Simplifying Process Parameters by Unfolding Algebraic Data Types

Anna Stramaglia, Jeroen J.A. Keiren, and Thomas Neele

Eindhoven University of Technology, the Netherlands
{a.stramaglia,j.j.a.keiren,t.s.neele}@tue.nl

Abstract. Complex abstract data types are often used to facilitate creating concise models of the behavior of realistic systems. However, static analysis techniques that aim to optimize such models often consider variables of complex types as a single indivisible unit. The use of complex data types thus negatively affects the optimizations that can be performed. In this paper we revisit and extend a technique by Groote and Lisser that can be used to replace a single, complex variable by multiple variables of simpler data types, improving the effectiveness of other static analyzes. We describe the technique in the context of the process algebraic specification language mCRL2, and establish its correctness. We demonstrate using an implementation in the mCRL2 toolset that it sometimes reduces the size of the underlying state spaces, and it typically reduces the verification times when using symbolic model checking.

1 Introduction

The mCRL2 language [7] is a process algebraic specification language with an associated toolset to model, validate and verify complex systems [3]. Models in mCRL2 typically consist of a number of (communicating) parallel processes that are parameterized with data. As preprocessing for further analysis, mCRL2 specifications are transformed into *linear process equations* (LPEs). In this step, parallelism and communication are removed from the process definition. Therefore, an LPE consists of a single (recursive) process definition, parameterized with data, and a number of condition-action-effect rules referred to as *summands*.

The mCRL2 toolset, among other features, offers several manipulation tools for LPEs (e.g. constant elimination and unused parameter elimination, see [6]). The transformations applied by these tools mainly operate on process parameters and aim to reduce the complexity of the LPE under consideration. They can result in a reduction (under bisimilarity) of the underlying state space.

To facilitate the modeling of realistic processes, mCRL2 supports complex algebraic data types. However, since the LPE transformations do not consider the structure within the data type of a process parameter, using complex data types reduces the effectiveness of these transformations. Thus, in order to benefit from their full potential, we need to simplify these complex data structures.

To address this, Groote and Lisser [6] originally introduced a technique for flattening the structure of process parameters and implemented this under the name `structelm` in µCRL [1] (the precursor of mCRL2). The idea behind this approach is to replace a given parameter p by multiple parameters $p_1, \ldots, p_n$ that together encode the data type of p, effectively exposing its internal structure. This enables the application of the aforementioned LPE simplification techniques to $p_1, \ldots, p_n$, which would otherwise not be possible [6]. This same technique was implemented in the mCRL2 tool `lpsparunfold`.

In this paper, we revisit the transformation behind `lpsparunfold` and identify several constructions that occur often in LPEs, but are not dealt with adequately, limiting the practical applicability of `lpsparunfold`. We extend this technique to enable further simplifications and preserve bisimilarity of the LPE. Our contributions are:

– we identify an alternative way of placing the functions that reconstruct the original parameter p from its unfolded constituents $p_1, \ldots, p_n$,
– we allow the technique to preserve global variables in such a way that they can be effectively used by other static analysis techniques,
– we simplify complex state update expressions by locally eliminating functions that are defined using *pattern matching*, and
– we experimentally demonstrate that our extensions are effective at enabling other LPE transformations and speeding up the model checker.

In particular, our experiments show that our extensions enable larger reductions of the underlying state space, directly benefiting explicit-state model checking. For symbolic reachability we observe that, even if no state space reduction is possible, the flattening achieved by `lpsparunfold` reduces the execution time.

Related work. Our work is most closely related to various analysis and transformation techniques for LPEs that have been developed over the years. The aforementioned elimination techniques from [6] are a prime example. A more advanced algorithm is *liveness analysis* [16], which reconstructs a control flow graph from a given LPE and uses knowledge of relevant data parameters to reduce the size of the underlying state space.

Similar ideas have been developed for *Parameterized Boolean Equation Systems* (PBES) [8]. For example, redundant and constant parameter elimination for PBES is presented in [13], liveness analysis in [10]; a generalization of constant elimination occurs in [12].

The implementation of symbolic reachability used in our experiments is based on the techniques from [2, 11], and uses the decision diagrams from Sylvan [17].

Overview. Section 2 introduces an example that is used throughout the paper. Next, preliminaries are provided in Section 3. Parameter unfolding from [6] is presented in Section 4. Our extensions to [6] are presented in Section 5. Finally, we validate our ideas with experiments in Section 6 and conclude in Section 7.

