A linear process-algebraic format with data for probabilistic automata

Joost-Pieter Katoen[a,b], Jaco van de Pol[a], Mariëlle Stoelinga[a], Mark Timmer[a]

[a] *Formal Methods and Tools, Faculty of EEMCS, University of Twente, The Netherlands*
[b] *Software Modeling and Verification, RWTH Aachen University, Germany*

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, leading to the symbolic analysis of parameterised probabilistic systems. We discuss several reduction techniques that can easily be applied to our models. A validation of our approach on two benchmark leader election protocols shows reductions of more than an order of magnitude.

Keywords: probabilistic process algebra, linearisation, data-dependent probabilistic choice, symbolic transformations, state space reduction

1. Introduction

Efficient model checking algorithms exist, supported by powerful software tools, for verifying qualitative and quantitative properties for a wide range of probabilistic models. While these techniques are important for areas like security, randomised distributed algorithms, systems biology, and dependability and performance analysis, two major deficiencies exist: the *state space explosion* and the restricted treatment of *data*.

Unlike process calculi like μCRL [1] and LOTOS NT [2], which support rich data types, modelling formalisms for probabilistic systems mostly treat data as a second-class citizen. 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 only allow a random choice over a fixed distribution, and input languages for probabilistic model checkers such as the reactive module language of PRISM [3] or the probabilistic variant of Promela [4] only support basic data types, but neither support more advanced data structures. To model realistic systems, however, convenient means for data modelling are indispensable.

Additionally, although parameterised probabilistic choice is semantically well-defined [5], the incorporation of data yields a significant increase of, or even an infinite, state space. However, current probabilistic minimisation techniques are not well-suited to be applied in the presence of data: aggressive abstraction techniques for probabilistic models (e.g., [6, 7, 8, 9, 10, 11]) reduce 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 [12, 13] and the probabilistic GCL [14].

Our aim is to develop symbolic minimisation techniques — operating at the syntax level — for data-dependent probabilistic systems. We therefore define a probabilistic variant of the process-algebraic μCRL language [1], named prCRL, which treats data as a first-class citizen. 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. Because of its process-algebraic nature, message passing can be used to define systems in a more modular manner than with for instance the PRISM language.

To enable symbolic reductions, we provide a two-phase algorithm to transform prCRL terms into LPPEs: a probabilistic variant of *linear process equations* (LPEs) [15], 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 [16]. Similar linearisations have been provided for plain μCRL [17], as well as a real-time variant [18] and a hybrid variant [19] therefore.

To motivate the advantage of the LPPE 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 led to elegant proof methods, like the use of invariants for process algebra [15], and the cones and foci method for proof checking process equivalence [20, 21]. It also enabled the application of model checking techniques to process algebra, such as optimisations from static analysis [22] (including dead variable reduction [23]), data abstraction [24], distributed model checking [25], symbolic model checking (either with BDDs [26] or by constructing the product of an LPE and a parameterised μ-calculus formula [27, 28]), and confluence reduction [29] (a variant 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. It also allowed the cross-fertilisation of the various techniques by composing them as LPE to LPE transformations.

We generalise several reduction techniques from LPEs to LPPEs: constant elimination, summation elimination, expression simplification, dead variable reduction, and confluence reduction. The generalisation of these techniques turned out to be very elegant. Also, we implemented a tool that can linearise prCRL models to LPPE, automatically apply all these reduction techniques, and generate state spaces. Experimental validation, using several variations of two benchmark protocols for probabilistic model checking, show that state space reductions of up to 95% can be achieved.

Organisation of the paper. After recalling some preliminaries in Section 2, we introduce our probabilistic process algebra prCRL in Section 3. The LPPE format is defined in Section 4, and a procedure to linearise a prCRL specification to LPPE is presented in Section 5. Section 6 then introduces parallel composition on LPPEs. Section 7 discusses the reduction techniques we implemented thus far for LPPEs, and an implementation and case studies are presented in Section 8. We conclude the paper in Section 9. An appendix is provided, containing a detailed proof for our main theorem.

This paper extends an earlier conference paper [30] by (1) formal proofs for all results, (2) a comprehensive exposition of reduction techniques for LPPEs, (3) a tool implementation of all these techniques, and (4) more extensive experimental results, showing impressive reductions.

2. 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, sets of vectors and Cartesian products in bold.

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 [31].

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

- S *is a countable set of states;*
- $s^0 \in S$ *is the initial state;*
- A *is a countable set of actions;*

- $\Delta\colon S \to \mathscr{P}(A \times \mathrm{Distr}(S))$ *is the 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 2. *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.

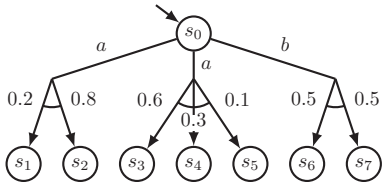

Figure 1: A probabilistic automaton.

Strong probabilistic bisimulation. Strong probabilistic bisimulation[1] [16] is a probabilistic extension of the traditional notion of bisimulation [32], equating any two processes that cannot be distinguished by an observer. It is well-known that strongly probabilistically bisimilar processes satisfy the same properties, as for instance expressed in the probabilistic temporal logic PCTL [33]. Two states s, t of a PA $\mathcal{A} = \langle S, s^0, A, \Delta\rangle$ are strongly probabilistically bisimilar (denoted by $s \approx t$) if there exists an equivalence relation $R \subseteq S \times S$ 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. Two PAs $\mathcal{A}_1, \mathcal{A}_2$ are strongly probabilistically bisimilar (denoted by $\mathcal{A}_1 \approx \mathcal{A}_2$) if their initial states are strongly probabilistically bisimilar in the disjoint union of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Isomorphism. Two states s and t of a PA $\mathcal{A} = \langle S, s^0, A, \Delta\rangle$ are *isomorphic* (denoted by $s \equiv t$) if there exists a bijection $f\colon S \to S$ such that $f(s) = t$ and $\forall s' \in S, \mu \in \mathrm{Distr}(S), a \in A\,.\, s' \xrightarrow{a} \mu \Leftrightarrow f(s') \xrightarrow{a} \mu_f$. Two PAs $\mathcal{A}_1, \mathcal{A}_2$ are isomorphic (denoted by $\mathcal{A}_1 \equiv \mathcal{A}_2$) if their initial states are isomorphic in the disjoint union of $\mathcal{A}_1$ and $\mathcal{A}_2$. Obviously, isomorphism implies strong probabilistic bisimulation.

3. A process algebra with probabilistic choice and data

3.1. 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, or a fixed data language), able to handle at least boolean expressions and real-valued expressions. Also, we assume that any expression that does not contain variables can be evaluated. Note that this restricts the expressiveness of the data language. In the examples we will use an intuitive data language, containing basic arithmetics and boolean operators. The meaning of all the functions we use will be clear.

We mostly refer to data types with upper-case letters $D, E, \ldots$, and to variables over them with lower-case letters $u, v, \ldots$. We assume the existence of a countable set of actions Act.

Definition 3. *A* process term *in prCRL is any term that can be generated by the following grammar:*

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

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

We say that a process term $Y(\boldsymbol{t})$ can go unguarded *to Y. Moreover, $c \Rightarrow p$ can go unguarded to Y if p can, $p + q$ if either p or q can, and $\sum_{\boldsymbol{x}:\boldsymbol{D}} p$ if p can, whereas $a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : p$ cannot go anywhere unguarded.*

[1] Note that Segala used the term probabilistic bisimulation when also allowing convex combinations of transitions [31]; we do not need to allow these, as the variant of strong bisimulation without them is already preserved by our procedures.

Given an expression t, a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ and a vector $\boldsymbol{d} = (d_1, \ldots, d_n)$, we use $t[\boldsymbol{x} := \boldsymbol{d}]$ to denote the expression obtained by substituting every occurrence of x_i in t by d_i. Given a process term p we use $p[\boldsymbol{x} := \boldsymbol{d}]$ to denote the process term p' obtained by substituting every expression t in p by $t[\boldsymbol{x} := \boldsymbol{d}]$.

In a process term, $Y(\boldsymbol{t})$ denotes *process instantiation*, where $\boldsymbol{t}$ instantiates Y's process variables (allowing recursion). The term $c \Rightarrow p$ behaves as p if the *condition* c holds, and cannot do anything otherwise. The $+$ operator denotes *nondeterministic choice*, and $\sum_{\boldsymbol{x}:\boldsymbol{D}} p$ a (possibly infinite) *nondeterministic choice over data type* $\boldsymbol{D}$. Finally, $a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : p$ performs the action $a(\boldsymbol{t})$ and then does a *probabilistic choice* over $\boldsymbol{D}$. It uses the value $f[\boldsymbol{x} := \boldsymbol{d}]$ as the probability of choosing each $\boldsymbol{d} \in \boldsymbol{D}$. We do not consider sequential composition of process terms (i.e., something of the form $p \cdot p$), because already in the non-probabilistic case this significantly increases the difficulty of linearisation as it requires a stack [18]. Therefore, it would distract from our main purpose: combining probabilities with data. Moreover, most specifications used in practice can be written without this form.

Definition 4. *A* prCRL specification $P = (\{X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = p_i\}, X_j(\boldsymbol{t}))$ *consists of a finite set of uniquely-named* processes X_i*, each of which is defined by a* process equation $X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = p_i$*, and an* initial process $X_j(\boldsymbol{t})$*. In a process equation, $\boldsymbol{x_i}$ is a vector of process variables with countable type $\boldsymbol{D_i}$, and p_i (the* right-hand side*) is a* process term *specifying the behaviour of X_i.*

A variable v in an expression in a right-hand side p_i is bound *if it is an element of $\boldsymbol{x_i}$ or it occurs within a construct $\sum_{\boldsymbol{x}:\boldsymbol{D}}$ or $\sum_{\boldsymbol{x}:\boldsymbol{D}}$ such that v is an element of $\boldsymbol{x}$. Variables that are not bound are said to be* free.

We mostly refer to process terms with lower-case letters p, q, r, and to processes with capitals X, Y, Z. Also, we will often write $X(x_1 : D_1, \ldots, x_n : D_n)$ for $X((x_1, \ldots, x_n) : (D_1 \times \cdots \times D_n))$.

Not all syntactically correct prCRL specifications can indeed be used to model a system in a meaningful way. The following definition states what we additionally require for them to be well-formed. The first two constraints make sure that a specification does not refer to undefined variables or processes, the third is needed to obtain valid probability distributions, and the fourth makes sure that the specification only has one unique solution (modulo strong probabilistic bisimulation).

