A linear process-algebraic format for probabilistic systems with data (extended version)

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,

This research has been partially funded by NWO under grant 612.063.817 (SYRUP) and grant Dn 63-257 (ROCKS), and by the European Union under FP7-ICT-2007-1 grant 214755 (QUASIMODO).

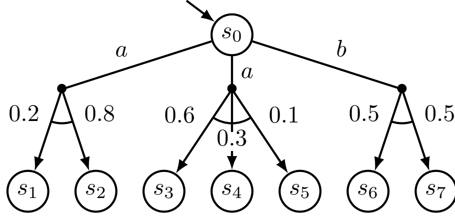

Figure 1. A probabilistic automaton.

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.

The paper is followed by an appendix, containing 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 [27].

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 [28], 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.*

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

Here, Y *is a process name,* c *a boolean expression,* $a \in \mathrm{Act}$ *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* $\sum\!\!\!\!\bullet_{x:D} f$ *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.

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

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

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

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.

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

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,

$$\begin{aligned} p \triangleleft c \triangleright q &\stackrel{\text{def}}{=} (c \Rightarrow p) + (\neg c \Rightarrow q) \\ a(\boldsymbol{t}) \cdot p &\stackrel{\text{def}}{=} a(\boldsymbol{t}) \textstyle\sum_{x:\{*\}} 1.0 : p \\ a(\boldsymbol{t}) \,\mathcal{U}_{d:D}\, c \Rightarrow p &\stackrel{\text{def}}{=} a(\boldsymbol{t}) \textstyle\sum_{d:D} f : p \end{aligned}$$

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 $\mathcal{U}_{d:D}\, c \Rightarrow p$ 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:

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

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

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

where 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 $b = 1$.

$$\begin{aligned} & X(a : \{1, 2\}, b : \{1, 2\}, x : D, y : D) = \\ & \quad \textstyle\sum_{d:D} \quad a = 1 \qquad\quad\;\; \Rightarrow \mathrm{read}(d) \cdot X(2, b, d, y) \qquad (1) \\ & \quad + \qquad\;\; a = 2 \land b = 1 \Rightarrow \mathrm{comm}(x) \cdot X(1, 2, x, x) \quad (2) \\ & \quad + \qquad\;\; b = 2 \qquad\quad\;\; \Rightarrow \mathrm{write}(y) \cdot X(a, 1, x, y) \qquad (3) \end{aligned}$$

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:*

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

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

$$c_i(\boldsymbol{g'}, \boldsymbol{d'_i}) \land a_i(\boldsymbol{b_i}(\boldsymbol{g'}, \boldsymbol{d'_i})) = a(\boldsymbol{q}) \land \forall \boldsymbol{e'_i} \in \boldsymbol{E_i} \, .$$
$$\mu(\boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d'_i}, \boldsymbol{e'_i})) = \sum_{\substack{\boldsymbol{e''_i} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d'_i}, \boldsymbol{e'_i}) = \boldsymbol{n_i}(\boldsymbol{g'}, \boldsymbol{d'_i}, \boldsymbol{e''_i})}} f_i(\boldsymbol{g'}, \boldsymbol{d'_i}, \boldsymbol{e''_i}),$$

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

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

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.

$$\begin{aligned} X(pc : \{1, 2\}, d : D) = & \\ & pc = 1 \Rightarrow \tau \sum_{d:D}^{\bullet} \tfrac{1}{|D|} \colon X(2, d) \\ & + pc = 2 \Rightarrow \mathrm{send}(d) \sum_{y:\{*\}}^{\bullet} 1.0 \colon X(1, d^0) \end{aligned}$$

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 (page 5), which uses Algorithms 2 and 3 (page 6). (2) An LPPE is created based on the IRF, using Algorithm 4 (page 7). 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.

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

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

$$\begin{aligned} & B(d : D) = \\ & \quad \tau \sum_{e:E}^{\bullet} \tfrac{1}{|E|} \colon \mathrm{send}(d + e) \sum_{i:\{1,2\}}^{\bullet} (\text{if } i = 1 \text{ then } 0.1 \text{ else } 0.9) \colon \\ & \qquad ((i = 1 \Rightarrow \mathrm{crash} \sum_{j:\{*\}}^{\bullet} 1.0 \colon B(d)) + (i \neq 1 \Rightarrow C(d + 1))) \\ & C(f : D) = \\ & \quad \mathrm{write}(f^2) \sum_{k:\{*\}}^{\bullet} 1.0 \colon \sum_{g:D} \mathrm{write}(f + g) \sum_{l:\{*\}}^{\bullet} 1.0 \colon B(f + g) \end{aligned}$$

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.

$$X_1(\boldsymbol{p}) = \tau \sum_{e:E} \tfrac{1}{|E|} \colon X_2(d, f^0, e, i^0, g^0)$$
$$X_2(\boldsymbol{p}) = \text{send}(d + e) \sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.1 \text{ else } 0.9) \colon X_3(d, f^0, e^0, i, g^0)$$
$$X_3(\boldsymbol{p}) = (i = 1 \Rightarrow \text{crash} \sum_{j:\{*\}} 1.0 \colon X_1(d, f^0, e^0, i^0, g^0)) + (i \neq 1 \Rightarrow \text{write}((d + 1)^2) \sum_{k:\{*\}} 1.0 \colon X_4(d', d + 1, e^0, i^0, g^0))$$
$$X_4(\boldsymbol{p}) = \sum_{g:D} \text{write}(f + g) \sum_{l:\{*\}} 1.0 \colon X_1(f + g, f^0, e^0, i^0, g^0)$$

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.

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

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:*

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

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 $s = s_i \Rightarrow a(\boldsymbol{t})(p_1 \colon s_1 \oplus p_2 \colon s_2 \ldots \oplus p_n \colon s_n)$ (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), ..., (g_3^1 : G_3^1), (g_3^2 : G_3^2), ...,
                (g_n^1 : G_n^1), (g_n^2 : G_n^2), ...] + n
       where n = [(v, D) | ∃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), ...] + newPars
3    v' := v + [D^0 | (v, D) ← newPars, D^0 is any constant of type D]
4    done := ∅
5    toTransform := {X'_1(pars) = p_1}
6    bindings := {X'_1(pars) = p_1}
   Construction
7    while toTransform ≠ ∅ do
8      Choose an arbitrary equation (X'_i(pars) = p_i) ∈ toTransform
9      (p'_i, newProcs) := transform(p_i, pars, bindings, P, v')
10     done := done ∪ {X'_i(pars) = p'_i}
11     bindings := bindings ∪ newProcs
12     toTransform := (toTransform ∪ newProcs) \ {X'_i(pars) = p_i}
13   return (done, 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*.

```
transform(p, pars, bindings, P, v') =
1   case p = a(t)∑_{x:D} f : q
2     (q', actualPars) := normalForm(q, pars, P, v')
3     if ∃j . (X'_j(pars) = q') ∈ bindings then
4       return (a(t)∑_{x:D} f : X'_j(actualPars), ∅)
5     else
6       k := |bindings| + 1
7       return (a(t)∑_{x:D} f : X'_k(actualPars), {(X'_k(pars) = q')})
8   case p = c ⇒ q
9     (newRHS, newProcs) := transform(q, pars, bindings, P, v')
10    return (c ⇒ newRHS, newProcs)
11  case p = q1 + q2
12    (newRHS1, newProcs1) := transform(q1, pars, bindings, P, v')
13    (newRHS2, newProcs2) := transform(q2, pars, bindings ∪
                                          newProcs1, P, v')
14    return (newRHS1 + newRHS2, newProcs1 ∪ newProcs2)
15  case p = Y(t)
16    (newRHS, newProcs) := transform(RHS(Y), pars, bindings, P, v')
17    newRHS' = newRHS, with all free variables substituted by the
                  value provided for them by t
18    return (newRHS', newProcs)
19  case p = ∑_{x:D} q
20    (newRHS, newProcs) := transform(q, pars, bindings, P, v')
21    return (∑_{x:D} newRHS, 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.