2 Motivating Example

As a running example we use an mCRL2 specification of a simple system, shown in Figure 1, inspired by the mCRL2 models generated from OIL specifications [4].

$$
\begin{array}{ll}
\textbf{sort} & \mathit{Sys} = \textbf{struct}\ \mathit{uninit} \mid \mathit{sys}(\mathit{get\_state} : \mathit{State}, \mathit{get\_ip} : \mathbb{N});\\
 & \mathit{State} = \textbf{struct}\ \mathit{p\_on} \mid \mathit{p\_off};\\
\textbf{map} & \mathit{set\_state}\colon \mathit{Sys} \times \mathit{State} \to \mathit{Sys};\\
 & \mathit{set\_ip}\colon \mathit{Sys} \times \mathbb{N} \to \mathit{Sys};\\
\textbf{var} & s\colon \mathit{Sys}; p_1, p_2\colon \mathit{State}; n, m\colon \mathbb{N};\\
\textbf{eqn} & \mathit{set\_state}(\mathit{uninit}, p_1) = \mathit{uninit};\\
 & \mathit{set\_state}(\mathit{sys}(p_1, n), p_2) = \mathit{sys}(p_2, n);\\
 & \mathit{set\_ip}(\mathit{uninit}, m) = \mathit{uninit};\\
 & \mathit{set\_ip}(\mathit{sys}(p_1, n), m) = \mathit{sys}(p_1, m);\\
\textbf{act} & \mathit{on}, \mathit{off}, \mathit{initialize};\\
\textbf{glob} & \mathit{dc1}, \mathit{dc2} : \mathbb{N};\\
\textbf{proc} & P(s\colon \mathit{Sys}) =\\
 & \quad (s \approx \mathit{uninit}) \to \mathit{initialize} \cdot P(\mathit{sys}(\mathit{p\_off}, \mathit{dc1}))\\
 & \quad + \sum_{n\colon \mathbb{N}} (s \not\approx \mathit{uninit} \land \mathit{get\_state}(s) \approx \mathit{p\_off})\\
 & \qquad\quad \to \mathit{on} \cdot P(\mathit{set\_state}(\mathit{set\_ip}(s, n), \mathit{p\_on}))\\
 & \quad +(s \not\approx \mathit{uninit} \land \mathit{get\_state}(s) \approx \mathit{p\_on}) \to \mathit{off} \cdot P(\mathit{set\_state}(\mathit{set\_ip}(s, \mathit{dc2}), \mathit{p\_off}));\\
\textbf{init} & P(\mathit{uninit});
\end{array}
$$

Fig. 1. Linear process specification of a simple system

The system starts out *uninitialized* (**init** $P(\mathit{uninit})$). If the system is *initialized*, it can be in one of two states, *off* or *on*, and can be toggled between them. Moreover, an IP address, abstracted as natural number, is assigned to the system.

The LPE is given after the **proc** keyword. The definition of the process consists of (possibly recursive) summands, that, essentially, describe a set of condition-action-effect rules. When *uninitialized*, it can be *initialized* to *off*, where the IP address is irrelevant. This is modeled using a global variable *dc1* (*dc* stands for don't-care). When the system is *off*, it can be switched *on*, and the IP address is set to an arbitrary value using the sum operator $\sum_{n\colon \mathbb{N}}$. If the system is *on*, the system can be switched *off*. Again the IP address is immaterial.

The LPE is defined in the context of the data specification, which consists of several parts. First, **sort** specifies two sorts. Structured sort *Sys* has two constructors, $\mathit{uninit}\colon \mathit{Sys}$ and $\mathit{sys}\colon \mathit{State} \times \mathbb{N} \to \mathit{Sys}$. For this, standard operations such as equality ($\approx$) and inequality ($\not\approx$) are predefined, e.g., that $\mathit{sys}(p, n) \not\approx \mathit{uninit}$ for all $p\colon \mathit{State}$, $n\colon \mathbb{N}$. Also, the projection functions $\mathit{get\_state}\colon \mathit{Sys} \to \mathit{State}$ and $\mathit{get\_ip}\colon \mathit{Sys} \to \mathbb{N}$ are defined such that, $\mathit{get\_state}(\mathit{sys}(p, n)) = p$ and $\mathit{get\_ip}(\mathit{sys}(p, n)) = n$. The *State* argument indicates the status of the system which can be set to *p_on* or *p_off*, e.g., given $s\colon \mathit{Sys}$, function $\mathit{set\_state}(s, \mathit{p\_on})$ sets the state of s to *p_on*. Similarly, $\mathit{set\_ip}(s, \mathit{1})$ sets the IP address of s to 1.

Note that the labeled transition system underlying this process has an infinite state space due to the use of natural numbers for IP addresses. However, this parameter does not affect the behavior of the system: the behavior of the system when it is *on*, *i.e.*, it is in a state $\mathit{sys}(\mathit{p\_on}, n)$, is bisimilar for all values of n. Yet, static analysis tools such as parameter elimination and constant elimination are not able to simplify the LPE because the real structure of the process is hidden in process parameter s.