Definition 5. *A prCRL specification $P = (\{X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = p_i\}, X_j(\boldsymbol{t}))$ is* well-formed *if the following four constraints are all satisfied:*

- *There are no free variables.*
- *There are no instantiations of undefined processes. That is, for every instantiation $Y(\boldsymbol{t'})$ occurring in some p_i, there exists a process equation $(X_k(\boldsymbol{x_k} : \boldsymbol{D_k}) = p_k) \in P$ such that $X_k = Y$ and $\boldsymbol{t'}$ is of type $\boldsymbol{D_k}$. Also, the vector $\boldsymbol{t}$ used in the initial process is of type $\boldsymbol{D_j}$.*
- *The probabilistic choices are well-defined. That is, for every construct $\sum_{\boldsymbol{x}:\boldsymbol{D}} f$ occurring in a right-hand side p_i it holds that $\sum_{\boldsymbol{d}\in\boldsymbol{D}} f[\boldsymbol{x} := \boldsymbol{d}] = 1$ for every possible valuation of the other variables that are used in f (the summation now used in the mathematical sense).*
- *There is no unguarded recursion.*[2] *That is, for every process Y, there is no sequence of processes $X_1, X_2, \ldots, X_n$ (with $n \geq 2$) such that $Y = X_1 = X_n$ and p_j can go unguarded to X_{j+1} for every $1 \leq j < n$.*

We assume from now on that every prCRL specification is well-formed.

Example 6. *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*

[2] This constraint could be relaxed a bit, as contradictory conditions of the processes might make an unguarded cycle harmless.

Table 1: SOS rules for prCRL.

$$\textsc{Inst}\ \frac{p[\boldsymbol{x} := \boldsymbol{d}] \xrightarrow{\alpha} \mu}{Y(\boldsymbol{d}) \xrightarrow{\alpha} \mu}\ \text{if } Y(\boldsymbol{x} : \boldsymbol{D}) = p \qquad \textsc{Implies}\ \frac{p \xrightarrow{\alpha} \mu}{c \Rightarrow p \xrightarrow{\alpha} \mu}\ \text{if } c \text{ equals } \mathit{true}$$

$$\textsc{NChoice-L}\ \frac{p \xrightarrow{\alpha} \mu}{p + q \xrightarrow{\alpha} \mu} \qquad \textsc{NSum}\ \frac{p[\boldsymbol{x} := \boldsymbol{d}] \xrightarrow{\alpha} \mu}{\sum_{\boldsymbol{x}:\boldsymbol{D}} p \xrightarrow{\alpha} \mu}\ \text{where } \boldsymbol{d} \in \boldsymbol{D} \qquad \textsc{NChoice-R}\ \frac{q \xrightarrow{\alpha} \mu}{p + q \xrightarrow{\alpha} \mu}$$

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

is never used. For brevity, here and in later examples we abuse notation by interpreting a single process equation as a specification (where in this case the initial process is implicit, as it can only be $X()$).

$$X() = \mathit{throw}() \sum_{x:D} \tfrac{1}{|D|} : \mathit{send}(x) \sum_{i:\{1,2\}} \mathit{if}\ i = 1\ \mathit{then}\ 0.1\ \mathit{else}\ 0.9 : (i = 1 \Rightarrow \mathit{beep}() \sum_{j:\{*\}} 1 : X()) + (i = 2 \Rightarrow X())$$

In principle, the data types used in prCRL specifications can be countably infinite. Also, infinite probabilistic choices (and therefore countably infinite branching) are allowed, as illustrated by the following example.

Example 7. *Consider a system that first writes the number* 0*, and then continuously writes natural numbers (excluding zero) in such a way that the probability of writing n is each time given by $\frac{1}{2^n}$. This system can be modelled by the prCRL specification $P = (\{X\}, X(0))$, where X is given by*

$$X(n : \mathbb{N}) = \mathit{write}(n) \sum_{m:\mathbb{N}} \tfrac{1}{2^m} : X(m)$$

3.2. Operational semantics

The operational semantics of a prCRL specification is given in terms of a PA. The states are all process terms without free variables, the initial state is the instantiation of the initial process, the action set is given by $\{a(\boldsymbol{t}) \mid a \in \mathrm{Act}, \boldsymbol{t}$ is a vector of expressions$\}$, and the transition relation is the smallest relation satisfying the SOS rules in Table 1. For brevity, we use α to denote an action name together with its parameters. A mapping to PAs is only provided for processes without free variables; this is consistent with Definition 5.

Given a prCRL specification and its underlying PA $\mathcal{A}$, two process terms are isomorphic (bisimilar) if their corresponding states in $\mathcal{A}$ are isomorphic (bisimilar). Two specifications with underlying PAs $\mathcal{A}_1, \mathcal{A}_2$ are isomorphic (bisimilar) if $\mathcal{A}_1$ is isomorphic (bisimilar) to $\mathcal{A}_2$.

Proposition 8. *The SOS-rule* PSum *defines a probability distribution μ over process terms.*

Proof. For μ to be a probability distribution function over process terms, 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 without free variables.

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

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

In the first step we apply the definition of μ from Table 1; in the second we interchange the summand indices (which is allowed because $f[\boldsymbol{x} := \boldsymbol{d}']$ is always non-negative); in the third we omit the second summation as for every $\boldsymbol{d}' \in \boldsymbol{D}$ there is exactly one $p' \in P$ satisfying $p' = p[\boldsymbol{x} := \boldsymbol{d}']$; in the fourth we use the fact that f is a real-valued expression yielding values in $[0, 1]$ such that $\sum_{\boldsymbol{d}\in\boldsymbol{D}} f[\boldsymbol{x} := \boldsymbol{d}] = 1$ (Definitions 3 and 5). □

3.3. 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 for $X()$, and a for $a()$. Moreover, we can define the syntactic sugar

$$\begin{array}{lcl} p \lhd c \rhd q & \stackrel{\text{def}}{=} & (c \Rightarrow p) + (\neg c \Rightarrow q) \\ a(\boldsymbol{t}) \cdot p & \stackrel{\text{def}}{=} & a(\boldsymbol{t}) \sum_{x:\{*\}} 1 : p \text{ (where } x \text{ is chosen such that it does not occur freely in } p) \\ a(\boldsymbol{t})\ U_{\boldsymbol{x}:\boldsymbol{D}}\, c \Rightarrow p & \stackrel{\text{def}}{=} & a(\boldsymbol{t}) \sum_{\boldsymbol{x}:\boldsymbol{D}} \Big(\text{if } c \text{ then } \frac{1}{|\{\boldsymbol{d}\in\boldsymbol{D} | c[\boldsymbol{x}:=\boldsymbol{d}]\}|} \text{ else } 0\Big) : p \end{array}$$

Note that $U_{\boldsymbol{x}:\boldsymbol{D}}\, c \Rightarrow p$ is the uniform choice among a set, choosing only from its elements that fulfil a certain condition c. For finite probabilistic sums,

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

is used to abbreviate $a(\boldsymbol{t}) \sum_{x:\{1,\ldots,n\}} f : p$, such that x does not occur freely in any p_i, $f[x := i] = u_i$ for every $1 \le i \le n$, and p is given by $(x = 1 \Rightarrow p_1) + (x = 2 \Rightarrow p_2) + \cdots + (x = n \Rightarrow p_n)$.

Example 9. *The process equation of Example 6 can now be represented as follows:*

$$X = \mathit{throw} \sum_{x:D} \tfrac{1}{|D|} : \mathit{send}(x)(0.1 : \mathit{beep} \cdot X \oplus 0.9 : X)$$

Example 10. *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 can be represented by*

$$X(s : Set_D) = \mathit{choose}\, U_{x:D}\, \mathit{contains}(s, x) \Rightarrow \mathit{send}(x) \cdot X(s),$$

where $\mathit{contains}(s, x)$ holds if s contains x.

4. A linear format for prCRL

4.1. The LPE and LPPE formats

In the non-probabilistic setting, a restricted version of μCRL that is well-suited for formal manipulation is captured by the LPE format [18]:

$$\begin{array}{rl} X(\boldsymbol{g} : \boldsymbol{G}) = & \sum_{\boldsymbol{d_1}:\boldsymbol{D_1}} c_1 \Rightarrow a_1(\boldsymbol{b_1}) \cdot X(\boldsymbol{n_1}) \\ & + \sum_{\boldsymbol{d_2}:\boldsymbol{D_2}} c_2 \Rightarrow a_2(\boldsymbol{b_2}) \cdot X(\boldsymbol{n_2}) \\ & \cdots \\ & + \sum_{\boldsymbol{d_k}:\boldsymbol{D_k}} c_k \Rightarrow a_k(\boldsymbol{b_k}) \cdot X(\boldsymbol{n_k}) \end{array}$$

Here, each of the k components is called a *summand*. Furthermore, $\boldsymbol{G}$ is a type for *state vectors* (containing the process variables, in this setting also called *global variables*), and each $\boldsymbol{D_i}$ is a type for the *local variable vector* of summand i. The summations represent nondeterministic choices 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* $\boldsymbol{n_i}$. Note that the LPE corresponds to the well-known precondition-effect style.

Example 11. *Consider a system consisting of two buffers, B_1 and B_2. Buffer B_1 reads a message of type D from the environment, and sends it synchronously to B_2. Then, B_2 writes the message. The following LPE has exactly this behaviour when initialised with $a = 1$ and $b = 1$ (x and y can be chosen arbitrarily).*

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

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 linear format for prCRL should 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 12. *An* LPPE (linear probabilistic process equation) *is a prCRL specification consisting of precisely one process, of the following format (where the outer summation is an abbreviation of the nondeterministic choice between the summands):*

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

Compared to the LPE we added a probabilistic choice over an additional vector of local variables $\boldsymbol{e_i}$. The corresponding probability expression f_i, as well as the next-state vector $\boldsymbol{n_i}$, can now also depend on $\boldsymbol{e_i}$.

As an LPPE consists of only one process, an initial process $X(\boldsymbol{v})$ can be represented by its *initial vector* $\boldsymbol{v}$. Often, we will use the same name for the specification of an LPPE and the single process it contains. Also, we sometimes use $X(\boldsymbol{v})$ to refer to the specification $X = (\{X(\boldsymbol{g}:\boldsymbol{G}) = \dots\}, X(\boldsymbol{v}))$.