```
normalForm(p, pars, P, v') =
1   case p = Y(t)
2     p' := RHS(Y)
3     actualPars := [n(v) | (v, D) ← pars]
        where
               { v^0   if v is no global variable of Y in P,
        n(v) = {       (v^0 can be found by inspecting pars and v')
               { t(i)  if v is the i^th global variable of Y in P
4     return (p', actualPars)
5   case otherwise
6     return (p, [n'(v) | (v, D) ← 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 $X_i'(\mathit{pars}) = p_i$ such that $X_i'(\mathit{pars})$ 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 $X_i'(\mathit{pars}) = p_i$ from *toTransform*, transform p_i to a strongly probabilistic bisimilar IRF p_i' using Algorithm 2, add the process $X_i'(\mathit{pars}) = p_i'$ to *done*, and remove $X_i'(\mathit{pars}) = p_i$ 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 $a(\boldsymbol{t})\sum_{x:D} f : X_i'(\mathit{actualPars})$, 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 $q = Y(v_1, v_2, \dots, v_n)$, all global variables are reset, except the ones corresponding to the original global variables of Y; for them $v_1, v_2, \dots, v_n$ 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 $P = \{X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1, \dots, X_n(\boldsymbol{g_n} : \boldsymbol{G_n}) = p_n\}$ be a prCRL specification with initial vector $\boldsymbol{v}$ for X_1. Given these inputs Algorithm 1 terminates, and the specification $P' = \{X_1'(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p_1', \dots, X_k'(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g}' : \boldsymbol{G}') = p_k'\}$ 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 equation $X'_i(\boldsymbol{g} : \boldsymbol{G}) = a(\boldsymbol{t}) \sum_{x:D} f : X'_j(e_1, \ldots, e_k)$, 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.

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 = ∅
2   forall (X'_i(g : G) = p'_i) ∈ P' do
3       S := S ∪ makeSummands(p'_i, i)
4   return X(pc : {1, ..., k}, g : G) = Σ_{s∈S} s