3 Preliminaries

The mCRL2 language is a modeling language based on process algebra with data [7]. In general, the language allows the specification of communicating, parallel processes. However, the first step in any automated analysis using the mCRL2 toolset [3] is to *linearize* the specification. In this process, parallel composition operators are eliminated, and replaced by sequential composition and choice, effectively making the allowed interleavings explicit. This results in a standardized format for processes, the *linear process equations* (LPEs). The technique we study in this paper operates on such LPEs. In the remainder of this section we first introduce the data, and subsequently the LPEs.

3.1 Data

The language for data types in mCRL2 is based on an algebraic specification. We here give a brief overview. For details, we refer to the treatment in [7]. A *signature* is a triple $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ where $\mathcal{S}$ is a set of sorts, $\mathcal{C}_\mathcal{S}$ and $\mathcal{M}_\mathcal{S}$ are disjoint sets of function symbols over $\mathcal{S}$, called *constructors*, and *mappings*, respectively. Such function symbols are of the form $f\colon D_1 \times \dots \times D_n \to D$ such that $D_i, D \in \mathcal{S}$ for $1 \leq i \leq n$. If $n = 0$, we say f is a constant. We generally assume that the signature contains Booleans and standard numeric types along with their constructors and operations. With a slight abuse of notation we use their semantic sets $\mathbb{B}, \mathbb{N}, \dots$ and operations such as $\wedge$ and $+$ also for the syntactic counterparts. For any sort D, we assume sort $\mathit{List}(D)$ is defined, with constructors `[]` for the empty list, and $\triangleright$ for the constructor that adds an element in front of a list. Sorts constructed using $\to$, such as $D_1 \times \dots \times D_n \to D$ are called function sorts. If $D = D_1 \times \dots \times D_n \to D'$ we write $\mathit{range}(D)$ for its range D'.

Constructors are used to inductively define the elements of a sort. We write $\mathcal{C}_\mathcal{S}(D) = \{f\colon D' \in \mathcal{C}_\mathcal{S} \mid \mathit{range}(D') = D\}$ for the constructors of sort D. We assume a bijection ι_D between $\mathcal{C}_\mathcal{S}(D)$ and $0..|\mathcal{C}_\mathcal{S}(D)|-1$ ordering the constructors, and write ι if D is clear from the context. We say that D is a *constructor sort* if, and only if, $\mathcal{C}_\mathcal{S}(D) \neq \emptyset$. A constructor sort D is syntactically non-empty if there is a constructor $f\colon D_1 \times \dots \times D_n \to D$ such that if D_i is a constructor sort, then D_i is syntactically non-empty, for $1 \leq i \leq n$. We require all constructor sorts to be non-empty, and for $f\colon D \in \mathcal{C}_\mathcal{S}$, $\mathit{range}(D)$ must not be a function sort.

Expressions in the data language are referred to as *data expressions* or *terms* over a set $\mathcal{X}_\mathcal{S}$ of $\mathcal{S}$-sorted variables. They are syntactically described by the following grammar:

$$t ::= x \mid f \mid t(t, \dots, t)$$

where $x \in \mathcal{X}_\mathcal{S}$ are variables, $f \in \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$ are function symbols, and $t(t, \dots, t)$ describes the application of a term to its arguments. We write $e[x := e']$ for the syntactic substitution of x with e' in e. The mCRL2 language additionally supports quantification and lambda expressions. Our technique straightforwardly extends to this setting, so we omit those constructs for the sake of simplicity. With every sort D, we associate a default expression, def_D.

Equality of terms is defined using a *data specification* $\mathcal{D} = (\Sigma, E)$, where Σ is a signature and E is a set of conditional equations of the form $\langle \mathcal{X}, c \to t = u \rangle$, where $\mathcal{X} \subseteq \mathcal{X}_{\mathcal{S}}$, and c, t, u are terms over $\mathcal{X}$. We typically write $\langle \mathcal{X}, t = u \rangle$, when $c = \mathit{true}$ and $c \to t = u$ or $t = u$, if $\mathcal{X}$ is clear from the context.

The semantics of data types is described using *model class semantics* [7]. Sorts are mapped into their semantic counterpart using applicative structures. A set $\{M_D \mid D \in \mathcal{S}\}$ is an applicative structure if, and only if, $M_{\mathbb{B}} = \{\mathbf{true}, \mathbf{false}\}$, and if $D = D_1 \times \cdots \times D_n \to D'$, then M_D contains all (semantic) functions from $M_{D_1} \times \cdots \times M_{D_n} \to M_{D'}$. Function $[\![-]\!]$ maps every function symbol in the data specification into its semantic counterpart, that is, for all $f \in \mathcal{C}_{\mathcal{S}} \cup \mathcal{M}_{\mathcal{S}}$ of sort D, $[\![f]\!] \in M_D$. This is generalized to arbitrary terms as follows:

$$\begin{aligned} [\![x]\!]^\sigma &= \sigma(x) && \text{if } x \in \mathcal{X}_{\mathcal{S}} \\ [\![f]\!]^\sigma &= [\![f]\!] && \text{if } f \in \mathcal{C}_{\mathcal{S}} \cup \mathcal{M}_{\mathcal{S}} \\ [\![t(t_1, \ldots, t_n)]\!]^\sigma &= [\![t]\!]^\sigma([\![t_1]\!]^\sigma, \ldots, [\![t_n]\!]^\sigma) \end{aligned}$$

where $\sigma \colon \mathcal{X}_{\mathcal{S}} \to \bigcup_{D \in \mathcal{S}} M_D$ is a valuation that ensures that $\sigma(x) \in M_D$ for all $x \colon D$. We write $\sigma[v/d]$ for the valuation that assigns v to d and otherwise behaves as σ. The model $\mathbb{M}$ of a data specification is an applicative structure together with an interpretation function, that in addition ensures that for equations $\langle \mathcal{X}, c \to t = u \rangle \in E$ and valuations σ, if $[\![c]\!]^\sigma = \mathbf{true}$ then $[\![t]\!]^\sigma = [\![u]\!]^\sigma$; $[\![\mathit{true}]\!]^\sigma = \mathbf{true}$, $[\![\mathit{false}]\!]^\sigma = \mathbf{false}$, for all valuations σ; and if D is a constructor sort, then every $v \in M_D$ is a *constructor element*. Element $v \in M_D$ is a constructor element if a constructor function $f \in \mathcal{C}_{\mathcal{S}}$ of sort $D_1 \times \cdots \times D_n \to D$ exists such that $v = [\![f]\!](v_1, \ldots, v_n)$ where v_i is either a constructor element of sort D_i, or sort D_i is not a constructor sort. We write $t \equiv t'$ for terms t and t' if for all models, $[\![t]\!]^\sigma = [\![t']\!]^\sigma$ for all valuations σ.

3.2 Linear Processes

A Linear Process Equation (LPE) defines the name of a recursive process, whose definition is a set of summands that are, essentially, condition-action-effect rules that may refer to local variables. An LPE is typically defined in the context of a data specification $\mathcal{D}$, that specifies algebraic data types, and a set of global variables $\mathcal{X}_g$. The combination of an LPE with a data specification and its global variables is a Linear Process Specification (LPS).

Definition 1. *A linear process specification (LPS) L is a tuple $(\mathcal{D}, \mathcal{X}_g, P, \vec{e})$ where $\mathcal{D}$ is a data specification describing the data types used in the LPS, $\mathcal{X}_g$ is a set of global variables, P is a linear process equation (LPE), and $\vec{e}$ is a vector of expressions of sort $\vec{D}$ that may refer to variables in $\mathcal{X}_g$. We typically say that $P(\vec{e})$ is the initial process. LPE P is described as follows:*

$$P(\vec{d} \colon \vec{D}) = \sum_{i \in I} \sum_{\vec{e_i} \,:\, \vec{E}_i} c_i \to a_i(f_i) \cdot P(g_i) + \sum_{j \in J} \sum_{\vec{e_j} \,:\, \vec{E}_j} c_j \to a_{\delta j}(f_j)$$

where $\vec{d}$ is a vector of process parameters whose types are $\vec{D}$. I and J are disjoint, finite index sets, such that for $i \in I$ and $j \in J$ we have that c_i and c_j are boolean expressions, a_i and $a_{\delta j}$ are actions, f_i and f_j are terms that form the action parameters, and g_i is the next state, providing the vector of terms assigned to the parameters of process P in the recursive call to P. Terms c_i, f_i, g_i (c_j, f_j) range over $\vec{d}$, $\mathcal{X}_g$, and $\vec{e_i}$ ($\vec{e_j}$).

In their full generality, LPEs can use timestamps on the actions. These timestamps are treated by our transformation in the same way as action parameters. For the sake of simplicity, we restrict ourselves to untimed LPEs in this paper. For the same reason, we will henceforth only consider recursive summands.

Transformations of LPEs are correct if they are behavior preserving. For this, we use a generalization of strong bisimulation to linear processes [6]. Two LPEs P and P' with initial values e and e', respectively, are strongly bisimilar if and only if the labeled transition systems induced by $P(e)$ and $P'(e')$ are strongly bisimilar. In this case, we write $P(e) \leftrightarrow P'(e')$.

4 Parameter Unfolding

Parameter unfolding was introduced by Groote and Lisser under the name `structelm` [6], and has later been implemented in the mCRL2 toolset in a tool called `lpsparunfold`. The idea behind parameter unfolding is that a term from a constructor sort whose head symbol is a constructor can be replaced by separate terms for the name of the constructor and each of the arguments. For instance, in our running example, the single process parameter s is then replaced by three process parameters: $e\colon U_{Sys}$, $s_1\colon State$ and $s_2\colon \mathbb{N}$, where e represents the constructor at the head of s, and s_1 and s_2 are the arguments of the first constructor. The term $sys(p\_off, dc1)$ in the first summand of our running example can be replaced by the terms $\overline{c}_{sys}$, p_off, and $dc1$; the Sys constructor $uninit$ in the initialization is replaced by the value $\overline{c}_{uninit}$. As $uninit$ does not have any parameter, the new parameters s_1 and s_2 can be set to a default value.