4.2. Operational semantics

Because of the immediate recursive call after each action, each state of an LPPE corresponds to a valuation of its global variables. Therefore, every reachable state in the underlying PA can be identified uniquely with one of the vectors $\boldsymbol{g'} \in \boldsymbol{G}$ (with the initial vector identifying the initial state). From the SOS rules it follows that for all $\boldsymbol{g'} \in \boldsymbol{G}$, there is a transition $\boldsymbol{g'} \xrightarrow{a(\boldsymbol{q})} \mu$ if and only if for at least one summand i there is a choice of local variables $\boldsymbol{d'_i} \in \boldsymbol{D_i}$ such that

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

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

Example 13. *Consider the following system, continuously sending a random element of a finite type D:*

$$X = \mathit{choose} \sum_{x:D} \tfrac{1}{|D|} : \mathit{send}(x) \cdot X$$

Now consider the following LPPE, where $d' \in D$ was chosen arbitrarily. It is easy to see that X is isomorphic to $Y(1, d')$. (Note that d' could be chosen arbitrarily as it is overwritten before used.)

$$\begin{aligned} Y(pc:\{1,2\}, x:D) = \; & pc = 1 \Rightarrow \mathit{choose} \textstyle\sum_{d:D} \frac{1}{|D|} : Y(2,d) \\ + \; & pc = 2 \Rightarrow \mathit{send}(x) \textstyle\sum_{y:\{*\}} 1 : Y(1,d') \end{aligned}$$

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

5. Linearisation

The process of transforming a prCRL specification to the LPPE format is called *linearisation*. As all our reductions will be defined for LPPEs, linearisation makes them applicable to every prCRL model. Moreover, state space generation is implemented more easily for the LPPE format, and parallel composition can be defined elegantly (as we will see in Section 6).

Linearisation of a prCRL specification P is performed in two steps. In the first step, a specification P' is created, such that $P' \approx P$ and P' is in so-called *intermediate regular form* (IRF). Basically, this form requires every right-hand side to be a summation of process terms, each of which contains exactly one action. This step is performed by Algorithm 1 (page 9), which uses Algorithms 2 and 3 (page 10 and page 11). In the second step, an LPPE X is created, such that $X \equiv P'$. This step is performed by Algorithm 4 (page 13).

We first illustrate both steps by two examples.

Example 14. *Consider the specification $P = (\{X = a \cdot b \cdot c \cdot X\}, X)$. The behaviour of P does not change if we introduce a new process $Y = b \cdot c \cdot X$ and let X instantiate Y after its action a. Splitting the new process as well, we obtain the strongly bisimilar (in this case even isomorphic) specification $P' = (\{X = a \cdot Y, Y = b \cdot Z, Z = c \cdot X\}, X)$. Clearly, this specification is in IRF. Now, an isomorphic LPPE is constructed by introducing a program counter pc that keeps track of the subprocess that is currently active, as shown below. It is easy to see that $P''(1) \equiv P$.*

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

Example 15. *Now consider the following specification, consisting of two processes with parameters. Let $X(d')$ be the initial process for some arbitrary $d' \in D$. (The types D and E are assumed to be finite and to have addition defined on them).*

$$\begin{aligned} X(d : D) = {} & choose \textstyle\sum_{e:E} \frac{1}{|E|} : send(d+e) \textstyle\sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.9 \text{ else } 0.1) : ((i = 1 \Rightarrow Y(d+1)) + \\ & (i = 2 \Rightarrow crash \textstyle\sum_{j:\{*\}} 1 : X(d))) \\ Y(f : D) = {} & write(f) \textstyle\sum_{k:\{*\}} 1 : \sum_{g:D} write(f+g) \textstyle\sum_{l:\{*\}} 1 : X(f+g) \end{aligned}$$

Again, we introduce a new process for each subprocess. The new initial process is $X_1(d', f', e', i')$, where f', e', and i' can be chosen arbitrarily (and d' should correspond to the original initial value d').

$$\begin{aligned} X_1(d : D, f : D, e : E, i : \{1,2\}) = {} & choose \textstyle\sum_{e:E} \frac{1}{|E|} : X_2(d, f', e, i') \\ X_2(d : D, f : D, e : E, i : \{1,2\}) = {} & send(d+e) \textstyle\sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.9 \text{ else } 0.1) : X_3(d, f', e', i) \\ X_3(d : D, f : D, e : E, i : \{1,2\}) = {} & (i = 1 \Rightarrow write(d+1) \textstyle\sum_{k:\{*\}} 1 : X_4(d', d+1, e', i')) \\ & + (i = 2 \Rightarrow crash \textstyle\sum_{j:\{*\}} 1 : X_1(d, f', e', i')) \\ X_4(d : D, f : D, e : E, i : \{1,2\}) = {} & \textstyle\sum_{g:D} write(f+g) \textstyle\sum_{l:\{*\}} 1 : X_1(f+g, f', e', i') \end{aligned}$$

Note that we added process variables to store the values of local variables that were bound by a nondeterministic or probabilistic summation. As the index variables j, k and l are never used, and g is only used directly after the summation that binds it, they are not stored. We reset variables that are not syntactically used in their scope to keep the state space small.

Again, the LPPE is obtained by introducing a program counter. Its initial vector is $(1, d', f', e', i')$.

$$\begin{aligned} X(pc : \{1,2,3,4\}, d : D, f : D, e : E, i : \{1,2\}) = & \\ pc = 1 & \Rightarrow choose \textstyle\sum_{e:E} \frac{1}{|E|} : X(2, d, f', e, i') \\ + \quad pc = 2 & \Rightarrow send(d+e) \textstyle\sum_{i:\{1,2\}} (\text{if } i = 1 \text{ then } 0.9 \text{ else } 0.1) : X(3, d, f', e', i) \end{aligned}$$

$$
\begin{aligned}
&+ \quad pc = 3 \wedge i = 1 \Rightarrow write(d+1) \textstyle\sum_{k:\{*\}} 1 : X(4, d', d+1, e', i') \\
&+ \quad pc = 3 \wedge i = 2 \Rightarrow crash \textstyle\sum_{j:\{*\}} 1 : X(1, d, f', e', i') \\
&+ \textstyle\sum_{g:D} pc = 4 \quad \Rightarrow write(f+g) \textstyle\sum_{l:\{*\}} 1 : X(1, f+g, f', e', i')
\end{aligned}
$$

5.1. Transforming a specification to intermediate regular form

We now formally define the intermediate regular form (IRF), and then discuss the transformation from prCRL to IRF in more detail.

Definition 16. *A process term is in* IRF *if it adheres to the following grammar:*

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

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 process variables.

Note that in IRF every probabilistic sum goes to a process instantiation, and that process instantiations do not occur in any other way. Therefore, every process instantiation is preceded by exactly one action.

For every specification P there exists a specification P' in IRF such that $P \approx P'$ (since we provide an algorithm to construct it). However, it is not hard to see that P' is not unique.

Remark 17. It is not necessarily true that $P \equiv P'$, as we will show in Example 20. Still, every specification P representing a finite PA *can* be transformed to an IRF describing an isomorphic PA: define a data

---

Algorithm 1: Transforming a specification to IRF

Input:

- A prCRL specification $P = (\{X_1(\boldsymbol{x_1} : \boldsymbol{D_1}) = p_1, \ldots, X_n(\boldsymbol{x_n} : \boldsymbol{D_n}) = p_n\}, X_1(\boldsymbol{v}))$, in which all variables (either declared as a process variable, or bound by a nondeterministic or probabilistic sum) are named uniquely.

Output:

- A prCRL specification $P' = (\{X'_1(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_1, \ldots, X'_k(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_k\}, X'_1(\boldsymbol{v'}))$ such that P' is in IRF and $P' \approx P$.

Initialisation

1 $[(y_1, E_1), \ldots, (y_m, E_m)] = [(y, E) \mid \exists i \,.\, p_i$ binds a variable y of type E by a nondeterministic or probabilistic sum, and syntactically uses y within its scope$]$

2 pars $:= (\boldsymbol{x_1} : \boldsymbol{D_1}, (\boldsymbol{x_2}, \boldsymbol{x_3}, \ldots, \boldsymbol{x_n}, y_1, \ldots, y_m) : (\boldsymbol{D_2} \times \boldsymbol{D_3} \times \cdots \times \boldsymbol{D_n} \times E_1 \times \cdots \times E_m))$

3 $\boldsymbol{v'} := \boldsymbol{v}$ ++ [any constant of type $D \mid D \leftarrow [\boldsymbol{D_2}, \boldsymbol{D_3}, \ldots, \boldsymbol{D_n}, E_1, \ldots, E_m]]$

4 done $:= \emptyset$

5 toTransform $:= \{X'_1(\text{pars}) = p_1\}$

6 bindings $:= \{X'_1(\text{pars}) = p_1\}$

Construction

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

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

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

10 done $:=$ done $\cup \{X'_i(\text{pars}) = p'_i\}$

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

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

13 **return** $(\text{done}, X'_1(\boldsymbol{v'}))$

Algorithm 2: Transforming process terms to IRF

Input:

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

Output:

- The IRF for p.
- The process equations to add to *toTransform*.

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

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

type S with an element s_i for every reachable state of the PA underlying P, and create a process $X(s : S)$ consisting of a summation of terms of the form

$$s = s_i \Rightarrow a(\boldsymbol{t})(p_1 : s_1 \oplus p_2 : s_2 \oplus \ldots \oplus p_n : s_n)$$

(one for each transition $s_i \xrightarrow{a(\boldsymbol{t})} \mu$, where $\mu(s_1) = p_1, \mu(s_2) = p_2, \ldots, \mu(s_n) = p_n$). However, this transformation completely defeats its purpose, as the whole idea behind the LPPE is to apply reductions *before* having to compute all states of the original specification.

Overview of the transformation to IRF. Algorithm 1 transforms a specification P to a specification P', in such a way that $P \approx P'$ and P' is in IRF. It requires that all process variables and local variables of P have unique names (which is easily achieved by renaming variables having names that are used more than once). 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; (3) *bindings* is a set

Algorithm 3: Normalising process terms

Input:

- A process term p.
- A list *pars* of typed global variables.
- A prCRL specification P.
- A new initial vector $\boldsymbol{v}' = (v'_1, v'_2, \ldots, v'_k)$.

Output:

- The normal form of p.
- The actual parameters needed to supply to a process which has right-hand side p' to make its behaviour strongly probabilistically bisimilar to p.