    where
      makeSummands(p, i) =
5       case p = a(t) Σ_{x:D} f : X'_j(e_1, ..., e_k)
6           return {pc = i ⇒ a(t) Σ_{x:D} f : X(j, e_1, ..., e_k)}
7       case p = c ⇒ q
8           return {c ⇒ q' | q' ∈ makeSummands(q, i)}
9       case p = q_1 + q_2
10          return makeSummands(q_1, i) ∪ makeSummands(q_2, i)
11      case p = Σ_{x:D} q
12          return {Σ_{x:D} q' | q' ∈ makeSummands(q, i)}
```

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.

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

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

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

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

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, $E, H \subseteq$ Act, and R: Act $\to$ 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 γ: Act $\times$ Act $\to$ 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

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

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

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.

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

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

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

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

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

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

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

where

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

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

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 [29]), 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 Appendix G. Computing the state space we obtain 3763 states and 6158 transitions. Due to the uniform linear format, we can now apply several classical reduction techniques to the result. Here we will demonstrate the applicability of four such techniques using one of the summands as an example:

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

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 Appendix G for the details). The summand discussed above became:

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

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

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

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

[1]The implementation can be found at http://fmt.cs.utwente.nl/tools/prcrl.

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

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

[29] 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.

[30] M. Stoelinga, "Alea jacta est: verification of probabilistic, real-time and parametric systems," Ph.D. dissertation, University of Nijmegen, 2002.

APPENDIX

This appendix provides proofs for all propositions and theorems that are stated in the paper.

We start with a general lemma, showing that strong probabilistic bisimulation is a congruence for nondeterministic choice (both $+$ and $\sum$) and implication. Here, a context C for a process term p is a valuation of all p's free variables.

Lemma 1. *Let $p, p', q,$ and q' be (possibly open) prCRL process terms such that $p \approx p'$ and $q \approx q'$ in every context C. Let c be a condition and D some data type, then in every context also*

$$p + q \approx p' + q' \quad (1)$$

$$\sum_{x:D} p \approx \sum_{x:D} p' \quad (2)$$

$$c \Rightarrow p \approx c \Rightarrow p' \quad (3)$$

Proof: Let C be an arbitrary context, and let R_p and R_q be the bisimulation relations for p and p', and q and q', respectively.

(1) Let R be the symmetric, reflexive, transitive closure of $R_p \cup R_q \cup \{(p + q, p' + q')\}$. We will now prove that R is a bisimulation relation, thereby showing that indeed $p + q \approx p' + q'$. As we chose C to be an arbitrary context, this then holds for all contexts.

Let $p+q \xrightarrow{\alpha} \mu$. We then prove that indeed also $p'+q' \xrightarrow{\alpha} \mu'$ such that $\mu \sim_R \mu'$. Note that by the operational semantics $p + q$ can exactly do the union of the transitions that p and q can do, so either $p \xrightarrow{\alpha} \mu$ or $q \xrightarrow{\alpha} \mu$. We assume the first possibility without loss of generality. Now, since $p \approx p'$ (by the bisimulation relation R_p), we know that $p' \xrightarrow{\alpha} \mu'$ such that $\mu \sim_{R_p} \mu'$. As $p' + q'$ can exactly do the union of the transitions that p' and q' can do, also $p' + q' \xrightarrow{\alpha} \mu'$. Moreover, as bisimulation relations are equivalence relations and $R_p \subseteq R$, $\mu \sim_{R_p} \mu'$ implies that $\mu \sim_R \mu'$ (using Proposition 5.2.1 of [30]).

By symmetry $p' + q' \xrightarrow{\alpha} \mu$ implies that $p + q \xrightarrow{\alpha} \mu'$ such that $\mu \sim_R \mu'$. Moreover, for all other elements of R the required implications follow from the assumption that R_p and R_q are bisimulation relations.

(2) Let R be the symmetric, reflexive, transitive closure of $R_p \cup \{(\sum_{x:D} p, \sum_{x:D} p')\}$, then R is a bisimulation relation. First, for all $(s, t) \in R_p$ the required implications immediately follow from the assumption that R_p is a bisimulation relation. Second, by the operational semantics, $\sum_{x:D} p \xrightarrow{\alpha} \mu$ if and only if there is a $d \in D$ such that $p[x := d] \xrightarrow{\alpha} \mu$. From the assumption that $p \approx p'$ in any context it immediately follows that $p[x := d] \approx p'[x := d]$ for any $d \in D$, so if $p[x := d] \xrightarrow{\alpha} \mu$ then $p'[x := d] \xrightarrow{\alpha} \mu'$ with $\mu \sim_{R_p} \mu'$. Now, using symmetry and Proposition 5.2.1 of [30] again, statement (2) follows.

(3) If c holds in C, then $(c \Rightarrow p) = p$ and $(c \Rightarrow p') = p'$. As we assumed that $p \approx p'$, trivially $c \Rightarrow p \approx c \Rightarrow p'$. If c does not hold in C, then both $c \Rightarrow p$ and $c \Rightarrow p'$ cannot do any transitions; therefore, $c \Rightarrow p \approx c \Rightarrow p'$ by the trivial bisimulation relation $\{(c \Rightarrow p, c \Rightarrow p')\}$. ■

A. Proof of Proposition 1

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

Proof: Recall that in Definition 2 we required f to be a real-valued expression yielding values in $[0, 1]$ such that $\sum_{d \in D} f[x := d] = 1$. Also recall from Table I that μ is defined by PSUM by

$$\forall d \in D \, . \, \mu(p[x := d]) = \sum_{\substack{d' \in D \\ p[x:=d]=p[x:=d']}} f[x := d'].$$

To prove that μ is a probability distribution function, it should hold that $\mu \colon S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$, where the state space S consists of all process terms.

First of all, note that μ is only defined to be nonzero for process terms p' that can be found by evaluating $p[x := d]$ for some $d \in D$. Let $P = \{p[x := d] \mid d \in D\}$ be the set of these process terms. Now, indeed,

$$\begin{aligned}
\sum_{p' \in P} \mu(p') &= \sum_{p' \in P} \sum_{\substack{d' \in D \\ p'=p[x:=d']}} f[x := d'] \\
&= \sum_{d' \in D} \sum_{\substack{p' \in P \\ p'=p[x:=d']}} f[x := d'] \\
&= \sum_{d' \in D} f[x := d'] \\
&= 1
\end{aligned}$$

In the first step we apply the definition of μ; in the second we interchange the summand indices (which is allowed because $f[x := d']$ is always non-negative); in the third step we omit the second summation as for every $d' \in D$ there is exactly one $p' \in P$ satisfying $p' = p[x := d']$; in the fourth step we use the assumption on f. ■

B. Proof of Theorem 1

Before proving Theorem 1, we first prove two lemmas. The first lemma proves termination of Algorithm 1, and the second provides an invariant for its loop. For the first lemma we need to introduce *subterms* of process terms and specifications.

Definition 6. *Let p be a process term, then a* subterm *of p is a process term complying to the syntax of prCRL and syntactically occurring in p. The set of all subterms of p is denoted by subterms(p).*