Unfolding of process parameters happens in two steps. First, the data specification is extended with a new sort to represent constructors, and mappings to move between the sort that is unfolded, and newly introduced parameters. Next, the parameters in the linear process are unfolded.

4.1 Extending the Data Specification

Our improvements to Groote and Lisser's technique concern the unfolding of the parameters in the linear process. The extension of the data specification is, in essence, left unchanged, and its formal definition can be found in [6]. We therefore only introduce the unfolding of the data type using our running example.

When unfolding a sort D, a new data specification is constructed that extends $\mathcal{D}$ with a new sort U_D, to represent the constructors of D, constructors for this new sort, as well as case functions, determinizers and projection functions and the associated equations.

Example 1. Recall the data specification from Figure 1. We unfold sort Sys. Note that $\mathcal{C}_{\mathcal{S}}(Sys) = \{sys\colon State \times \mathbb{N} \to Sys, uninit\colon Sys\}$, that is it has two constructors, sys and $uninit$. The data specification of the running example is extended with the following.

$$
\begin{array}{ll}
\textbf{sort} & U_{Sys};\\
\textbf{cons} & \overline{c}_{sys}, \overline{c}_{uninit}\colon U_{Sys};\\
\textbf{map} & C_{Sys}\colon U_{Sys} \times Sys \times Sys \to Sys\\
 & det_{Sys}\colon Sys \to U_{Sys};\\
 & \pi^1_{sys}\colon Sys \to State;\\
 & \pi^2_{sys}\colon Sys \to \mathbb{N};\\
\textbf{var} & x, x_1, x_2\colon Sys; e\colon U_{Sys};\\
 & y_1\colon State; y_2\colon \mathbb{N};
\end{array}
\qquad
\begin{array}{ll}
\textbf{eqn} & C_{Sys}(\overline{c}_{uninit}, x_1, x_2) = x_1;\\
 & C_{Sys}(\overline{c}_{sys}, x_1, x_2) = x_2;\\
 & C_{Sys}(e, x, x) = x;\\
 & det_{Sys}(uninit) = \overline{c}_{uninit};\\
 & det_{Sys}(sys(y_1, y_2)) = \overline{c}_{sys};\\
 & \pi^1_{sys}(uninit) = p\_on;\\
 & \pi^2_{sys}(uninit) = 0;\\
 & \pi^1_{sys}(sys(y_1, y_2)) = y_1;\\
 & \pi^2_{sys}(sys(y_1, y_2)) = y_2;
\end{array}
$$

The explanation of the additions is as follows. We add constructor sort U_{Sys}, with constructors $\overline{c}_{sys}, \overline{c}_{uninit}$, i.e., we introduce one new constructor in sort U_{Sys} for every constructor in the unfolded sort. Case function C_{Sys} is used in the unfolding of processes to reconstruct an expression of sort Sys from the unfolded parts, e.g., $C_{Sys}(\overline{c}_{sys}, uninit, sys(p\_on, 3)) = sys(p\_on, 3)$. The case $C_{sys}(e, x, x) = x$ is used to facilitate simplifications in the implementation even when the arguments do not yet have a concrete value. We add determinizer functions det_{Sys} that are used to recognize the head symbol of an expression of sort Sys, and map it onto the corresponding constructor in U_{Sys}, e.g., $det_{Sys}(sys(p\_on, 3)) = \overline{c}_{sys}$. Projection functions π^1_{sys} and π^2_{sys} are added to extract the arguments of an expression with head symbol sys, e.g., $\pi^2_{sys}(sys(p\_on, 3)) = 3$; if this projection function is applied to $uninit$ it returns a default value. Since constructor $uninit$ has no arguments, there are no projection functions π_{uninit}.

To be effective in practice, the projection and determinizer functions need to distribute over if-then-else and the case functions. Therefore, also the following distribution laws are added.

$$
\begin{array}{ll}
\textbf{var} & x_1, x_2\colon Sys; e\colon U_{Sys}; b\colon \mathbb{B}\\
\textbf{eqn} & \pi^1_{sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, \pi^1_{sys}(x_1), \pi^1_{sys}(x_2));\\
 & \pi^1_{sys}(\mathit{if}(b, x_1, x_2)) = \mathit{if}(b, \pi^1_{sys}(x_1), \pi^1_{sys}(x_2));\\
 & \pi^2_{sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, \pi^2_{sys}(x_1), \pi^2_{sys}(x_2));\\
 & \pi^2_{sys}(\mathit{if}(b, x_1, x_2)) = \mathit{if}(b, \pi^2_{sys}(x_1), \pi^2_{sys}(x_2));\\
 & det_{Sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, det_{Sys}(x_1), det_{Sys}(x_2));\\
 & det_{Sys}(\mathit{if}(b, x_1, x_2)) = \mathit{if}(b, det_{Sys}(x_1), det_{Sys}(x_2));
\end{array}
$$

4.2 Unfolding Process Parameters in an LPE

We next describe how to unfold a process parameter d in an LPE, and how to split expressions e that were assigned to d into expressions that can be assigned to the new process parameters. As our extensions modify these definitions, we present them in more detail. For the sake of simplicity, we describe the unfolding in the setting of an LPE with a single process parameter. For processes with multiple process parameters, this generalizes in the obvious way.