---

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

1 **case** $p = Y(t_1, t_2, \ldots, t_n)$

2 **return**$(\text{RHS}(Y), [\text{inst}(v) \mid (v, D) \leftarrow \text{pars}])$

where $\text{inst}(v) = \begin{cases} t_i & \text{if } v \text{ is the } i^{\text{th}} \text{ global variable of } Y \text{ in } P \\ v'_i & \text{if } v \text{ is not a global variable of } Y \text{ in } P, \text{ and } v \text{ is the } i^{\text{th}} \text{ element of pars} \end{cases}$

3 **case** otherwise

4 **return** $(p, [\text{inst}'(v) \mid (v, D) \leftarrow \text{pars}])$

where $\text{inst}'(v) = \begin{cases} v & \text{if } v \text{ occurs syntactically in } p \\ v'_i & \text{if } v \text{ does not occur syntactically in } p, \text{ and } v \text{ is the } i^{\text{th}} \text{ element of pars} \end{cases}$

---

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

Initially, *pars* is assigned the vector of all variables declared in P, either globally or in a summation (and syntactically used after being bound), together with the corresponding type. The new initial vector $\boldsymbol{v}'$ is constructed by appending dummy values to the original initial vector for all added variables (denoted by Haskell-like list comprehension). Also, *done* is empty, the right-hand side of the initial process is bound to $X'_1(\mathit{pars})$, and this equation is added to *toTransform*. Then, we repeatedly take an equation $X'_i(\mathit{pars}) = p_i$ from *toTransform*, transform p_i to a strongly probabilistically bisimilar IRF p'_i using Algorithm 2, add the equation $X'_i(\mathit{pars}) = p'_i$ to *done*, and remove $X'_i(\mathit{pars}) = p_i$ from *toTransform*. The transformation may introduce new processes, which are added to *toTransform*, and *bindings* is updated accordingly.

Transforming single process terms to IRF. Algorithm 2 transforms individual process terms to IRF recursively by means of a case distinction over the structure of the terms (using Algorithm 3).

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 $\sum_{\boldsymbol{x}:\boldsymbol{D}} q_1$ it is $\sum_{\boldsymbol{x}:\boldsymbol{D}} q'_1$. 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 $\boldsymbol{t}$.

The base case is a probabilistic choice $a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : q$. The corresponding process term in IRF depends on whether or not there already is a process name X'_j mapped to q (as stored in *bindings*). If this is the case, apparently q has been linearised before and the result simply is $a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : X'_j(\mathit{actualPars})$, with *actualPars* as explained below. If q was not linearised before, a new process name X'_k is chosen, the result is $a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : X'_k(\mathit{actualPars})$ and X'_k is mapped to q by adding this information to *bindings*. Since a newly created process X'_k is added to *toTransform*, in a next iteration of Algorithm 1 it will be linearised.

More precisely, instead of q we use its *normal form*, computed by Algorithm 3. The reason behind this is that, when linearising a process in which for instance both the process instantiations $X(n)$ and $X(n+1)$ occur, we do not want to have a distinct term for both of them. We therefore define the normal form of a process instantiation $Y(\boldsymbol{t})$ to be the right-hand side of Y, and of any other process term q to just be q. This

Table 2: Transforming $P_1 = (\{X_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2, X_2 = a \cdot b \cdot c \cdot X_1\}, X_1)$ to IRF.

	$done_1$	$toTransform_1$	$bindings_1$
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 3: Transforming $P_2 = (\{X_3(d : D) = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)\}, X_3(d'))$ to IRF.

	$done_2$	$toTransform_2$	$bindings_2$
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	$X_2'' = c(e) \cdot X_1''(5, e')$	$\emptyset$	

way, different process instantiations of the same process and the right-hand side of that process all have the same normal form, and no duplicate terms are generated.

Algorithm 3 is also used to determine the actual parameters that have to be provided to either X_j' (if q was already linearised before) or to X_k' (if q was not linearised before). This depends on whether or not q is a process instantiation. If it is not, the actual parameters for X_j' are just the global variables (possibly resetting variables that are not used in q). If it is, for instance $q = Y(t_1, t_2, \ldots, t_n)$, all global variables are reset, except the ones corresponding to the original global variables of Y; for them $t_1, t_2, \ldots, t_n$ are used.

Note that in Algorithm 3 we use $(v, D) \leftarrow pars$ to denote the list of all pairs (v_i, D_i), given $pars = (v_1, \ldots, v_n) : (D_1 \times \cdots \times D_n)$. We use $\mathrm{RHS}(Y)$ for the right-hand side of the process equation defining Y.

Example 18. *We linearise two example specifications:*

$$P_1 = (\{X_1 = a \cdot b \cdot c \cdot X_1 + c \cdot X_2, X_2 = a \cdot b \cdot c \cdot X_1\}, X_1)$$
$$P_2 = (\{X_3(d : D) = \sum_{e:D} a(d + e) \cdot c(e) \cdot X_3(5)\}, X_3(d'))$$

Tables 2 and 3 show done, toTransform *and* bindings *at line 7 of Algorithm 1 for every iteration. As* 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 3. The results in IRF are* $P_1' = (done_1, X_1')$ *and* $P_2' = (done_2, X_1''(d', e'))$ *for an arbitrary* $e' \in D$.

The following theorem, proven in Appendix A, states the correctness of our transformation.

Theorem 19. *Let P be a prCRL specification such that all variables are named uniquely. Given this input, Algorithm 1 terminates, and the specification P' it returns is such that $P' \approx P$. Also, P' is in IRF.*

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

Example 20. *Let* $P = (\{X = \sum_{d:D} a(d) \cdot b(f(d)) \cdot X\}, X)$, *with* $f(d) = 0$ *for all* $d \in D$. *Then, our procedure will yield the specification*

$$P' = (\{X_1'(d : D) = \sum_{d:D} a(d) \cdot X_2'(d), X_2'(d : D) = b(f(d)) \cdot X_1'(d')\}, X_1'(d'))$$

for some $d' \in D$. *Note that the reachable number of states of* P' *is* $|D| + 1$ *for any* $d' \in D$. *However, the reachable state space of P only consists of the two states X and* $b(0) \cdot X$.

Algorithm 4: Constructing an LPPE from an IRF

Input:

- A specification $P' = (\{X'_1(\boldsymbol{x} : \boldsymbol{D}) = p'_1, \ldots, X'_k(\boldsymbol{x} : \boldsymbol{D}) = p'_k\}, X'_1(\boldsymbol{v}))$ in IRF (without variable pc).

Output:

- A semi-LPPE $X = (\{X(pc : \{1, \ldots, k\}, \boldsymbol{x} : \boldsymbol{D}) = p''\}, X(1, \boldsymbol{v}))$ such that $P' \equiv X$.

Construction

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

where
makeSummands(p, i) =
5 **case** $p = a(\boldsymbol{t}) \sum_{\boldsymbol{y}:\boldsymbol{E}} f : X'_j(t'_1, \ldots, t'_k)$
6 **return** $\{pc = i \Rightarrow a(\boldsymbol{t}) \sum_{\boldsymbol{y}:\boldsymbol{E}} f : X(j, t'_1, \ldots, t'_k)\}$
7 **case** $p = c \Rightarrow q$
8 **return** $\{c \Rightarrow q' \mid q' \in \text{makeSummands}(q, i)\}$
9 **case** $p = q_1 + q_2$
10 **return** makeSummands$(q_1, i) \cup$ makeSummands(q_2, i)
11 **case** $p = \sum_{\boldsymbol{x}:\boldsymbol{D}} q$
12 **return** $\{\sum_{\boldsymbol{x}:\boldsymbol{D}} q' \mid q' \in \text{makeSummands}(q, i)\}$

5.2. Transforming from IRF to LPPE

Given a specification P' in IRF, Algorithm 4 constructs an LPPE X. The global variables of X are a program counter pc and all global variables of P'. To construct the summands for X, we range over the process equations in P'. For each equation $X'_i(\boldsymbol{x} : \boldsymbol{D}) = a(\boldsymbol{t}) \sum_{\boldsymbol{y}:\boldsymbol{E}} f : X'_j(t'_1, \ldots, t'_k)$, a summand $pc = i \Rightarrow a(\boldsymbol{t}) \sum_{\boldsymbol{y}:\boldsymbol{E}} f : X(j, t'_1, \ldots, t'_k)$ is constructed. For an equation $X'_i(\boldsymbol{x} : \boldsymbol{D}) = q_1 + q_2$ the union of the summands produced by $X'_i(\boldsymbol{x} : \boldsymbol{D}) = q_1$ and $X'_i(\boldsymbol{x} : \boldsymbol{D}) = q_2$ is taken. For $X'_i(\boldsymbol{x} : \boldsymbol{D}) = c \Rightarrow q$ the condition c is prefixed to the summands produced by $X'_i(\boldsymbol{x} : \boldsymbol{D}) = q$; nondeterministic sums are handled similarly.

To be precise, the specification produced by the algorithm is not literally an LPPE yet, as there might be several conditions and nondeterministic sums, and their order might still be wrong (we call such specifications semi-LPPEs). An isomorphic LPPE is obtained by moving the nondeterministic sums to the front and merging separate nondeterministic sums (using vectors) and separate conditions (using conjunctions). When moving nondeterministic sums to the front, some variable renaming might need to be done to avoid clashes with the conditions.

Example 21. *Looking at the IRFs obtained in Example 18, it follows that $P'_1 \equiv X$ and $P'_2 \equiv Y$, with*

$$\begin{aligned} X = (\{&X(pc : \{1,2,3,4\}) \\ &= pc = 1 \Rightarrow a \cdot X(2) \\ &+ pc = 1 \Rightarrow c \cdot X(3) \\ &+ pc = 2 \Rightarrow b \cdot X(4) \\ &+ pc = 3 \Rightarrow a \cdot X(2) \\ &+ pc = 4 \Rightarrow c \cdot X(1)\}, \\ &X(1)) \end{aligned}$$

$$\begin{aligned} Y = (\{&Y(pc : \{1,2\}, d : D, e : D) \\ &= \sum_{e:D} pc = 1 \Rightarrow a(d+e) \cdot Y(2, d', e) \\ &+ \quad pc = 2 \Rightarrow c(e) \cdot Y(1, 5, e'))\}, \\ &Y(1, d', e')) \end{aligned}$$

Theorem 22. *Let P' be a specification in IRF without a variable pc, and let the output of Algorithm 4 applied to P' be the specification X. Then, $P' \equiv X$.*

Let Y be like X, except that for each summand all nondeterministic sums have been moved to the beginning while substituting their variables by fresh names, and all separate nondeterministic sums and separate conditions have been merged (using vectors and conjunctions, respectively). Then, Y is an LPPE and $Y \equiv X$.

Proof. Algorithm 4 transforms a specification $P' = (\{X'_1(\boldsymbol{x} : \boldsymbol{D}) = p'_1, \ldots, X'_k(\boldsymbol{x} : \boldsymbol{D}) = p'_k\}, X'_1(\boldsymbol{v}))$ to an LPPE $X = (\{X(pc : \{1, \ldots, k\}, \boldsymbol{x} : \boldsymbol{D})\}, X(1, \boldsymbol{v}))$ 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 $X(j, \boldsymbol{v})$.