Let P be a specification, then we define subterms$(P) = \{p \mid \exists (X_i(\boldsymbol{g_i}) = p_i) \in P \, . \, p \in$ subterms$(p_i)\}$.

Lemma 2. *Algorithm 1 terminates for every finite specification P and initial vector $\boldsymbol{v}$.*

Proof: The algorithm terminates when *toTransform* eventually becomes empty. First of all note that every iteration removes exactly one element from *toTransform*. So, if the total number of additions to *toTransform* is finite (and the call to Algorithm 2 never goes into an infinite recursion), the algorithm will terminate.

The elements that are added to *toTransform* are of the form $X'_i(\mathit{pars}) = p_i$, where $p_i \in \text{subterms}(P)$. Since P is a finite set of equations with finite right-hand sides, there exists only a finite number of such p_i. Moreover, every process equation $X'_i(\mathit{pars}) = p_i$ that is added to *toTransform* is also added to *bindings*. This makes sure that no process equation $X'_k(\mathit{pars}) = p_i$ is ever added to *toTransform* again, as can be observed from line 3 of Algorithm 2. Hence, the total number of possible additions to *toTransform* is finite.

The fact the Algorithm 2 always terminates relies on not allowing specifications with unguarded recursion. After all, the base case of Algorithm 2 is the action prefix. Therefore, when every recursion in a specification is guarded at some point by an action prefix, this base case is always reached eventually. ■

Lemma 3. *Let $P = \{X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1, \dots, X_n(\boldsymbol{g_n} : \boldsymbol{G_n}) = p_n\}$ with initial vector $\boldsymbol{v}$ for X_1 be the input prCRL specification for Algorithm 1, and let* done $= \{X'_1(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_1, \dots, X'_k(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_k\}$ *be the intermediate specification before or after an arbitrary iteration of the algorithm's while loop. Moreover, let $\boldsymbol{v'}$ be the computed new initial vector. Then, the process $X'_1(\boldsymbol{v'})$ in the prCRL specification* done $\cup$ toTransform $\cup\, P$ *is strongly probabilistic bisimilar to $X_1(\boldsymbol{v})$ in P.*

Proof: For brevity, in this proof we will write 'bisimilar' as an abbreviation for 'strongly probabilistic bisimilar in any context'. Also, the notation $p \approx q$ will be used to denote that p and q are strongly probabilistic bisimilar in any context.

We prove this lemma by induction on the number of iterations that have already been performed. Let $P = \{X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1, \dots,\ X_n(\boldsymbol{g_n} : \boldsymbol{G_n}) = p_n\}$ and $\boldsymbol{v}$ be arbitrary.

Before the first iteration, the initialisation makes sure that *done* $= \emptyset$ and *toTransform* $= \{X'_1(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p_1\}$. Also, the parameters of the new processes are determined. Every process will have the same parameters; the union of all parameters of the original processes, extended with a parameter for every nondeterministic or probabilistic sum binding a variable that is used later on. Also, the new initial state vector $\boldsymbol{v'}$ is computed by taking the original initial vector $\boldsymbol{v}$, and appending dummy values for all newly added parameters.

Clearly, $X'_1(\boldsymbol{v'})$ is identical to $X_1(\boldsymbol{v})$, except that it has more global variables (without overlap, as we assumed specifications to have unique variable names). However, these additional global variables are not used in p_1, otherwise they would be unbound in $X_1(\boldsymbol{g_1} : \boldsymbol{G_1}) = p_1$ (which is not allowed by Definition 2). Therefore, $X'_1(\boldsymbol{v'})$ and $X_1(\boldsymbol{v})$ are obviously bisimilar.

Now assume that k iterations have passed. Without loss of generality, assume that each time a process $(X'_i(\mathit{pars}) = p_i) \in$ *toTransform* had to be chosen, it was the one with the smallest i. Then, after these k iterations, *done* $= \{X'_1(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_1, \dots, X'_k(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_k\}$. Also, *toTransform* $= \{X'_{k+1}(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_{k+1}, \dots, X'_l(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_l\}$ for some l. The induction hypothesis is that $X'_1(\boldsymbol{v'})$ in *done* $\cup$ *toTransform* $\cup\, P$ is bisimilar to $X_1(\boldsymbol{v})$ in P.

We prove that after $k+1$ iterations, $X'_1(\boldsymbol{v'})$ in *done* $\cup$ *toTransform* $\cup\, P$ is still bisimilar to $X_1(\boldsymbol{v})$ in P. Note that during iteration $k+1$ three things happen: (1) the process equation $X'_{k+1}(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_{k+1}$ is removed from *toTransform*; (2) a new equation $X'_{k+1}(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p''_{k+1}$ is added to *done*; (3) potentially, one or more new equations $X'_{l+1}(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_{l+1}, \dots, X'_m(\boldsymbol{g_1} : \boldsymbol{G_1}, \boldsymbol{g'} : \boldsymbol{G'}) = p'_m$ are added to *toTransform*.

As the other equations in *done* $\cup$ *toTransform* $\cup\, P$ do not change, $X'_1(\boldsymbol{v'})$ in *done* $\cup$ *toTransform* $\cup\, P$ is still bisimilar to $X_1(\boldsymbol{v})$ in P if and only if $p'_{k+1} \approx p''_{k+1}$. We show that these process terms are indeed bisimilar by induction on the structure of p'_{k+1}.

The base case is $p'_{k+1} = a(\boldsymbol{t}) \sum_{x:D} f \colon q$. We now make a case distinction based on whether there already is a process equation in either *done* or *toTransform* whose right-hand side is the normal form of q (which is, as defined before, just q when q is not a process instantiation, otherwise it is the right-hand side of the process it instantiates), as indicated by the variable *bindings*.

Case 1a: *There does not already exist a process equation $X'_j(\mathit{pars}) = q'$ in* bindings *such that q' is the normal form of q.*

In this case, a new process equation $X'_{l+1}(\mathit{pars}) = q'$ is added to *toTransform* via line 7 of Algorithm 2, and $p''_{k+1} = a(\boldsymbol{t}) \sum_{x:D} f \colon X'_{l+1}(\mathit{actualPars})$.
When q was *not* a process instantiation, the actual parameters for X'_{l+1} are just the unchanged global variables, with those that are not used in q reset (line 6 of Algorithm 3). As (by the definition of the normal form) the right-hand side of X'_{l+1} is identical to q, the behaviour of p''_{k+1} is obviously identical to the behaviour of p'_{k+1}, i.e., they are bisimilar.
When $q = Y(v_1, v_2, \dots, v_n)$, there should occur some substitutions to ascertain that $X'_{l+1}(\mathit{actualPars})$ is bisimilar to q. Since $X'_{l+1}(\mathit{actualPars}) = q'$, with q' the right-hand side of Y, the actual parameters to be provided to X'_{l+1} should include $v_1, v_2, \dots, v_n$ for the

global variables of X'_{l+1} that correspond to the original global variables of Y. All other global variables can be reset, as they cannot be used by Y anyway. This indeed happens in line 3 of Algorithm 3, so $p''_{k+1} \approx p'_{k+1}$.

Case 1b: *There already exists a process equation* $X'_j(\mathit{pars}) = q'$ *in* bindings *such that* q' *is the normal form of* q.

In this case, we obtain $p''_{k+1} = a(\boldsymbol{t}) \sum_{x:D} f \colon X'_j(\mathit{actualPars})$ from line 4 of Algorithm 2. Note that the fact that $X'_j(\mathit{pars}) = q'$ is in *bindings* implies that at some point $X'_j(\mathit{pars}) = q'$ was in *toTransform*. In case it was already transformed in an earlier iteration there is now a process $X'_j(\mathit{pars}) = q''$ in *done* such that $q'' \approx q'$. Otherwise, $X'_j(\mathit{pars}) = q'$ is still in *toTransform*.

