMENTATION AND CASE STUDY

We developed a Haskell implementation of all procedures for linearisation of prCRL specifications, parallel composition, hiding, encapsulation and renaming[1]. As Haskell is a functional language, the implementations are almost identical to their mathematical representations in this paper. To test the correctness of the procedures, we used the implementation to linearise all examples in this paper, and indeed found exactly the LPPEs we expected.

To illustrate the possible reductions for LPPEs, we model a protocol, inspired by the various leader election protocols that can be found in literature (e.g., Itai-Rodeh [30]), in prCRL. On this model we apply one reduction manually, and several more automatically. Future work will focus on defining and studying more reductions in detail.

We consider a system consisting of two nodes, deciding on a leader by rolling two dice and comparing the results. When both roll the same number, the experiment is repeated. Otherwise, the node that rolled highest wins. The system can be modelled by the prCRL specification shown in Figure 2. We assume that *Die* is a data type consisting of the numbers from 1 to 6, and that *Id* is a data type consisting of the identifiers *one* and *two*. The function *other* is assumed to provide the identifier different from its argument.

Each component has been given an identifier for reference during communication, and consists of a passive thread P and an active thread A. The passive thread waits to receive what the other component has rolled, and then provides the active thread an opportunity to obtain this result. The active thread first rolls a die, and sends the result to the other component (communicating via the *comm* action). Then it tries to read the result of the other component through the passive process (or blocks until this result has been received). Based on the results, either the processes start over, or they declare their victory or loss.

Linearising this specification we obtain a process with 18 parameters and 14 summands, shown in [27]. Computing the state space we obtain 3763 states and 6158 transitions. Due to the uniform linear format, we can now apply several

$$\begin{aligned}
&P(id : Id, val : Die, set : Bool) = \\
&\quad set = false \Rightarrow \sum_{d:Die} receive(id, other(id), d) \cdot P(id, d, true) \\
&\quad + set = true \Rightarrow getVal(val).P(id, val, false) \\
&A(id : Id) = \\
&\quad roll(id) \sum_{d:Die} \tfrac{1}{6} : send(other(id), id, d) \cdot \sum_{e:Die} readVal(e). \\
&\quad\quad (\quad (d = e \Rightarrow A(id)) \\
&\quad\quad + (d > e \Rightarrow leader(id) \cdot A(id)) \\
&\quad\quad + (e > d \Rightarrow follower(id) \cdot A(id))) \\
&C(id : Id) = \partial_{getVal, readVal}(P(id, 1, false) \| A(id)) \\
&S = \partial_{send, receive}(C(one) \| C(two)) \\[4pt]
&\gamma(receive, send) = comm \qquad \gamma(getVal, readVal) = checkVal
\end{aligned}$$

Figure 2. A prCRL model of a leader election protocol.

classical reduction techniques to the result. Here we will demonstrate the applicability of four such techniques using one of the summands as an example:

$$\sum_{e21:Die} pc21 = 3 \wedge pc11 = 1 \wedge set11 \wedge val11 = e21 \Rightarrow$$
$$checkVal(val11) \sum_{(k1,k2):\{*\}\times\{*\}} multiply(1.0, 1.0):$$
$$Z(1, id11, val11, false, 1, 4, id21, d21, e21,$$
$$pc12, id12, val12, set12, d12, pc22, id22, d22, e22)$$

Constant elimination [19]. Syntactic analysis of the LPPE revealed that $pc11$, $pc12$, $id11$, $id12$, $id21$, $id22$, $d11$ and $d12$ never get any value other than their initial value. Therefore, these parameters can be removed and everywhere they occur their initial value is substituted for them.

Summation elimination [16]. The summand at hand ranges $e21$ over *Die*, but the condition requires it to be equal to $val11$. Therefore, the summation can be removed and occurrences of $e21$ substituted by $val11$. This way, all summations of the LPPE can be removed.

Data evaluation / syntactic clean-up. After constant elimination, the condition $pc11 = 1$ has become $1 = 1$ and can therefore be eliminated. Also, the multiplication can be evaluated to 1.0, and the Cartesian product can be simplified.

Liveness analysis [20]. Using the methods of [20] we found that after executing the summand at hand $val11$ is always first reset before used again. Therefore, we can also immediately reset it after this summand, thereby reducing the state space. This way, two resets have been added.

Combining all these methods to the complete LPPE (the first three automatically, the last one manually), a strongly probabilistic bisimilar LPPE was obtained (see [27] for the details). The summand discussed above became:

$$pc21 = 3 \wedge set11 \Rightarrow checkVal(val11) \sum_{k:\{*\}} 1.0:$$
$$Z(1, false, 4, d21, val11, val12, set12, pc22, d22, e22)$$

Computing the state space of the reduced LPPE we obtained 1693 states (-55%) and 2438 transitions (-60%).

VIII. Conclusions and Future Work

This paper introduced a linear process algebraic format for systems incorporating both nondeterministic and probabilistic choice. The key ingredients are: (1) the combined treatment of data and data-dependent probabilistic choice in a fully symbolic manner; (2) a symbolic transformation of probabilistic process algebra terms with data into this linear format, while preserving strong probabilistic bisimulation. The linearisation is the first essential step towards the symbolic minimisation of probabilistic state spaces, as well as the analysis of parameterised probabilistic protocols. The results show that the treatment of probabilities is simple and elegant, and rather orthogonal to the traditional setting [16]. Future work will concentrate on branching bisimulation preserving symbolic minimisation techniques such as confluence reduction [26], and on applying proof techniques such as the cones and foci method to LPPEs.