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 $\boldsymbol{u} \in \boldsymbol{D}$. (As (1) isomorphism is defined over the disjoint union of the two systems, and (2) each PA contains not just the reachable states but all process terms, formally we should also define (1) $h(X(i, \boldsymbol{u})) = X'_i(\boldsymbol{u})$, and (2) $h(p) = p$ for all process terms of a different form. However, this does not influence the proof, as (1) we prove an equivalence, and (2) isomorphism is reflexive). Note that h indeed 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 $\boldsymbol{u} \in \boldsymbol{D}$, we assume an arbitrary $X'_l(\boldsymbol{u})$ and use induction over its structure.

The base case is $X'_l(\boldsymbol{u}) = a(\boldsymbol{t})\sum_{\boldsymbol{y}:\boldsymbol{E}} f : X'_j(t'_1, \ldots, t'_k)$. For this process, Algorithm 4 constructs the summand $pc = l \Rightarrow a(\boldsymbol{t})\sum_{\boldsymbol{y}:\boldsymbol{E}} f : X(j, t'_1, \ldots, t'_k)$. 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, $X''_l(\boldsymbol{u}) = q$ 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, $X'_l(\boldsymbol{u}) \xrightarrow{\alpha} \mu$ 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 $X'_l(\boldsymbol{u}) = \sum_{\boldsymbol{x}:\boldsymbol{D}} q$. Hence, $P' \equiv X$.

Now, let Y be equal to X, except that within each summand all nondeterministic sums have been moved to the beginning while substituting their variables by fresh names, and all separate nondeterministic sums and separate conditions have been merged (using vectors and conjunctions, respectively).

To see that Y is an LPPE, first observe that X already was 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. Furthermore, each probabilistic sum is indeed followed by a process instantiation, as can be seen from line 6 of Algorithm 4.

The only discrepancy for X to be an LPPE is that the nondeterministic sums and the conditions are not yet necessarily in the right order, and there might be several of them (e.g., $d > 0 \Rightarrow \sum_{d:D_1} e > 0 \Rightarrow \sum_{f:D_2} act(d).X(d, f)$). However, since they are swapped in Y such that all nondeterministic sums precede all conditions, and conditions are merged using conjunctions and summations are merged using vectors, Y is an LPPE. In case of the example before: $\sum_{(d',f):D_1 \times D_2} d > 0 \wedge e > 0 \Rightarrow act(d').X(d', f)$. Note that, indeed, some renaming had to be done such that $Y \equiv X$. After all, by renaming the summation variables, we can rely on the obvious equality of $c \Rightarrow \sum_{d:D} p$ and $\sum_{d:D} c \Rightarrow p$ given that d is not a free variable in c. Thus, swapping the conditions and nondeterministic sums this way does not modify the process' semantics in any way. Also, it is easy to see from the SOS rules in Table 1 that merging nondeterministic sums and conditions does not change anything about the enabled transitions. Therefore, Y is indeed isomorphic to X. □

To discuss the complexity of linearisation, we first define the size of prCRL specifications.

Definition 23. *The size of a process term is defined as follows:*

$$size(c \Rightarrow p) = 1 + size(c) + size(p) \qquad size(Y(\boldsymbol{t})) = 1 + size(\boldsymbol{t});$$

$$size(\textstyle\sum_{\boldsymbol{x}:\boldsymbol{D}} p) = 1 + |\boldsymbol{x}| + size(p) \qquad size\big(a(\boldsymbol{t})\textstyle\sum_{\boldsymbol{x}:\boldsymbol{D}} f : p\big) = 1 + size(\boldsymbol{t}) + |\boldsymbol{x}| + size(f) + size(p)$$

$$size(p + q) = 1 + size(p) + size(q); \qquad size((t_1, t_2, \ldots, t_n)) = size(t_1) + size(t_2) + \cdots + size(t_n)$$

The size of the expressions f, c and t_i are given by their number of function symbols and constants. Also, $size(X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = p_i) = |\boldsymbol{x_i}| + size(p_i)$. Given a specification $P = (E, I)$, $size(P) = \sum_{p \in E} size(p) + size(I)$.

Proposition 24. *Let P be a prCRL specification such that $size(P) = n$. Then, the worst-case time complexity of linearising P is in $O(n^3)$. The size of the resulting LPPE is worst-case in $O(n^2)$.*

Proof. Let $P = (E, I)$ be a specification such that $size(P) = n$. First of all, note that

$$\left|pars\right| \leq \sum_{(X_i(\boldsymbol{x_i}:\boldsymbol{D_i})=p_i)\in E} |\boldsymbol{x_i}| + \left|\text{subterms}'(P)\right| \leq n \tag{1}$$

after the initialisation of Algorithm 1, where $|pars|$ denotes the numbers of new global variables and $subterms'(P)$ denotes the *multiset* containing all subterms of P (counting a process term that occurs twice as two subterms, and including nondeterministic and probabilistic choices over a vector of k variables k times). When mentioning the subterms of P in this proof, we will be referring to this multiset (for a formal definition of subterms, see Definition 35 in the appendix).

The first inequality follows from the fact that *pars* is defined to be the sequence of all $\boldsymbol{x_i}$ appended by all local variables of P (that are syntactically used), and the observation that there are at most as many local variables as there are subterms. The second inequality follows from the definition of *size* and the observation that $size(p_i)$ counts the number of subterms of p plus the size of their expressions.

Time complexity. We first determine the worst-case time complexity of Algorithm 1. As the function *transform* is called at most once for every subterm of P, it follows from Equation (1) that the number of times this happens is in $O(n)$. The 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 (1) 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 $O(n^2)$. Therefore, the worst-case time complexity of Algorithm 1 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 $O(n^3)$.

LPPE size complexity. 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 a more precise time complexity, we can define $m = |\text{subterms}'(P)|$ and

$$k = |\text{subterms}'(P)| + \sum_{(X_i(\boldsymbol{x_i}:\boldsymbol{D_i})=p_i)\in E} |\boldsymbol{x_i}|.$$

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

Although the transformation to LPPE increases the size of the specification, it facilitates optimisations to reduce the state space (which is worst-case exponential), as we will see in Section 7.

6. Parallel composition

Using prCRL processes as basic building blocks, we support the modular construction of large systems via top-level parallelism, encapsulation, hiding, and renaming.

Definition 25. *A* process term *in parallel prCRL is any term that can be generated by 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 term, $E, H \subseteq$ Act are sets of actions, and R: Act $\to$ Act maps actions to actions. A parallel prCRL specification $P = (\{X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = q_i\}, X_j(\boldsymbol{t}))$ *is a set of* parallel prCRL process equations *(which are like prCRL process equations, but with parallel prCRL process terms as right-hand sides) together with an initial process. The wellformedness criteria of Definition 5 are lifted in the obvious way.*

Table 4: SOS rules for parallel prCRL.

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

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

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

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

In a parallel prCRL process term, $q_1 \,||\, q_2$ is parallel composition. Furthermore, $\partial_E(q)$ encapsulates the actions in E, $\tau_H(q)$ hides the actions in H (renaming them to τ and removing their parameters), and $\rho_R(q)$ renames actions using R. Parallel processes by default interleave all their actions. However, we assume a partial function $\gamma\colon \text{Act} \times \text{Act} \to \text{Act}$ that specifies which actions can communicate; more precisely, $\gamma(a,b) = c$ denotes that a and b can communicate if their parameters are equal, resulting in the action c with these parameters (as in ACP [34]). The SOS rules for parallel prCRL are shown in Table 4 (relying on the SOS rules for prCRL from Table 1), where 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 $\partial_E(\mu)$. Note that there is no Encap-T rule, to remove transitions labelled by an encapsulated action.

6.1. 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). Assume the following two LPPEs (omitting the initial states, since they are not needed in the process of parallelisation).

$$X(\boldsymbol{g} : \boldsymbol{G}) = \sum_{i \in I} \sum_{\boldsymbol{d_i} : \boldsymbol{D_i}} c_i \Rightarrow a_i(\boldsymbol{b_i}) \sum_{\boldsymbol{e_i} : \boldsymbol{E_i}} f_i : X(\boldsymbol{n_i}),$$
$$Y(\boldsymbol{g'} : \boldsymbol{G'}) = \sum_{i \in I'} \sum_{\boldsymbol{d'_i} : \boldsymbol{D'_i}} c'_i \Rightarrow a'_i(\boldsymbol{b'_i}) \sum_{\boldsymbol{e'_i} : \boldsymbol{E'_i}} f'_i : Y(\boldsymbol{n'_i}),$$

Also assuming (without loss of generality) that all global and local variables are named uniquely, 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 introduced by Usenko for traditional LPEs [18]. Note that the first set of summands represents X doing a transition independent from Y, and that the second set of summands represents Y doing a transition independent from X. The third set corresponds to their communications.

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

Here, $I\gamma I'$ 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, $I\gamma I' = \{(k,l) \in I \times I' \mid (a_k, a'_l) \in \mathrm{domain}(\gamma)\}$.

Proposition 26. *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 become, are of the form $Z(\hat{\boldsymbol{v}},\hat{\boldsymbol{v}}')$, and the only processes a parallel composition $X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}')$ can become, are of the form $X(\hat{\boldsymbol{v}}) \,||\, Y(\hat{\boldsymbol{v}}')$. Therefore, the isomorphism h needed to prove the proposition is as follows: $h(X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}')) = Z(\boldsymbol{v},\boldsymbol{v}')$ and $h(Z(\boldsymbol{v},\boldsymbol{v}')) = X(\boldsymbol{v}) \,||\, Y(\boldsymbol{v}')$ for all $\boldsymbol{v} \in \boldsymbol{G}, \boldsymbol{v}' \in \boldsymbol{G}'$, and $h(p) = p$ for every process term p of a different form. 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) $X(\boldsymbol{v}) \xrightarrow{a'(\boldsymbol{q})} \mu' \wedge Y(\boldsymbol{v}') \xrightarrow{a''(\boldsymbol{q})} \mu'' \wedge \gamma(a',a'') = a \wedge \forall \hat{\boldsymbol{v}} \in \boldsymbol{G}, \hat{\boldsymbol{v}}' \in \boldsymbol{G}' \,.\, \mu(X(\hat{\boldsymbol{v}}) \,||\, Y(\hat{\boldsymbol{v}}')) = \mu'(\hat{\boldsymbol{v}}) \cdot \mu''(\hat{\boldsymbol{v}}')$

It immediately follows from the construction of Z 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. □

6.2. Linearisation of hiding, encapsulation and renaming

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

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

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

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

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

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

where

$$a'_i = \begin{cases} \tau & \text{if } a_i \in H \\ a_i & \text{otherwise} \end{cases} \qquad \boldsymbol{b'_i} = \begin{cases} () & \text{if } a_i \in H \\ \boldsymbol{b_i} & \text{otherwise} \end{cases} \qquad a''_i = R(a_i) \qquad I' = \{i \in I \mid a_i \notin E\}$$

Proposition 27. *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. We will prove that $U(\boldsymbol{v}) \equiv \tau_H(X(\boldsymbol{v}))$ for all $\boldsymbol{v} \in \boldsymbol{G}$; the other two statements are proven similarly.