Definition 2. *Let $d\colon D$ be a variable of constructor sort D, with $\mathcal{C}_{\mathcal{S}}(D) = \{f_0, \dots, f_n\}$. Let D be such that for all constructors $f\colon D_1 \times \cdots \times D_n \to D \in \mathcal{C}_{\mathcal{S}}(D)$, and terms $t_1, \dots, t_n, t'_1, \dots, t'_n$, if $f(t_1, \dots, t_n) \equiv f(t'_1, \dots, t'_n)$ then $t_i \equiv t'_i$ for all i.*

We first define how process parameters are manipulated.

- *First, if we unfold parameter d, new parameters need to be introduced to store the arguments for each of the constructors of sort d. For $f_i\colon D_i^1 \times \cdots \times D_i^{n_i} \to D \in \mathcal{C}_{\mathcal{S}}(D)$, this is the vector $\mathsf{params}(d, f_i) = d_i^1\colon D_i^1, \dots, \dots d_i^{n_i}\colon D_i^{n_i}$. Note that if f_i is a constant, $\mathsf{params}(d, f_i)$ is the empty vector.*
- *To define the parameters unfolding d we need one variable that represents the constructor, and parameters for the arguments of each of the constructors.*

$$\mathsf{params}(d) = e_d\colon U_D, \mathsf{params}(d, f_1), \dots, \mathsf{params}(d, f_n)$$

  Note $e_d\colon U_D$ determines the constructor of sort D, with U_D the corresponding constructor sort.
- *If d is replaced by $\mathsf{params}(d)$, any use of d needs to be reconstructed using an equivalent expression in terms of the new parameters. We abbreviate this by $\mathsf{reconstruct}(d)$.*

$$\mathsf{reconstruct}(d) = C(e_d, f_0(\mathsf{params}(d, f_0)), \dots, f_n(\mathsf{params}(d, f_n)))$$

If originally e was assigned to d, after d has been replaced by $\mathsf{params}(d)$, expression e also needs to be split into expressions that can be assigned to these new parameters. We define the following. Let e be an expression of type D. Then

$$\mathsf{unfold}(e) = det_D(e), \pi^1_{f_0}(e), \dots, \pi^{m_0}_{f_0}(e), \dots, \pi^1_{f_n}(e), \dots, \pi^{m_n}_{f_n}(e)$$

where m_i denotes the index of the last argument of constructor f_i.

The unfolding of process parameters described in [6] is as follows. In the rest of this paper, we will refer to this as using *default case placement.*

Definition 3 (Unfolding of process parameters [6]). *Let $L = (\mathcal{D}, \mathcal{X}_g, P, e)$ be an LPS, where P is the following LPE.*

$$P(d\colon D) = \sum_{i \in I} \sum_{\vec{e_i}\colon \vec{E}_i} c_i \to a_i(f_i) \cdot P(g_i)$$

The result of unfolding process parameter $d\colon D$ in L, denoted $\mathsf{parunfold}(d)(L)$ is the LPS $(\mathcal{D}', \mathcal{X}_g, P', \mathsf{unfold}(e))$, where $\mathcal{D}'$ is data specification $\mathcal{D}$ in which sort D is unfolded, and LPE P' is as follows:

$$\begin{aligned} P'(\mathsf{params}(d)) = \sum_{i \in I} \sum_{\vec{e_i}\colon \vec{E}_i} & c_i[d := \mathsf{reconstruct}(d)] \\ & \to a_i(f_i[d := \mathsf{reconstruct}(d)]) \cdot P'(\mathsf{unfold}(g_i[d := \mathsf{reconstruct}(d)])) \end{aligned}$$

So, essentially, unfolding parameter d replaces d by the vector $\mathsf{params}(d)$. In the right hand side of the equation, every occurrence of d is replaced syntactically by $\mathsf{reconstruct}(d)$, i.e., an application of the corresponding case function. Finally, in the recursive calls to P, the expression that after the previous step has become $g_i[d := \mathsf{reconstruct}(d)]$, is unfolded using $\mathsf{unfold}(g_i[d := \mathsf{reconstruct}(d)])$. Similarly, using $\mathsf{unfold}(e)$, the initial process is unfolded.