In both cases, *done* $\cup$ *toTransform* $\cup$ P contains a process $X'_j(\mathit{pars}) = q''$ such that $q'' \approx q'$, and therefore it is correct to take $p''_{k+1} = a(\boldsymbol{t}) \sum_{x:D} f \colon X'_j(\mathit{actualPars})$. The reasoning to see that indeed $p''_{k+1} \approx p'_{k+1}$ then only depends on the choice of *actualPars*, and is the same as for Case 1a.

Now, assume that q_1 and q_2 are process terms for which Algorithm 2 provided the bisimilar process terms p'''_{k+1} and p''''_{k+1}. Then, we prove that p''_{k+1} (as obtained from Algorithm 2) is bisimilar to p'_{k+1} for the remaining possible structures of p'_{k+1}. In Case 2, 3 and 5 we apply Lemma 1.

Case 2: $p'_{k+1} = c \Rightarrow q_1$.

In this case, Algorithm 2 yields $p''_{k+1} = c \Rightarrow p'''_{k+1}$, which is bisimilar to p'_{k+1} since $q_1 \approx p'''_{k+1}$.

Case 3: $p'_{k+1} = q_1 + q_2$.

In this case, Algorithm 2 yields $p''_{k+1} = p'''_{k+1} + p''''_{k+1}$, which is bisimilar to p'_{k+1} since $q_1 \approx p'''_{k+1}$ and $q_2 \approx p''''_{k+1}$.

Case 4: $p'_{k+1} = Y(\boldsymbol{t})$, *where we assume that* $Y(\boldsymbol{x}) = q_1$.

In this case, Algorithm 2 yields $p''_{k+1} = p'''_{k+1}$, with $\boldsymbol{x}$ substituted by $\boldsymbol{t}$, which is bisimilar to p'_{k+1} (as it precisely follows the SOS rule INST).

Case 5: $p'_{k+1} = \sum_{x:D} q_1$.

In this case, Algorithm 2 yields $p''_{k+1} = \sum_{x:D} p'''_{k+1}$, which is bisimilar to p'_{k+1} since $q_1 \approx p'''_{k+1}$.

As in all cases the process term p''_{k+1} obtained from Algorithm 2 is strongly probabilistic bisimilar to p'_{k+1} in any context, the lemma holds. ∎

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.*

Proof: Lemma 2 already provided termination, and Lemma 3 the invariant that the process $X'_1(\boldsymbol{v}')$ in the prCRL specification *done* $\cup$ *toTransform* $\cup$ P is strongly probabilistic bisimilar to $X_1(\boldsymbol{v})$ in P. As at the end of the algorithm only the equations in *done* are returned, it remains to prove that upon termination $X'_1(\boldsymbol{v}')$ in *done* does not depend on any of the process equations in *toTransform* $\cup$ P, and that *done* is in IRF.

First of all, note that upon termination *toTransform* $= \emptyset$ by the condition of the while loop. Moreover, note that the processes that are added to *done* all have a right-hand side determined by Algorithm 2, which only produces process terms that refer to processes in *done* or *toTransform* (in line 4 and line 7). Therefore, $X'_1(\boldsymbol{v}')$ in *done* indeed can only depend on process equations in *done*.

Finally, to show that *done* is indeed in IRF, we need to prove that all probabilistic sums immediately go to a process instantiation, and that process instantiations do not occur in any other way. This is immediately clear from Algorithm 2, as process instantiations are only constructed in line 4 and line 7; there, they indeed are always preceded by a probabilistic sum. Moreover, probabilistic sums are also only constructed by these lines, and are, as required, always succeeded by a process instantiation. Finally, all processes clearly have the same list of global variables (because they are created on line 10 on Algorithm 1 using *pars*, and *pars* never changes). ∎

C. *Proof of Theorem 2*

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).*

Proof: Algorithm 4 transforms the specification $P' = \{X'_1(\boldsymbol{g} : \boldsymbol{G}) = p'_1, \ldots, X'_k(\boldsymbol{g} : \boldsymbol{G}) = p'_k\}$ to an LPPE $X(pc : \{1, \ldots, k\}, \boldsymbol{g} : \boldsymbol{G})$ by constructing one or more summands for X for every process in P'. Basically, the algorithm just introduces a program counter *pc* to keep track of the process that is currently active. That is, instead of starting in $X'_1(\boldsymbol{v})$, the system will start in $X(1, \boldsymbol{v})$. Moreover, instead of advancing to $X'_j(\boldsymbol{v})$ the system will advance to $X(j, \boldsymbol{v})$. Obviously, this has no effect on the behaviour of the system, so indeed, intuitively, $X(1, \boldsymbol{v}) \equiv X'_1(\boldsymbol{v})$.

Formally, the isomorphism h to prove the theorem is given by $h(X'_i(\boldsymbol{u})) = X(i, \boldsymbol{u})$ for every $1 \leq i \leq k$ and $\boldsymbol{u} \in \boldsymbol{G}$. Note that h clearly is a bijection.

By definition $h(X'_1(\boldsymbol{v})) = X(1, \boldsymbol{v})$. To prove that $X'_i(\boldsymbol{u}) \xrightarrow{\alpha} \mu \Leftrightarrow X(i, \boldsymbol{u}) \xrightarrow{\alpha} \mu_h$ for all $1 \leq i \leq k$ and $\boldsymbol{u} \in \boldsymbol{G}$, we assume an arbitrary $X'_l(\boldsymbol{u})$ and use induction on its structure.

The base case is $X'_l(\boldsymbol{u}) = a(\boldsymbol{t}) \sum_{x:D} f \colon X'_j(e_1, \ldots, e_k)$. For this process, Algorithm 4 constructs the summand

$pc = l \Rightarrow a(\boldsymbol{t}) \sum_{x:D} f \colon X(j, e_1, \ldots, e_k)$, As every summand constructed by the algorithm contains a condition $pc = i$, and the summands produced for $X'_l(\boldsymbol{u})$ are the only ones producing a summand with $i = l$, it follows that $X'_l(\boldsymbol{u}) \xrightarrow{\alpha} \mu$ if and only if $X(l, \boldsymbol{u}) \xrightarrow{\alpha} \mu_h$.

Now assume that $X'_l(\boldsymbol{u}) = c \Rightarrow q$. By induction, the process $X''_l(\boldsymbol{u}) = q$ would result in the construction of one or more summands such that $X''_l(\boldsymbol{u}) \equiv X(l, \boldsymbol{u})$. For $X'_l(\boldsymbol{u})$ the algorithm takes those summands, and adds the condition c to all of them. Therefore, clearly $X'_l(\boldsymbol{u}) \xrightarrow{\alpha} \mu$ if and only if $X(l, \boldsymbol{u}) \xrightarrow{\alpha} \mu_h$. Similar arguments show that $X'_l(\boldsymbol{u}) \xrightarrow{\alpha} \mu$ if and only if $X(l, \boldsymbol{u}) \xrightarrow{\alpha} \mu_h$ when $X'_l(\boldsymbol{u}) = q_1 + q_2$ or $X'_l(\boldsymbol{u}) = \sum_{x:D} q$. Hence, $X(1, \boldsymbol{v}) \equiv X'_1(\boldsymbol{v})$.

Finally, X indeed is of the LPPE format (or actually, can easily be transformed to this format):

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

First of all, clearly it is a single process equation consisting of a set of summands. Each of these contains a number of nondeterministic sums and conditions, followed by a probabilistic sum. The only discrepancy is still that the nondeterministic sums and the conditions are not yet necessarily in the right order. However, they can easily be swapped such that all nondeterministic sums precede all conditions, since all variables were assumed to be unique (which can be achieved by α-conversion). Therefore, conditions preceding nondeterministic sums cannot depend on the variables bound by the sums, and hence the behaviour remains identical when they are swapped. Furthermore, each probabilistic sum is indeed followed by a process instantiation, as can be seen from line 6 of Algorithm 4. ■

D. Proof of Proposition 2

Before proving Proposition 2, we first formally define the notion of *size*.

Definition 7. *Let $\boldsymbol{g} = (g_1 : D_1, g_2 : D_2, \ldots, g_k : D_k)$ be a state vector, then we define size$(\boldsymbol{g}) = k$. The size of a process term is as follows:*

$$\begin{aligned}
size\big(a(\boldsymbol{t}) \textstyle\sum_{x:D} f \colon p\big) &= 1 + size(\boldsymbol{t}) + size(f) + size(p);\\
size(\textstyle\sum_{x:D} p) &= 1 + size(p);\\
size(p + q) &= 1 + size(p) + size(q);\\
size(c \Rightarrow p) &= 1 + size(c) + size(p);\\
size(Y(\boldsymbol{t})) &= 1 + size(\boldsymbol{t});\\
size(\boldsymbol{t}) &= \sum_{i \in \{1,\ldots,n\}} size(e_i)
\end{aligned}$$

where $\boldsymbol{t} = (e_1, e_2, \ldots, e_n)$.

The size of the expressions f, c and e_i are given by their number of function symbols and constants.

Furthermore, size$(X_i(\boldsymbol{g_i}) = p_i)$ = size$(\boldsymbol{g_i})$ + size(p_i). Finally, given a specification P, we define

$$size(P) = \sum_{p \in P} size(p)$$

Note that the definition of n in Proposition 2 is equal to *size*(P).