The only processes an LPPE $X(\boldsymbol{v})$ can become, 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 become, are processes of the form $\tau_H(X(\boldsymbol{v}'))$. Therefore, the isomorphism h needed to prove the proposition is easy: for all $\boldsymbol{v} \in \boldsymbol{G}$, we define $h(\tau_H(X(\boldsymbol{v}))) = U(\boldsymbol{v})$ and $h(U(\boldsymbol{v})) = \tau_H(X(\boldsymbol{v}))$, and $h(p) = p$ for every p of a different form. Clearly, h is bijective.

We will now show that h indeed is an isomorphism. First, observe that $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu$ is enabled if and only if there is a summand $i \in I$ such that

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

We show that h is an isomorphism, by showing that there exists a transition $\tau_H(X(\boldsymbol{v})) \xrightarrow{a(\boldsymbol{q})} \mu$ 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 $a \neq \tau$. 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 corresponds to $X(\boldsymbol{v}) \xrightarrow{a(\boldsymbol{q})} \mu \wedge a \notin H$ by 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 by definition of a_i' and $\boldsymbol{b_i'}$ that $U(\boldsymbol{v}) \xrightarrow{\tau} \mu_h$ is enabled under exactly these conditions. □

7. The LPPE-based verification approach: linearisation, reduction, instantiation

The LPPE format allows us to verify systems efficiently. The main idea, as in the μCRL approach [1], is to start with a prCRL specification, linearise to an LPPE, and then generate (instantiate) the state space.

Using the LPPE, several existing reduction techniques can now also be applied to probabilistic specifications. We distinguish between two kinds of reductions: (1) *LPPE simplification techniques*, which do not change the actual state space, but improve readability and speed up state space generation, and (2) *state space reduction techniques* that do change the LPPE or instantiation in such a way that the resulting state space will be smaller (while preserving some notion of equivalence, often strong or branching probabilistic bisimulation). Figure 2 illustrates this approach. We list all the reductions techniques that we generalised (and indicate their category within parentheses). All these techniques work on the syntactic level, i.e., they do not unfold the data types at all, or only locally to avoid a data explosion. Hence, a smaller state space is obtained without first having to generate the original one. Here, we discuss the reduction techniques.

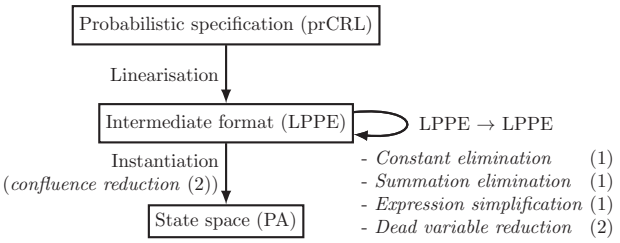

Figure 2: The LPPE-based verification approach.

7.1. LPPE simplification techniques

The simplification techniques that already existed for the LPE format [22, 18] can be generalised in a straight-forward way. Here, we discuss constant and summation elimination, and expression simplification.

Constant elimination. In case a parameter of an LPPE never changes its value, we can clearly just omit it and replace every reference to it by its initial value. Basically, we detect a parameter p to be constant if in every summand it is either unchanged, or 'changed' to its initial value.

More precisely, we do a greatest fixed-point computation to find all non-constants, initially assuming all parameters to be constant. If no new non-constants are found (which happens after a finite number of iterations as there are finitely many parameters), the procedure terminates and the remaining parameters are constant. In every iteration, we check for each parameter x that is still assumed constant whether there exists a summand s (with an enabling condition that cannot be shown to be always unsatisfied) that might change it. This is the case if either x is bound by a probabilistic or nondeterministic summation in s, or if its next state is determined by an expression that is syntactically different from x, different from the initial value of x, and different from the name of another parameter that is still assumed constant and has the same initial value as x.

Example 28. *Consider the specification* $P = (\{X(id : \{one, two\}) = say(id) \cdot X(id)\}, X(one))$. *Clearly, the process variable id never changes value, so the specification can be simplified to* $P' = (\{X = say(one) \cdot X\}, X)$.

Proposition 29. *The underlying PAs of an LPPE before and after constant elimination are isomorphic.*

Proof (sketch). We first show that every parameter that is marked constant by the procedure, indeed has the same value in every reachable state of the state space. Let x be a parameter with initial value x^0 that is not constant, so there exists a state $\boldsymbol{v}$ where x is not equal to x^0. Then there must be a summand that semantically changes x to a value different from x^0. If the next state of x in this summand is syntactically given by the expression x, and x is changed because it is bound to a value $x' \neq x^0$ by a nondeterministic or probabilistic sum, this is detected by the procedure in the first iteration. If the next state is given by an expression e not equal to x, then e must also be different from x^0 (otherwise the value of x would still not change). In the general case, this will also be detected in the first iteration, except when e is the name of another parameter y that is initially, but not constantly, equal to x^0. Either the first iteration detects y to be non-constant and the second iteration will then see that x is also non-constant, or y is also non-constant because of a valuation based on another parameter. In the latter case, it might take some more iterations, but as y is non-constant this recursion clearly ends at some point (as a cyclicity would imply that x is constant, violating our assumption).

Now, as the procedure correctly computes the parameters of an LPPE that are constant, it is trivial that changing all their occurrences to their initial values is a valid transformation; it does not change the semantics of the LPPE and therefore leaves the reachable state space (including its transitions) untouched. Moreover, as the constant parameters are not used anymore after this step, they have no influence on the semantics of the LPPE, and removing them also does also not change anything (except for state names, which is allowed by isomorphism). □

Summation elimination. When composing parallel components that communicate based on message passing actions, often this results in summations that can be eliminated. For instance, summands with a nondeterministic choice of the form $\sum_{d:D}$ and a condition $d = e$ can arise, which can obviously be simplified by omitting the summation and substituting e for every occurrence of d.

More precisely, to eliminate a sum $\sum_{d:D}$ in a summand that has the enabling condition c, we compute the set S of possible values that c allows for d (and use the empty set if we cannot establish specific values). When c is given by $d = e$ or $e = d$, where e is an expression in which d does not occur freely, we take $S = \{e\}$. When c is a conjunction $e_1 \wedge e_2$, we take $S = S_1 \cap S_2$, where S_i is the set of possible values for c given by e_i. For a disjunction, we take a union (unless $S_1 = \emptyset$ or $S_2 = \emptyset$; in that case also $S = \emptyset$). If it turns out that S is a singleton set $\{d'\}$, then we omit the summation and substitute every free occurrence of d by d'.

Example 30. *Consider the specification* $X = \sum_{d:\{1,2,3\}} d = 2 \Rightarrow \mathit{send}(d) \cdot X$. *As the summand is only enabled for* $d = 2$, *the specification can be simplified to* $X = 2 = 2 \Rightarrow \mathit{send}(2) \cdot X$.

Proposition 31. *The underlying PAs of an LPPE before and after summation elimination are isomorphic.*

Proof (sketch). Assume a summand with a summation $\sum_{d:D}$ and a condition c. Clearly, when c is given by $d = e$ and e does not contain d, the condition can indeed only be satisfied when d is equal to e, so, for any other value the summand would not be enabled. Given a condition $e_1 \wedge e_2$, the summand is only enabled when both e_1 and e_2 hold, so clearly indeed only when d has a value in the intersection of the sets containing the values for those two conditions to hold. For a disjunction $e_1 \vee e_2$, knowing that for e_i to hold d should have a value in S_i, clearly it should have a value in $S_1 \cup S_2$ for the disjunction to hold. However, when either S_1 or S_2 is empty, this implies that we don't know the values that satisfy this disjunct, so then we also don't know anything about the complete disjunction.

If in the end there is precisely one value d' that enables the condition c, this means that the summand can only be taken for this value. Therefore, clearly we can just as well omit the nondeterministic choice and substitute d' for every free occurrence of d in the resulting summand. This obviously changes nothing about the underlying PA, so the transformation preserves isomorphism. □

Expression simplification. Expressions occurring as enabling conditions, action parameters or next state parameters can often be simplified. We apply two kinds of simplifications (recursively): (1) functions for which all parameters are constants are evaluated, and (2) basic laws from logic are applied.

Additionally, summands for which the enabling condition simplified to *false* are removed, as they cannot contribute to the behaviour of an LPPE anyway. More thoroughly, we also check for each summand whether every local and global parameter with a finite type has at least one possible value for which the enabling condition does not simplify to *false*. If there exists a parameter without at least one such a value, the summand can never be taken and is consequently removed.

Example 32. *Consider the expression* $3 = 1 + 2 \vee x > 5$*. As all parameters of the addition function are given, the expression is first simplified to* $3 = 3 \vee x > 5$*. Then, the equality function can be evaluated, obtaining* $true \vee x > 5$*. Finally, logic tells us that we can simplify once more, obtaining the expression true.*

Proposition 33. *The underlying PAs of an LPPE before and after expression simplification are isomorphic.*

Proof. Trivial: replacing expressions by equivalent ones does not change anything. □

7.2. State space reduction techniques

We generalised two state space reduction techniques: dead variable reduction and confluence reduction.

Dead variable reduction. A non-probabilistic version of dead variable reduction was introduced in [23], reducing the state space of an LPE while preserving strong bisimulation. The technique takes into account the control flow of an LPPE, and tries to detect states in which the value of a certain global variable is irrelevant. Basically, this is the case if it will be overwritten before being used for all possible futures. Then, it will be reset to its initial value. Dead variable reduction works best in the presence of large data types, as in that case resets have the most effect.

The generalisation to LPPEs is straight-forward: just also take probabilistic sums and probability expressions into account when investigating when variables are used or overwritten. Trivially, the proofs for the correctness of the technique also generalise to the probabilistic case.

Confluence reduction. The second state space reduction technique we generalised to LPPEs is confluence reduction, introduced in [29] for LPEs to reduce state spaces while preserving branching bisimulation. Basically, this technique detects which internal τ-transitions do not influence a system's behaviour, and uses this information when generating the state space (giving such transitions priority over other transitions). Confluence reduction works best in the presence of a lot of parallelism and hiding, as in that case many unobservable traces can occur in multiple orders.