Example 2. Recall our motivating example, for which we have described the unfolding of sort *Sys* in the data specification in Example 1. If we unfold parameter s, we get the LPE and initialization shown below.

$$
\begin{array}{ll}
\textbf{proc} & P(e\colon U_{Sys}, s_1\colon State, s_2\colon \mathbb{N}) \\
& \quad = C(e, uninit, sys(s_1, s_2) \approx uninit) \\
& \qquad \to initialize\cdot \\
& \qquad\quad P(det_{Sys}(sys(p\_off, dc1)), \pi^1_{sys}(sys(p\_off, dc1)), \pi^2_{sys}(sys(p\_off, dc1))) \\
& \quad + \sum_{n\colon \mathbb{N}}(!(C(e, uninit, sys(s_1, s_2)) \approx uninit)\wedge \\
& \qquad get\_state(C(e, uninit, sys(s_1, s_2))) \approx p\_off) \\
& \qquad \to on \cdot P(det_{Sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), n), p\_on)), \\
& \qquad\qquad \pi^1_{sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), n), p\_on)), \\
& \qquad\qquad \pi^2_{sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), n), p\_on))) \\
& \quad +(!(C(e, uninit, sys(s_1, s_2)) \approx uninit)\wedge \\
& \qquad get\_state(C(e, uninit, sys(s_1, s_2))) \approx p\_on) \\
& \qquad \to off \cdot P(det_{Sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), dc2), p\_off)), \\
& \qquad\qquad \pi^1_{sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), dc2), p\_off)), \\
& \qquad\qquad \pi^2_{sys}(set\_state(set\_ip(C(e, uninit, sys(s_1, s_2)), dc2), p\_off))); \\
\textbf{init} & P(det_{Sys}(uninit), \pi^1_{sys}(uninit), \pi^2_{sys}(uninit));
\end{array}
$$

It has three parameters. Parameter e keeps track of the constructor of the expression in s, e.g., initially s is *uninit*, so the corresponding value in e is $\overline{c}_{uninit}$. Parameters s_1 and s_2 are used to track the arguments of the constructor *sys*. If e is $\overline{c}_{sys}$, then $sys(s_1, s_2)$ is equivalent to s (the orginal parameter that is unfolded). As *uninit* does not have arguments, no parameters need to be introduced for its arguments. The original expression s is then reconstructed in the process by replacing s with $C(e, uninit, sys(s_1, s_2))$. The functions det_{Sys}, π^1_{sys} and π^2_{sys} are used to move from an expression of sort *Sys* to expressions of sort U_{Sys}, *State* and $\mathbb{N}$.

Using the equations for det_{Sys}, π^1_{sys} and π^2_{sys} for rewriting, this can be simplified slightly. The recursion of the first summand then becomes $P(\overline{c}_{sys}, p\_off, dc1)$ and the initialization becomes $\textbf{init}\, P(\overline{c}_{uninit}, p\_on, 0)$, as per the default values of $\pi^i_{sys}(uninit)$. The resulting LPE cannot be simplified further. Since parameters s_1 and s_2 appear in the conditions of each of the summands, existing static analysis tools for constant elimination and parameter elimination are not able to remove any of the parameters from this process.

Correctness of the unfolding is established by the following theorem.

Theorem 1 ([6]). *Let* $L = (\mathcal{D}, \mathcal{X}_g, P, e)$ *and* $\mathsf{parunfold}(d)(L) = (\mathcal{D}', \mathcal{X}_g, P', \mathsf{unfold}(e))$ *be the LPSs as in Definition 3. Then* $P(e) \leftrightarrows P'(\mathsf{unfold}(e))$.

5 Improving Parameter Unfolding

In this section we present three improvements to parameter unfolding: we alter the way case functions are placed in the processes, we explicitly take global variables into account during the transformation, and we show how pattern matching rules in the data specification can be used to simplify the data in the resulting process expressions.

5.1 Alternative Case Placement

In our standard definition of unfolding, each occurrence of d is replaced by $\mathsf{reconstruct}(d)$, and thus case functions are placed at an *innermost* level. This can limit simplification by rewriting; e.g., expression $C(e, \mathit{uninit}, \mathit{sys}(s_1, s_2)) \approx \mathit{uninit}$ in the condition of the first summand in Example 2 cannot be simplified.

In many cases, placing the case function at an *outermost* level aids rewriting and subsequent analysis of the LPE. Formally, every condition c_i now becomes $C(e_d, c_i[d := f_0(\mathsf{params}(d, f_0))], \dots, c_i[d := f_n(\mathsf{params}(d, f_n))])$. However, this may lead to an exponential blow-up in the size of the conditions if multiple parameter unfoldings are performed successively. Therefore, we propose an intermediate approach that places case functions at the level where subexpressions are no longer Boolean. We call this *alternative case placement.* Intuitively, starting from the outermost placement, we distribute the case function over the standard Boolean operators.