References

[1] J. Groote and A. Ponse, "The syntax and semantics of μCRL," in *Proc. of Algebra of Communicating Processes*, ser. Workshops in Computing, 1995, pp. 26–62.

[2] http://www.prismmodelchecker.org/.

[3] C. Baier, F. Ciesinski, and M. Größer, "PROBMELA: a modeling language for communicating probabilistic processes," in *Proc. of the 2nd ACM/IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004, pp. 57–66.

[4] H. Bohnenkamp, P. D'Argenio, H. Hermanns, and J.-P. Katoen, "MODEST: A compositional modeling formalism for hard and softly timed systems," *IEEE Transactions of Software Engineering*, vol. 32, no. 10, pp. 812–830, 2006.

[5] P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen, "Reachability analysis of probabilistic systems by successive refinements," in *Proc. of the Joint Int. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV)*, ser. LNCS, vol. 2165, 2001, pp. 39–56.

[6] L. de Alfaro and P. Roy, "Magnifying-lens abstraction for Markov decision processes," in *Proc. of the 19th Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 4590, 2007, pp. 325–338.

[7] T. Henzinger, M. Mateescu, and V. Wolf, "Sliding window abstraction for infinite Markov chains," in *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 5643, 2009, pp. 337–352.

[8] J.-P. Katoen, D. Klink, M. Leucker, and V. Wolf, "Three-valued abstraction for continuous-time Markov chains," in *Proc. of the 19th Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 4590, 2007, pp. 311–324.

[9] M. Kwiatkowska, G. Norman, and D. Parker, "Game-based abstraction for Markov decision processes," in *Proc. of the 3rd Int. Conf. on Quantitative Evaluation of Systems (QEST)*, 2006, pp. 157–166.

[10] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic CEGAR," in *Proc. of the 20th Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 5123, 2008, pp. 162–175.

[11] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Abstraction refinement for probabilistic software," in *Proc. of the 19th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, vol. 5403, 2009, pp. 182–197.

[12] J. Hurd, A. McIver, and C. Morgan, "Probabilistic guarded commands mechanized in *HOL*," *Theoretical Computer Science*, vol. 346, no. 1, pp. 96–112, 2005.

[13] M. Bezem and J. Groote, "Invariants in process algebra with data," in *Proc. of the 5th Int. Conf. on Concurrency Theory (CONCUR)*, ser. LNCS, vol. 836, 1994, pp. 401–416.

[14] K. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Information and Computation*, vol. 94, no. 1, pp. 1–28, 1991.

[15] D. Bosscher and A. Ponse, "Translating a process algebra with symbolic data values to linear format," in *Proc. of the 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. BRICS Notes Series, vol. NS-95-2, 1995, pp. 119–130.

[16] Y. Usenko, "Linearization in μCRL," Ph.D. dissertation, Eindhoven University of Technology, 2002.

[17] J. Groote and J. Springintveld, "Focus points and convergent process operators: a proof strategy for protocol verification," *Journal of Logic and Algebraic Programming*, vol. 49, no. 1-2, pp. 31–60, 2001.

[18] W. Fokkink, J. Pang, and J. van de Pol, "Cones and foci: A mechanical framework for protocol verification," *Formal Methods in System Design*, vol. 29, no. 1, pp. 1–31, 2006.

[19] J. Groote and B. Lisser, "Computer assisted manipulation of algebraic process specifications," CWI, Tech. Rep. SEN-R0117, 2001.

[20] J. van de Pol and M. Timmer, "State space reduction of linear processes using control flow reconstruction," in *Proc. of the 7th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, ser. LNCS, vol. 5799, 2009, pp. 54–68.

[21] M. Espada and J. van de Pol, "An abstract interpretation toolkit for μCRL," *Formal Methods in System Design*, vol. 30, no. 3, pp. 249–273, 2007.

[22] S. Blom, B. Lisser, J. van de Pol, and M. Weber, "A database approach to distributed state-space generation," *Journal of Logic and Computation*, 2009, Advance Access, March 5.

[23] S. Blom and J. van de Pol, "Symbolic reachability for process algebras with recursive data types," in *Proc. of the 5th Int. Colloquium on Theoretical Aspects of Computing (ICTAC)*, ser. LNCS, vol. 5160, 2008, pp. 81–95.

[24] J. Groote and R. Mateescu, "Verification of temporal properties of processes in a setting with data," in *Proc. of the 7th Int. Conf. on Algebraic Methodology and Software Technology (AMAST)*, ser. LNCS, vol. 1548, 1998, pp. 74–90.

[25] J. Groote and T. Willemse, "Model-checking processes with data," *Science of Computer Programming*, vol. 56, no. 3, pp. 251–273, 2005.

[26] S. Blom and J. van de Pol, "State space reduction by proving confluence," in *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 2404, 2002, pp. 596–609.

[27] J.-P. Katoen, J. van de Pol, M. Stoelinga, and M. Timmer, "A linear process algebraic format for probabilistic systems with data (extended version)," TR-CTIT-10-11, CTIT, University of Twente, Tech. Rep., 2010.

[28] R. Segala, "Modeling and verification of randomized distributed real-time systems," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[29] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[30] W. Fokkink and J. Pang, "Variations on Itai-Rodeh leader election for anonymous rings and their analysis in PRISM," *Journal of Universal Computer Science*, vol. 12, no. 8, pp. 981–1006, 2006.