As the generalisation of confluence reduction is far from straight-forward, the technical details are outside the scope of this paper. We therefore refer to [35, 36] for an exposition of the technique, but will already provide encouraging experimental results in the next section, as evidence for the strength of the LPPE format.

8. Implementation and case study

8.1. Implementation

We developed a prototype tool[3] in Haskell, based on a simple data language that allows the modelling of several kinds of protocols and systems. A web-based interface makes the tool convenient to use. The tool is capable of linearising prCRL specifications, as well as applying parallel composition, hiding, encapsulation and renaming. As Haskell is a functional language, the algorithms in our implementation are almost identical to their mathematical representations presented in this paper.

The tool also implements all the reduction techniques mentioned in Section 7. It automatically applies the basic LPPE simplification techniques, and allows the user to choose whether or not to apply dead variable reduction and/or confluence reduction.

[3]The implementation, including the web-based interface, can be found at `http://fmt.cs.utwente.nl/~timmer/scoop`.

$$P(id : Id, val : Die, set : Bool) = !set \Rightarrow \sum_{d:Die} receive(id, other(id), d) \cdot P(id, d, true)$$
$$+\ set \Rightarrow getVal(val).P(id, val, false)$$

$$A(id : Id) = roll(id) \sum_{d:Die} \tfrac{1}{6} : send(other(id), id, d) \cdot \sum_{e:Die} readVal(e).\big(\quad d = e \Rightarrow A(id)$$
$$+ d > e \Rightarrow leader(id) \cdot A(id)$$
$$+ e > d \Rightarrow follower(id) \cdot A(id)\ \big)$$

$$S = \partial_{send,receive}(\ \partial_{getVal,readVal}(P(one, 1, false) \,||\, A(one))\ \ ||\ \ \partial_{getVal,readVal}(P(two, 1, false) \,||\, A(two))\ )$$

$$\gamma(receive, send) = comm$$
$$\gamma(getVal, readVal) = checkVal$$

Figure 3: A prCRL model of a leader election protocol.

After generating an LPPE, the tool can also generate its state space (a probabilistic automaton) and display it in several ways. We can export to the AUT format for analysis with the CADP toolset [37], or to a PRISM transition matrix for analysis using PRISM [3]. Interestingly, the tool can also translate a (possibly reduced) LPPE directly to a PRISM specification, enabling the user to obtain the exact same state space in PRISM and use this probabilistic model checker to symbolically compute quantitative properties of the model. Note that, in general, not every prCRL specification can be transformed to PRISM due to the richer data types that we allow. In principle, translating PRISM models to LPPE is always possible.

8.2. Case study

To illustrate the process of specifying in prCRL and the potentials of LPPEs, we modelled two leader election protocols (such protocols are a benchmark problem in probabilistic model checking). We applied all our reduction techniques, and found significant reductions in both the state space sizes and the time needed to generate them (Table 5).

Modelling. We modelled a basic variant of the Itai-Rodeh protocol for synchronous leader election between two parties [38], as well an adaption of the Itai-Rodeh protocol for asynchronous rings for any number of parties (Algorithm $\mathcal{B}$ from [39]). As an example we discuss the first specification here; for the other, we refer to the website of our tool.

The protocol we model elects a leader between two nodes by rolling two dice and comparing the results. If 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 3. Here, *Die* is a data type consisting of the numbers from 1 to 6, and *Id* is a data type consisting of the identifiers *one* and *two*. The function *other* provides 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 (after which it is *set*), and then provides the active thread an opportunity to obtain this result (communicating via the *checkVal* action). 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.

After linearising the model with our implementation when disabling all reduction techniques, we obtain an LPPE with 18 parameters and 14 summands (see [40] for a listing). Applying the LPPE simplification techniques, 8 of these parameters are removed by constant elimination, all nondeterministic summations are removed by summation elimination, and 2 summands are removed as their enabling condition simplifies to *false* due to expression simplification. So, we end up with an LPPE with 10 parameters and 12 summands.

Table 5: Applying state space reduction based on LPPEs to two leader election protocols.

Specification	Original States	Original Trans.	Reduced States	Reduced Trans.	Visited States	Visited Trans.	Running time Before	Running time After
`leaderBasic`	3,763	6,158	541	638	1,249	1,370	0.45 sec	0.14 sec
`leader-2-36`	154,473	205,198	5,293	5,330	17,029	18,362	29.53 sec	6.88 sec
`leader-3-12`	468,139	763,149	35,485	41,829	130,905	137,679	136.33 sec	31.59 sec
`leader-3-15`	1,043,635	1,697,247	68,926	80,838	251,226	264,123	313.35 sec	65.96 sec
`leader-3-18`	2,028,181	3,293,493	118,675	138,720	428,940	450,867	1161.58 sec	124.74 sec
`leader-3-21`	out of memory		187,972	219,201	675,225	709,656	–	205.90 sec
`leader-3-24`	out of memory		280,057	326,007	1,001,259	1,052,235	–	328.25 sec
`leader-3-27`	out of memory		398,170	462,864	1,418,220	1,490,349	–	497.94 sec
`leader-4-4`	298,653	626,028	25,617	38,840	127,325	137,964	126.16 sec	31.59 sec
`leader-4-5`	759,952	1,577,516	61,920	92,304	300,569	324,547	322.62 sec	75.14 sec
`leader-4-6`	1,648,975	3,399,456	127,579	188,044	608,799	655,986	1073.16 sec	155.74 sec
`leader-4-7`	out of memory		235,310	344,040	1,108,391	1,192,695	–	291.25 sec
`leader-4-8`	out of memory		400,125	581,468	1,865,627	2,005,676	–	1069.56 sec
`leader-5-2`	260,994	693,960	14,978	29,420	97,006	110,118	155.37 sec	29.40 sec
`leader-5-3`	out of memory		112,559	208,170	694,182	774,459	–	213.10 sec

Experimental results. To investigate the effects of the state space reduction techniques, we generated the state space of the basic synchronous leader election protocol introduced above, as well as of several variants of the asynchronous leader election protocol from [39]. For the second protocol, we considered either 2, 3, 4 or 5 parties, who throw either a normal die or one with more or fewer sides. We use `leader-i-j` to denote the variant with `i` parties throwing a `j`-sided die. The results were obtained on a 2.4 GHz, 2 GB Intel Core 2 Duo MacBook, and can be found in Table 5.

For each variant of the protocol we list the size of the state space before and after reduction, and the number of states explored during the state space generation (as confluence reduction does need to visit some of the states it omits from the state space). Finally, we present the running times of the state space generation with and without the reduction techniques.

A combination of dead variable reduction and confluence reduction reduces the state spaces quite impressively. The number of states decreases between 90% and 95%, and the number of transitions between 94% and 97%. Moreover, the running time needed to generate these reduced state spaces is only one fourth of the time to generate the original state spaces (up until the point where swapping is needed; then, the generation of the smaller state spaces is even faster, relatively).

9. Conclusions and Future Work

This paper introduced a linear process algebraic format, the LPPE, 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 [18] (which is very desirable, as it simplifies the generalisation of existing techniques to the probabilistic setting).

The LPPE format already led to the generalisation of dead variable reduction and confluence reduction to the probabilistic setting, and we demonstrated by a case study remarkable results in reducing both a system's state space and the time needed to generate it.

Interesting directions for future work are the development of additional minimisation techniques, the application of proof techniques such as the cones and foci method to LPPEs, and the investigation of abstraction methods in the context of LPPEs. Also, more case studies could be conducted, to evaluate the effects of our reduction techniques.

Acknowledgements. 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).

We thank Dave Parker from Oxford University for useful discussions about the functionality exporting LPPEs to PRISM models, and Axel Belinfante for implementing a web-based interface for our tool.

References

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

[2] H. Garavel, M. Sighireanu, Towards a second generation of formal description techniques - rationale for the design of E-LOTOS, in: Proc. of the 3rd Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS), CWI, 1998, pp. 198–230.

[3] A. Hinton, M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: Proc. of the 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 3920 of *LNCS*, Springer, 2006, pp. 441–444.

[4] C. Baier, F. Ciesinski, 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), IEEE, 2004, pp. 57–66.

[5] H. C. Bohnenkamp, P. R. D'Argenio, H. Hermanns, J.-P. Katoen, MODEST: A compositional modeling formalism for hard and softly timed systems, IEEE Transactions of Software Engineering 32 (2006) 812–830.

[6] P. R. D'Argenio, B. Jeannet, H. E. Jensen, K. G. 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), volume 2165 of *LNCS*, Springer, 2001, pp. 39–56.

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

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

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

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

[11] M. Kattenbelt, M. Z. Kwiatkowska, G. Norman, D. Parker, A game-based abstraction-refinement framework for Markov decision processes, Formal Methods in System Design 36 (2010) 246–280.

[12] H. Hermanns, B. Wachter, L. Zhang, Probabilistic CEGAR, in: Proc. of the 20th Int. Conf. on Computer Aided Verification (CAV), volume 5123 of *LNCS*, Springer, 2008, pp. 162–175.

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

[14] C. Morgan, A. McIver, pGCL: formal reasoning for random algorithms, South African Computer Journal 22 (1999) 1427.

[15] M. Bezem, J. F. Groote, Invariants in process algebra with data, in: Proc. of the 5th Int. Conf. on Concurrency Theory (CONCUR), volume 836 of *LNCS*, Springer, 1994, pp. 401–416.

[16] K. G. Larsen, A. Skou, Bisimulation through probabilistic testing, Information and Computation 94 (1991) 1–28.

[17] D. Bosscher, 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), volume NS-95-2 of *BRICS Notes Series*, University of Aarhus, 1995, pp. 119–130.

[18] Y. S. Usenko, Linearization in μCRL, Ph.D. thesis, Eindhoven University of Technology, 2002.

[19] P. C. W. van den Brand, M. A. Reniers, P. J. L. Cuijpers, Linearization of hybrid processes, Journal of Logic and Algebraic Programming 68 (2006) 54–104.

[20] J. F. Groote, J. Springintveld, Focus points and convergent process operators: a proof strategy for protocol verification, Journal of Logic and Algebraic Programming 49 (2001) 31–60.

[21] W. Fokkink, J. Pang, J. C. van de Pol, Cones and foci: A mechanical framework for protocol verification, Formal Methods in System Design 29 (2006) 1–31.

[22] J. F. Groote, B. Lisser, Computer Assisted Manipulation of Algebraic Process Specifications, Technical Report SEN-R0117, CWI, 2001.

[23] J. C. van de Pol, 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), volume 5799 of *LNCS*, Springer, 2009, pp. 54–68.

[24] M. V. Espada, J. C. van de Pol, An abstract interpretation toolkit for μCRL, Formal Methods in System Design 30 (2007) 249–273.