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 $O(n^2)$.*

Proof: First of all, note that

$$\big|pars\big| \leq \big|\text{subterms}'(P)\big| + \sum_{(X_i(\boldsymbol{g_i}) = p_i) \in P} \text{size}(\boldsymbol{g_i}) \leq n \quad (4)$$

after the initialisation of Algorithm 1, where we use *subterms*$'(P)$ to denote the *multiset* containing all subterms of P (considering a process term that occurs twice to count as two subterms). In the rest of this proof we will refer to the elements of this multiset when we refer to the subterms of P.

The first inequality follows from the fact that *pars* is defined to be the sequence of all $\boldsymbol{g_i}$ appended by all local variables of P (that are syntactically used). As every subterm can introduce at most one local variable, the inequality follows. The second inequality follows from the definition of n and the observation that *size*(p_i) counts the number of subterms of p plus the size of their expressions.

We first determine the worst-case time complexity of the transformation to IRF that is performed by Algorithm 1. It is easy to see that the function *transform* is called at most once for every subterm of P, so it follows from Equation 4 that the number of times this happens is in $O(n)$. The worst-case time complexity of every such call is governed by the call to *normalForm*.

The function *normalForm* checks for each global variable in *pars* whether or not it can be reset; from Equation 4 we know that the number of such variables is in $O(n)$. To check whether a global variable can be reset given a process term p, we have to examine every expression in p; as the size of the expressions is accounted for by n, this is also in $O(n)$. So, the worst-case time complexity of *normalForm* is in $O(n^2)$.

In conclusion, the worst-case time complexity of the transformation to IRF is in $O(n^3)$.

As the transformation from IRF to LPPE by Algorithm 4 is easily seen to be in $O(n)$, we find that, in total, linearisation has a worst-case time complexity in $O(n^3)$.

Every summand of the LPPE X that is constructed has a size in $O(n)$. After all, each contains a process instantiation with an expression for every global variable in *pars*, and we already saw that the number of them is in $O(n)$. Furthermore, the number of summands is bound from above by the

number of subterms of P, so this is in $O(n)$. Therefore, the size of X is in $O(n^2)$. ∎

To get an even more precise time complexity, we can define

$$m = |\text{subterms}'(P)|,$$

and

$$k = |\text{subterms}'(P)| + \sum_{(X_i(\boldsymbol{g_i})=p_i)\in P} \text{size}(\boldsymbol{g_i}).$$

Then, it follows from the reasoning above that the worst-case time complexity of linearisation is in $O(m \cdot k \cdot n)$.

E. Proof of Proposition 3

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

Proof: The only processes an LPPE $Z(\boldsymbol{v}, \boldsymbol{v}')$ can evolve in are processes of the form $Z(\hat{\boldsymbol{v}}, \hat{\boldsymbol{v}}')$. Moreover, the only processes a parallel composition $X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}')$ can evolve in are processes of the form $X(\hat{\boldsymbol{v}}) \,||\, Y(\hat{\boldsymbol{v}}')$. Therefore, the isomorphism h needed to prove the proposition is trivial: for all $\boldsymbol{v}, \boldsymbol{v}' \in \boldsymbol{G}$, we define $h(X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}')) = Z(\boldsymbol{v}, \boldsymbol{v}')$. Clearly, h is bijective. We will now show that indeed $X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}') \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if $Z(\boldsymbol{v}, \boldsymbol{v}') \xrightarrow{a(\boldsymbol{q})} \mu_h$.

Let $\boldsymbol{v} \in \boldsymbol{G}$ and $\boldsymbol{v}' \in \boldsymbol{G}'$ be arbitrary global variable vectors for X and Y. Then, by the operational semantics $X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}') \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if at least one of the following three conditions holds.

(1) $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu' \wedge \forall \hat{\boldsymbol{v}} \in \boldsymbol{G} \,.\, \mu(X(\hat{\boldsymbol{v}}) \,||\, Y(\boldsymbol{v}')) = \mu'(\hat{\boldsymbol{v}})$

(2) $Y(\boldsymbol{v}') \xrightarrow{a(\boldsymbol{q})} \mu' \wedge \forall \hat{\boldsymbol{v}}' \in \boldsymbol{G}' \,.\, \mu(X(\boldsymbol{v}) \,||\, Y(\hat{\boldsymbol{v}}')) = \mu'(\hat{\boldsymbol{v}}')$

(3) $X(\boldsymbol{v}) \xrightarrow{a'(\boldsymbol{q})} \mu' \wedge Y(\boldsymbol{v}') \xrightarrow{a''(\boldsymbol{q})} \mu'' \wedge \gamma(a', a'') = a \wedge$
$\forall \hat{\boldsymbol{v}} : \boldsymbol{G}, \hat{\boldsymbol{v}}' : \boldsymbol{G}' \,.\, \mu(X(\hat{\boldsymbol{v}}) \,||\, Y(\hat{\boldsymbol{v}}')) = \mu'(\hat{\boldsymbol{v}}) \cdot \mu''(\hat{\boldsymbol{v}}')$

It immediately follows from the operational semantics that $Z(\hat{\boldsymbol{v}}, \hat{\boldsymbol{v}}') \xrightarrow{a(\boldsymbol{q})} \mu_h$ is enabled under exactly the same conditions, as condition (1) is covered by the first set of summands of Z, condition (2) is covered by the second set of summands of Z, and condition (3) is covered by the third set of summands of Z. ∎

F. Proof of Proposition 4

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 $W(\boldsymbol{v}) \equiv \partial_E(X(\boldsymbol{v}))$.*

Proof: The only processes an LPPE $X(\boldsymbol{v})$ can evolve in are processes of the form $X(\boldsymbol{v}')$. Moreover, as hiding does not change the process structure, the only processes that $\tau_H(X(\boldsymbol{v}))$ can evolve in are processes of the form $\tau_H(X(\boldsymbol{v}'))$. Similar arguments hold for renaming and encapsulation. Therefore, the three isomorphisms h, h' and h'' needed to prove the proposition are trivial: for all $\boldsymbol{v} \in \boldsymbol{G}$, we define $h(\tau_H(X(\boldsymbol{v}))) = U(\boldsymbol{v})$, $h'(\rho_R(X(\boldsymbol{v}))) = V(\boldsymbol{v})$, and $h''(\partial_E(X(\boldsymbol{v}))) = W(\boldsymbol{v})$. Clearly, they are all bijective.

We will now show that h, h' and h'' are indeed isomorphisms. In all three proofs we will use the fact that $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if there is a summand $i \in I$ such that

$$\exists \boldsymbol{d_i'} \in \boldsymbol{D_i} \,.\, c_i(\boldsymbol{v}, \boldsymbol{d_i'}) \wedge a_i(\boldsymbol{b_i}(\boldsymbol{v}, \boldsymbol{d_i'})) = a(\boldsymbol{q}) \wedge \forall \boldsymbol{e_i'} \in \boldsymbol{E_i} \,.\, \mu(\boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'})) = \sum_{\substack{\boldsymbol{e_i''} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'}) = \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''})}} f_i(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''}).$$

1) We show that h is an isomorphism, by showing that $\tau_H(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if $h(\tau_H(X(\boldsymbol{v}))) \xrightarrow{a(\boldsymbol{q})} \mu_h$, i.e., if and only if $U(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_h$.
First assume that $a \neq \tau$. By the operational semantics, $\tau_H(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled and $a \notin H$. Moreover, $U(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_h$ is enabled if and only if there is a summand $i \in I$ such that
$$\exists \boldsymbol{d_i'} \in \boldsymbol{D_i} \,.\, c_i(\boldsymbol{v}, \boldsymbol{d_i'}) \wedge a_i'(\boldsymbol{b_i'}(\boldsymbol{v}, \boldsymbol{d_i'})) = a(\boldsymbol{q}) \wedge \forall \boldsymbol{e_i'} \in \boldsymbol{E_i} \,.\, \mu(\boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'})) = \sum_{\substack{\boldsymbol{e_i''} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'}) = \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''})}} f_i(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''}),$$
which indeed exactly corresponds to $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu \wedge a \notin H$ by the definition of a_i' and $\boldsymbol{b_i'}$ and the assumption that $a \neq \tau$.