[25] S. C. C. Blom, B. Lisser, J. C. van de Pol, M. Weber, A database approach to distributed state-space generation, Journal of Logic and Computation 21 (2011) 45–62.

[26] S. C. C. Blom, J. C. 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), volume 5160 of *LNCS*, Springer, 2008, pp. 81–95.

[27] J. F. Groote, 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), volume 1548 of *LNCS*, Springer, 1998, pp. 74–90.
[28] J. F. Groote, T. A. C. Willemse, Model-checking processes with data, Science of Computer Programming 56 (2005) 251–273.
[29] S. C. C. Blom, J. C. van de Pol, State space reduction by proving confluence, in: Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV), volume 2404 of *LNCS*, Springer, 2002, pp. 596–609.
[30] J.-P. Katoen, J. C. van de Pol, M. I. A. Stoelinga, M. Timmer, A linear process-algebraic format for probabilistic systems with data, in: Proc. of the 10th Int. Conf. on Application of Concurrency to System Design (ACSD), IEEE, 2010, pp. 213–222.
[31] R. Segala, Modeling and Verification of Randomized Distributed Real-Time Systems, Ph.D. thesis, MIT, 1995.
[32] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[33] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Formal Aspects of Computing 6 (1994) 512–535.
[34] J. A. Bergstra, J. W. Klop, ACP_τ: A universal axiom system for process specification, in: Algebraic Methods: Theory, Tools and Applications, volume 394 of *LNCS*, Springer, 1989, pp. 447–463.
[35] M. Timmer, M. I. A. Stoelinga, J. C. van de Pol, Confluence reduction for probabilistic systems, in: Proc. of the 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 6605 of *LNCS*, Springer, 2011, pp. 311–325.
[36] M. Timmer, M. I. A. Stoelinga, J. C. van de Pol, Confluence Reduction for Probabilistic Systems (extended version), Technical Report 1011.2314, ArXiv e-prints, 2010.
[37] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A toolbox for the construction and analysis of distributed processes, in: Proc. of the 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 6605 of *LNCS*, Springer, 2011, pp. 372–387.
[38] A. Itai, M. Rodeh, Symmetry breaking in distributed networks, Information and Computation 88 (1990) 60–87.
[39] W. Fokkink, J. Pang, Simplifying Itai-Rodeh leader election for anonymous rings, in: Proc. of the 4th Int. Workshop on Automated Verification of Critical Systems (AVoCS), volume 128(6) of *ENTCS*, Elsevier, 2005, pp. 53–68.
[40] J.-P. Katoen, J. C. van de Pol, M. I. A. Stoelinga, M. Timmer, A linear process algebraic format for probabilistic systems with data (extended version), Technical Report, TR-CTIT-10-11, CTIT, University of Twente, 2010.
[41] M. I. A. Stoelinga, Alea jacta est: verification of probabilistic, real-time and parametric systems, Ph.D. thesis, University of Nijmegen, 2002.

Appendix A. Proof of Theorem 19

Before proving Theorem 19, we first provide three lemmas. 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 34. *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 $\boldsymbol{D}$ some data type, then in every context also*

$$p + q \approx p' + q' \tag{A.1}$$

$$\sum_{\boldsymbol{x}:\boldsymbol{D}} p \approx \sum_{\boldsymbol{x}:\boldsymbol{D}} p' \tag{A.2}$$

$$c \Rightarrow p \approx c \Rightarrow p' \tag{A.3}$$

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.

(A.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, 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 $\mu \sim_{R_p} \mu'$, and thus $p' + q' \xrightarrow{\alpha} \mu'$. 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 [41]).

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.

(A.2). Let R be the symmetric, reflexive, and transitive closure of $R_p \cup \{(\sum_{\boldsymbol{x}:\boldsymbol{D}} p, \sum_{\boldsymbol{x}:\boldsymbol{D}} p')\}$. We show that 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, $\sum_{\boldsymbol{x}:\boldsymbol{D}} p \xrightarrow{\alpha} \mu$ if and only if there is a $\boldsymbol{d} \in \boldsymbol{D}$ such that $p[\boldsymbol{x} := \boldsymbol{d}] \xrightarrow{\alpha} \mu$. From the assumption that $p \approx p'$ in any context it immediately follows that $p[\boldsymbol{x} := \boldsymbol{d}] \approx p'[\boldsymbol{x} := \boldsymbol{d}]$ for any $\boldsymbol{d} \in \boldsymbol{D}$, so if $p[\boldsymbol{x} := \boldsymbol{d}] \xrightarrow{\alpha} \mu$ then $p'[\boldsymbol{x} := \boldsymbol{d}] \xrightarrow{\alpha} \mu'$ with $\mu \sim_{R_p} \mu'$. Now, using symmetry and Proposition 5.2.1 of [41] again, statement (A.2) follows.

(A.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, $c \Rightarrow p \approx c \Rightarrow p'$, since they both have no behaviour. □

The following two lemmas prove termination of Algorithm 1, and provide an invariant for its loop. For the first lemma we need to introduce *subterms* of process terms and specifications.

Definition 35. *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 $P = (E, I)$ be a specification, then subterms(P) $= \{p \mid \exists (X_i(\boldsymbol{x_i} : \boldsymbol{D_i}) = p_i) \in E \,.\, p \in \mathit{subterms}(p_i)\}$.*

Lemma 36. *Algorithm 1 terminates for every finite specification P.*

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 has 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 that 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 37. *Let $P = (E, X_1(\boldsymbol{v}))$ be the input prCRL specification for Algorithm 1 (having unique variable names), and let $\boldsymbol{v}'$ be the computed new initial vector. Then, before and after an arbitrary iteration of the algorithm's while loop, $(E \cup \text{done} \cup \text{toTransform}, X_1'(\boldsymbol{v}')) \approx P$.*

Proof. For brevity, in this proof we abbreviate 'strongly probabilistically bisimilar in any context' by 'bisimilar'. Also, the notation $p \approx q$ will be used to denote that p and q are strongly probabilistically bisimilar in any context.

We prove this lemma by induction on the number of iterations that have already been performed. Let $P = (\{X_1(\boldsymbol{x_1} : \boldsymbol{D_1}) = p_1, \ldots, X_n(\boldsymbol{x_n} : \boldsymbol{D_n}) = p_n\}, X_1(\boldsymbol{v}))$ be an arbitrary specification, provided as input to Algorithm 1.

Before the first iteration, the parameters of the new processes are determined. Every process will have the same parameters: $\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}$. This is the union of all process variables 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 added parameters. Furthermore, *done* is set to $\emptyset$ and *toTransform* to $\{X_1'(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p_1\}$.

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 free in $X_1(\boldsymbol{x_1} : \boldsymbol{D_1}) = p_1$ (which is not allowed by Definition 5). Therefore, $(P \cup \mathit{done} \cup \mathit{toTransform}, X_1'(\boldsymbol{v}'))$ and P are obviously bisimilar.

Now assume that k iterations have passed. Without loss of generality, assume that each time a process $(X_i'(\mathit{pars}) = p_i) \in \mathit{toTransform}$ had to be chosen, it was the one with the smallest i. Then, after these

k iterations, $\mathit{done} = \{X'_1(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_1, \ldots, X'_k(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_k\}$. Also, $\mathit{toTransform} = \{X'_{k+1}(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_{k+1}, \ldots, X'_l(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_l\}$ for some $l \geq k$. The induction hypothesis is that $(P \cup \mathit{done} \cup \mathit{toTransform}, X'_1(\boldsymbol{v'})) \approx P$.

We prove that after $k+1$ iterations, still $(P \cup \mathit{done} \cup \mathit{toTransform}, X'_1(\boldsymbol{v'})) \approx P$. During iteration $k+1$ three things happen: (1) the process equation $X'_{k+1}(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_{k+1}$ is removed from $\mathit{toTransform}$; (2) an equation $X'_{k+1}(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p''_{k+1}$ is added to done; (3) potentially, one or more equations $X'_{l+1}(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_{l+1}, \ldots, X'_m(\boldsymbol{x_1} : \boldsymbol{D_1}, \boldsymbol{x'} : \boldsymbol{D'}) = p'_m$ are added to $\mathit{toTransform}$.

As the other equations in $P \cup \mathit{done} \cup \mathit{toTransform}$ do not change, $(P \cup \mathit{done} \cup \mathit{toTransform}, X'_1(\boldsymbol{v'}))$ is still bisimilar to 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 $p'_{k+1} = a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : q$. We now make a case distinction based on whether there already is a process equation in either done or $\mathit{toTransform}$ whose right-hand side is an IRF corresponding to the normal form of q (which is 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 $\mathit{toTransform}$ via line 6 of Algorithm 2, and $p''_{k+1} = a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : X'_{l+1}(\mathit{actualPars})$.
>
> 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 4 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}, hence, they are bisimilar.
>
> When $q = Y(t_1, t_2, \ldots, t_n)$, 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 $t_1, t_2, \ldots, t_n$ 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 2 of Algorithm 3, so $p''_{k+1} \approx p'_{k+1}$.

Case 1b: *There 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 $p''_{k+1} = a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : X'_j(\mathit{actualPars})$ 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 $\mathit{toTransform}$. 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 $\mathit{toTransform}$.
>
> In both cases, $\mathit{done} \cup \mathit{toTransform} \cup P$ contains a process $X'_j(\mathit{pars}) = q''$ such that $q'' \approx q'$, and therefore it is correct to take $p''_{k+1} = a(\boldsymbol{t})\sum_{\boldsymbol{x}:\boldsymbol{D}} f : X'_j(\mathit{actualPars})$. The reasoning to see that indeed $p''_{k+1} \approx p'_{k+1}$ then only depends on the choice of $\mathit{actualPars}$, 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 34.

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* $Y(\boldsymbol{x} : \boldsymbol{D}) = q_1$.

> 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 probabilistically bisimilar to p'_{k+1} in any context, the lemma holds. □

Theorem 19. *Let P be a prCRL specification such that all variables are named uniquely. Given this input, Algorithm 1 terminates, and the specification P' it returns is such that $P' \approx P$. Also, P' is in IRF.*

Proof. Let $P = (E, X_1(\boldsymbol{v}))$ be a specification such that all variables are named uniquely. Lemma 36 already provided termination, and Lemma 37 provided the invariant that $(E \cup \mathit{done} \cup \mathit{toTransform}, X'_1(\boldsymbol{v}')) \approx P$. 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 $E \cup \mathit{toTransform}$, and that *done* is in IRF.

First of all, note that upon termination $\mathit{toTransform} = \emptyset$ 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 6). 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 6; 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). □