Now assume that $a = \tau$ and $\boldsymbol{q} = []$. By the operational semantics, $\tau_H(X(\boldsymbol{v})) \xrightarrow{\tau} \mu$ is enabled if and only if $X(\boldsymbol{v}) \xrightarrow{\tau} \mu$ is enabled or there exists some $a \in H$ with parameters $\boldsymbol{q}'$ such that $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q}')} \mu$ is enabled. It immediately follows from the definitions of a_i' and $\boldsymbol{b_i}'$ that $U(\boldsymbol{v}) \xrightarrow{\tau} \mu_h$ is enabled under exactly these conditions.

2) We show that h' is an isomorphism, by showing that $\rho_R(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if $h'(\rho_R(X(\boldsymbol{v}))) \xrightarrow{a(\boldsymbol{q})} \mu_{h'}$, i.e., if and only if $V(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_{h'}$.
By the operational semantics, $\rho_R(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if there is a $b \in \text{Act}$ such that $X(\boldsymbol{v}) \xrightarrow{b(\boldsymbol{q})} \mu$ is enabled and $R(b) = a$. Moreover, $V(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_{h'}$ is enabled if and only if there is a summand $i \in I$ such that
$$\exists \boldsymbol{d_i'} \in \boldsymbol{D_i} \,.\, c_i(\boldsymbol{v}, \boldsymbol{d_i'}) \wedge R(a_i)(\boldsymbol{b_i}(\boldsymbol{v}, \boldsymbol{d_i'})) = a(\boldsymbol{q}) \wedge \forall \boldsymbol{e_i'} \in \boldsymbol{E_i} \,.\, \mu(\boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'})) = \sum_{\substack{\boldsymbol{e_i''} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'}) = \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''})}} f_i(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''}),$$
which indeed exactly corresponds to $\exists b \in \text{Act} \,.\, X(\boldsymbol{v}) \xrightarrow{b(\boldsymbol{q})} \mu \wedge R(b) = a$.

3) We show that h'' is an isomorphism, by showing that $\partial_E(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if $h''(\partial_E(X(\boldsymbol{v}))) \xrightarrow{a(\boldsymbol{q})} \mu_{h''}$, i.e., if and only if $W(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_{h''}$.

By the operational semantics, $\partial_E(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled and $a \notin E$. Moreover, $W(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu_{h''}$ is enabled if and only if there is a summand $i \in I$ such that $a_i \notin E$ and

$$\exists \boldsymbol{d_i'} \in \boldsymbol{D_i} \,.\, c_i(\boldsymbol{v}, \boldsymbol{d_i'}) \wedge a_i(\boldsymbol{b_i}(\boldsymbol{v}, \boldsymbol{d_i'})) = a(\boldsymbol{q}) \wedge \\ \forall \boldsymbol{e_i'} \in \boldsymbol{E_i} \,.\, \mu(\boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'})) = \\ \sum_{\substack{\boldsymbol{e_i''} \in \boldsymbol{E_i} \\ \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i'}) = \boldsymbol{n_i}(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''})}} f_i(\boldsymbol{v}, \boldsymbol{d_i'}, \boldsymbol{e_i''}),$$

which indeed exactly corresponds to $X(\boldsymbol{v'}) \xrightarrow{a(\boldsymbol{q})} \mu \wedge a \notin E$. ■

G. Case study

The LPPE obtained by linearising the prCRL specification of Figure 2 (page 9) is shown in Figure 3. Its initial vector is $(1, \text{one}, 1, \text{false}, 1, 1, \text{one}, 1, 1, 1, \text{two}, 1, \text{false}, 1, 1, \text{two}, 1, 1)$.

Initial vector.

Note that some of the initial values were chosen arbitrarily, whereas others should be exactly like this for the LPPE to be strongly probabilistic bisimilar to S. From the model in Figure 2 it follows that the values of both passive threads (*val11* and *val12*) should be initialised to 1, and that both *set11* and *set12* should be initialised to *false*. The identifiers *id11* and *id21* both belong to the first component, and are therefore initialised to *one*. The identifiers *id12* and *id22* belong to the second component and are therefore initialised to *two*. Moreover, as stated by Theorem 2, program counters (in this case *pc11, pc12, pc21*, and *pc22*) are always initialised to 1. The remaining parameters were introduced by the linearisation algorithm, and could therefore be initialised in any way. We choose to initialise them all to 1.

Representation of the LPPE.

For readability, in Figure 3 we use a slightly different notation for the process instantiations; instead of showing all parameters, we only show the parameters that are updated. For instance, *pc11* $:= 1$ denotes that the parameter *pc11* is instantiated by the value 1. As summations can use an existing parameter name for a local variable, statements such as *d22* $:=$ *d22* occur. This means that, in the next state, the global parameter *d22* will have the value of the local variable *d22*. Moreover, we use the notation $\text{reset}(x)$ to denote that the variable x is reset to its initial value.

Reductions.

On this result we applied several reductions, resulting in the LPPE shown in Figure 4. Its initial vector is $(1, \text{false}, 1, 1, 1, 1, \text{false}, 1, 1, 1)$. In Section VII we already provided a high-level discussion of the reductions we applied. Here, we will explicitly mention all the changes that have been made to the LPPE.

Constant elimination. Looking at the entire LPPE, it turns out that the parameters *pc11*, *pc12*, *id11*, *id12*, *id21*, *id22*, *d11* and *d12* are constant. Although they sometimes are assigned a 'new' value in the next-state function, this always corresponds to their initial values. Therefore, wherever used they could be replaced by their initial values and removed as parameters.

Summation elimination. All nondeterministic sums are followed by a condition restricting their 'choice' to a single value. Therefore, all these sums could be removed. As a result, we also had to remove the corresponding restricting conditions, and substituted the single choice for all occurrences of the variable that was summed over.

Data evaluation / syntactic clean-up. After constant elimination many constraints turned out to be tautologies, such as $1 = 1$, and consequently could be removed. Also, the function *other* could now be evaluated. After this step two summands were removed, as they where conditioned by the unsatisfiable *one* $=$ *two*. Then, we did a syntactic clean-up, changing the probabilistic sums over $(k1, k2) : \{*\} \times \{*\}$ to sums over $k : \{*\}$, and evaluating *multiply*$(1.0, 1.0)$ to 1.0.

Liveness analysis. Note that all these changes do not yet provide any state space reduction, but they do make the LPPE much more readable. However, as a final step we did achieve state space reduction by manually applying the liveness analysis techniques from [20]. This resulted in the observation that the parameter *val11* can be reset in the fifth summand, and that the parameter *val12* can be reset in the sixth.

$$
\begin{aligned}
&Z(\mathit{pc11} : \{1..1\}, \mathit{id11} : \mathit{Id}, \mathit{val11} : \mathit{Die}, \mathit{set11} : \mathit{Bool}, \mathit{d11} : \mathit{Die}, \mathit{pc21} : \{1..4\}, \mathit{id21} : \mathit{Id}, \mathit{d21} : \mathit{Die}, \mathit{e21} : \mathit{Die},\\
&\quad \mathit{pc12} : \{1..1\}, \mathit{id12} : \mathit{Id}, \mathit{val12} : \mathit{Die}, \mathit{set12} : \mathit{Bool}, \mathit{d12} : \mathit{Die}, \mathit{pc22} : \{1..4\}, \mathit{id22} : \mathit{Id}, \mathit{d22} : \mathit{Die}, \mathit{e22} : \mathit{Die}) =\\
&\qquad\qquad \mathit{pc21} = 1 \Rightarrow\\
&\qquad\qquad \mathit{roll}(\mathit{id21}) \sum^{\bullet}_{\mathit{d21}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc21} := 2, \mathit{d21} := \mathit{d21}, \mathrm{reset}(\mathit{e21}))\\
&\quad + \qquad \mathit{pc22} = 1 \Rightarrow\\
&\qquad\qquad \mathit{roll}(\mathit{id22}) \sum^{\bullet}_{\mathit{d22}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc22} := 2, \mathit{d22} := \mathit{d22}, \mathrm{reset}(\mathit{e22}))\\
&\quad + \sum_{\mathit{d11}:\mathit{Die}} \mathit{pc21} = 2 \land \mathit{pc11} = 1 \land \neg \mathit{set11} \land \mathit{id11} = \mathit{other}(\mathit{id21}) \land \mathit{other}(\mathit{id11}) = \mathit{id21} \land \mathit{d11} = \mathit{d21} \Rightarrow\\
&\qquad\qquad \mathit{comm}(\mathit{id11}, \mathit{other}(\mathit{id11}), \mathit{d11}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc11} := 1, \mathit{val11} := \mathit{d11}, \mathit{set11} := \mathit{true}, \mathrm{reset}(\mathit{d11}), \mathit{pc21} := 3,\\
&\qquad\qquad\qquad \mathrm{reset}(\mathit{e21}))\\
&\quad + \sum_{\mathit{d12}:\mathit{Die}} \mathit{pc22} = 2 \land \mathit{pc12} = 1 \land \neg \mathit{set12} \land \mathit{id12} = \mathit{other}(\mathit{id22}) \land \mathit{other}(\mathit{id12}) = \mathit{id22} \land \mathit{d12} = \mathit{d22} \Rightarrow\\
&\qquad\qquad \mathit{comm}(\mathit{id12}, \mathit{other}(\mathit{id12}), \mathit{d12}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc12} := 1, \mathrm{val12} := \mathit{d12}, \mathit{set12} := \mathit{true}, \mathrm{reset}(\mathit{d12}), \mathit{pc22} := 3,\\
&\qquad\qquad\qquad \mathrm{reset}(\mathit{e22}))\\
&\quad + \sum_{\mathit{d12}:\mathit{Die}} \mathit{pc21} = 2 \land \mathit{pc12} = 1 \land \neg \mathit{set12} \land \mathit{other}(\mathit{id21}) = \mathit{id12} \land \mathit{id21} = \mathit{other}(\mathit{id12}) \land \mathit{d21} = \mathit{d12} \Rightarrow\\
&\qquad\qquad \mathit{comm}(\mathit{other}(\mathit{id21}), \mathit{id21}, \mathit{d21}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc21} := 3, \mathrm{reset}(\mathit{e21}), \mathit{pc12} := 1, \mathit{val12} := \mathit{d12}, \mathit{set12} := \mathit{true},\\
&\qquad\qquad\qquad \mathrm{reset}(\mathit{d12}))\\
&\quad + \sum_{\mathit{d11}:\mathit{Die}} \mathit{pc22} = 2 \land \mathit{pc11} = 1 \land \neg \mathit{set11} \land \mathit{id11} = \mathit{other}(\mathit{id22}) \land \mathit{other}(\mathit{id11}) = \mathit{id22} \land \mathit{d11} = \mathit{d22} \Rightarrow\\
&\qquad\qquad \mathit{comm}(\mathit{id11}, \mathit{other}(\mathit{id11}), \mathit{d11}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc11} := 1, \mathit{val11} := \mathit{d11}, \mathit{set11} := \mathit{true}, \mathrm{reset}(\mathit{d11}), \mathit{pc22} := 3,\\
&\qquad\qquad\qquad \mathrm{reset}(\mathit{e22}))\\
&\quad + \sum_{\mathit{e21}:\mathit{Die}} \mathit{pc21} = 3 \land \mathit{pc11} = 1 \land \mathit{set11} \land \mathit{val11} = \mathit{e21} \Rightarrow\\
&\qquad\qquad \mathit{checkVal}(\mathit{val11}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc11} := 1, \mathit{set11} := \mathit{false}, \mathrm{reset}(\mathit{d11}), \mathit{pc21} := 4, \mathit{e21} := \mathit{e21})\\
&\quad + \sum_{\mathit{e22}:\mathit{Die}} \mathit{pc22} = 3 \land \mathit{pc12} = 1 \land \mathit{set12} \land \mathit{val12} = \mathit{e22} \Rightarrow\\
&\qquad\qquad \mathit{checkVal}(\mathit{val12}) \sum^{\bullet}_{(k1,k2):\{*\}\times\{*\}} \mathit{multiply}(1.0, 1.0) : Z(\mathit{pc12} := 1, \mathit{set12} := \mathit{false}, \mathrm{reset}(\mathit{d12}), \mathit{pc22} := 4, \mathit{e22} := \mathit{e22})\\
&\quad + \qquad \mathit{pc21} = 4 \land \mathit{d21} = \mathit{e21} \Rightarrow\\
&\qquad\qquad \mathit{roll}(\mathit{id21}) \sum^{\bullet}_{\mathit{d21}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc21} := 2, \mathit{d21} := \mathit{d21}, \mathrm{reset}(\mathit{e21}))\\
&\quad + \qquad \mathit{pc22} = 4 \land \mathit{d22} = \mathit{e22} \Rightarrow\\
&\qquad\qquad \mathit{roll}(\mathit{id22}) \sum^{\bullet}_{\mathit{d22}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc22} := 2, \mathit{d22} := \mathit{d22}, \mathrm{reset}(\mathit{e22}))\\
&\quad + \qquad \mathit{pc21} = 4 \land \mathit{d21} > \mathit{e21} \Rightarrow\\
&\qquad\qquad \mathit{leader}(\mathit{id21}) \sum^{\bullet}_{k:\{*\}} 1.0 : Z(\mathit{pc21} := 1, \mathrm{reset}(\mathit{d21}), \mathrm{reset}(\mathit{e21}))\\
&\quad + \qquad \mathit{pc22} = 4 \land \mathit{d22} > \mathit{e22} \Rightarrow\\
&\qquad\qquad \mathit{leader}(\mathit{id22}) \sum^{\bullet}_{k:\{*\}} 1.0 : Z(\mathit{pc22} := 1, \mathrm{reset}(\mathit{d22}), \mathrm{reset}(\mathit{e22}))\\
&\quad + \qquad \mathit{pc21} = 4 \land \mathit{d21} < \mathit{e21} \Rightarrow\\
&\qquad\qquad \mathit{follower}(\mathit{id21}) \sum^{\bullet}_{k:\{*\}} 1.0 : Z(\mathit{pc21} := 1, \mathrm{reset}(\mathit{d21}), \mathrm{reset}(\mathit{e21}))\\
&\quad + \qquad \mathit{pc22} = 4 \land \mathit{d22} < \mathit{e22} \Rightarrow\\
&\qquad\qquad \mathit{follower}(\mathit{id22}) \sum^{\bullet}_{k:\{*\}} 1.0 : Z(\mathit{pc22} := 1, \mathrm{reset}(\mathit{d22}), \mathrm{reset}(\mathit{e22}))
\end{aligned}
$$

Figure 3. The LPPE of the leader election protocol before any reductions.

$$
\begin{aligned}
&Z(\mathit{val11} : \mathit{Die}, \mathit{set11} : \mathit{Bool}, \mathit{pc21} : \{\mathit{1..4}\}, \mathit{d21} : \mathit{Die}, \mathit{e21} : \mathit{Die}, \mathit{val12} : \mathit{Die}, \mathit{set12} : \mathit{Bool}, \mathit{pc22} : \{\mathit{1..4}\}, \mathit{d22} : \mathit{Die}, \mathit{e22} : \mathit{Die}) = \\
&\qquad \mathit{pc21} = 1 \Rightarrow \\
&\qquad \mathit{roll}(\mathit{one}) \sum_{\mathit{d21}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc21} := 2, \mathit{d21} := \mathit{d21}, \mathrm{reset}(\mathit{e21})) \\
&\quad + \mathit{pc22} = 1 \Rightarrow \\
&\qquad \mathit{roll}(\mathit{two}) \sum_{\mathit{d22}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc22} := 2, \mathit{d22} := \mathit{d22}, \mathrm{reset}(\mathit{e22})) \\
&\quad + \mathit{pc21} = 2 \wedge \neg \mathit{set12} \Rightarrow \\
&\qquad \mathit{comm}(\mathit{two}, \mathit{one}, \mathit{d21}) \sum_{k:\{*\}} 1.0 : Z(\mathit{pc21} := 3, \mathrm{reset}(\mathit{e21}), \mathit{val12} := \mathit{d21}, \mathit{set12} := \mathit{true}) \\
&\quad + \mathit{pc22} = 2 \wedge \neg \mathit{set11} \Rightarrow \\
&\qquad \mathit{comm}(\mathit{one}, \mathit{two}, \mathit{d22}) \sum_{k:\{*\}} 1.0 : Z(\mathit{val11} := \mathit{d22}, \mathit{set11} := \mathit{true}, \mathit{pc22} := 3, \mathrm{reset}(\mathit{e22})) \\
&\quad + \mathit{pc21} = 3 \wedge \mathit{set11} \Rightarrow \\
&\qquad \mathit{checkVal}(\mathit{val11}) \sum_{k:\{*\}} 1.0 : Z(\mathrm{reset}(\mathit{val11}), \mathit{set11} := \mathit{false}, \mathit{pc21} := 4, \mathit{e21} := \mathit{val11}) \\
&\quad + \mathit{pc22} = 3 \wedge \mathit{set12} \Rightarrow \\
&\qquad \mathit{checkVal}(\mathit{val12}) \sum_{k:\{*\}} 1.0 : Z(\mathrm{reset}(\mathit{val12}), \mathit{set12} := \mathit{false}, \mathit{pc22} := 4, \mathit{e22} := \mathit{val12}) \\
&\quad + \mathit{pc21} = 4 \wedge \mathit{d21} = \mathit{e21} \Rightarrow \\
&\qquad \mathit{roll}(\mathit{one}) \sum_{\mathit{d21}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc21} := 2, \mathit{d21} := \mathit{d21}, \mathrm{reset}(\mathit{e21})) \\
&\quad + \mathit{pc22} = 4 \wedge \mathit{d22} = \mathit{e22} \Rightarrow \\
&\qquad \mathit{roll}(\mathit{two}) \sum_{\mathit{d22}:\mathit{Die}} \tfrac{1}{6} : Z(\mathit{pc22} := 2, \mathit{d22} := \mathit{d22}, \mathrm{reset}(\mathit{e22})) \\
&\quad + \mathit{pc21} = 4 \wedge \mathit{d21} > \mathit{e21} \Rightarrow \\
&\qquad \mathit{leader}(\mathit{one}) \sum_{k:\{*\}} 1.0 : Z(\mathit{pc21} := 1, \mathrm{reset}(\mathit{d21}), \mathrm{reset}(\mathit{e21})) \\
&\quad + \mathit{pc22} = 4 \wedge \mathit{d22} > \mathit{e22} \Rightarrow \\
&\qquad \mathit{leader}(\mathit{two}) \sum_{k:\{*\}} 1.0 : Z(\mathit{pc22} := 1, \mathrm{reset}(\mathit{d22}), \mathrm{reset}(\mathit{e22})) \\
&\quad + \mathit{pc21} = 4 \wedge \mathit{d21} < \mathit{e21} \Rightarrow \\
&\qquad \mathit{follower}(\mathit{one}) \sum_{k:\{*\}} 1.0 : Z(\mathit{pc21} := 1, \mathrm{reset}(\mathit{d21}), \mathrm{reset}(\mathit{e21})) \\
&\quad + \mathit{pc22} = 4 \wedge \mathit{d22} < \mathit{e22} \Rightarrow \\
&\qquad \mathit{follower}(\mathit{two}) \sum_{k:\{*\}} 1.0 : Z(\mathit{pc22} := 1, \mathrm{reset}(\mathit{d22}), \mathrm{reset}(\mathit{e22}))
\end{aligned}
$$

Figure 4. The LPPE of the leader election protocol after reductions.