Definition 4. *Given a data expression c and a variable d, the* alternative case placement *is the expression $\mathsf{acp}(c, d)$, where acp is the recursive function:*

$$\begin{aligned}
\mathsf{acp}(b, d) &= C(e, b[d := f_1(\mathsf{params}(d, f_1))], \dots, b[d := f_n(\mathsf{params}(d, f_n))]) \\
\mathsf{acp}(\neg\varphi, d) &= \neg\mathsf{acp}(\varphi, d) \\
\mathsf{acp}(\varphi \wedge \psi, d) &= \mathsf{acp}(\varphi, d) \wedge \mathsf{acp}(\psi, d) \\
\mathsf{acp}(\varphi \vee \psi, d) &= \mathsf{acp}(\varphi, d) \vee \mathsf{acp}(\psi, d) \\
\mathsf{acp}(\varphi \Rightarrow \psi, d) &= \mathsf{acp}(\varphi, d) \Rightarrow \mathsf{acp}(\psi, d)
\end{aligned}$$

Here, φ and ψ are arbitrary terms and b is a data expression that does not have $\neg, \wedge, \vee, \Rightarrow$ as its top-level operator.

Note that in the first case of the definition of acp, $\mathsf{acp}(b, d)$ is equivalent to b if d does not occur in b, by the rule $C(e, x, x) = x$. Under alternative case placement, the unfolded LPE of Definition 3 becomes:

$$P'(\mathsf{params}(d)) = \sum_{i \in I} \sum_{\vec{e_i} : \vec{E_i}} \mathsf{acp}(c_i, d) \to a_i(\mathsf{acp}(f_i, d)) \cdot P'(\mathsf{acp}(\mathsf{unfold}(g_i), d))$$

Correctness follows immediately from the observation that $\mathsf{acp}(b, d) \equiv b[d := \mathsf{reconstruct}(d)]$ (by case analysis on e). We next discuss the benefits of alternative case placement on our running example.

Example 3. Recall our motivating example, for which we have described the unfolding of sort *Sys* in Examples 1 and 2. We show the result of the unfolding using the alternative case placement. As all summands are changed in a similar way, we focus on the last summand of the LPE:

$$\begin{aligned}
&(!C(e, \mathit{uninit} \approx \mathit{uninit}, \mathit{sys}(s_1, s_2) \approx \mathit{uninit}) \wedge \\
&\qquad C(e, \mathit{get\_state}(\mathit{uninit}) \approx \mathit{p\_on}, \mathit{get\_state}(\mathit{sys}(s_1, s_2)) \approx \mathit{p\_on})) \\
&\qquad \to \mathit{off} \cdot P(C(e, \mathit{det}_{\mathit{Sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{uninit}, \mathit{dc2}), \mathit{p\_off})), \\
&\qquad\qquad\qquad\qquad \mathit{det}_{\mathit{Sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{sys}(s_1, s_2), \mathit{dc2}), \mathit{p\_off}))), \\
&\qquad\qquad\quad C(e, \pi^1_{\mathit{sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{uninit}, \mathit{dc2}), \mathit{p\_off})), \\
&\qquad\qquad\qquad\qquad \pi^1_{\mathit{sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{sys}(s_1, s_2), \mathit{dc2}), \mathit{p\_off}))), \\
&\qquad\qquad\quad C(e, \pi^2_{\mathit{sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{uninit}, \mathit{dc2}), \mathit{p\_off})), \\
&\qquad\qquad\qquad\qquad \pi^2_{\mathit{sys}}(\mathit{set\_state}(\mathit{set\_ip}(\mathit{sys}(s_1, s_2), \mathit{dc2}), \mathit{p\_off}))))
\end{aligned}$$

Compared to the LPE using default case placement, observe that the case functions now appear at a higher level. For instance, the second conjunct of the condition was changed from $\mathit{get\_state}(C(e, \mathit{uninit}, \mathit{sys}(s_1, s_2))) \approx \mathit{p\_on})$ to $C(e, \mathit{get\_state}(\mathit{uninit}) \approx \mathit{p\_on}, \mathit{get\_state}(\mathit{sys}(s_1, s_2)) \approx \mathit{p\_on}))$. In the original, the case function cannot be simplified further, as the first argument e is a variable, and it cannot be matched to any of the rewrite rules; also, there are no rules that allow distributing equality over the case function. When applying alternative case placement, the equality appears within the scope of the arguments of the case function, and the (implicit) equations for $\approx$ can be used to simplify the individual arguments.

Similar changes can be seen in the arguments of the recursive processes. Using the equations for $\approx$, $\mathit{det}_{\mathit{Sys}}$, π^1_{sys}, π^2_{sys}, $\mathit{set\_ip}$ and $\mathit{set\_state}$, the last summand is simplified to:

$$\begin{aligned}
&(!C(e, \mathit{true}, \mathit{false}) \wedge C(e, \mathit{get\_state}(\mathit{uninit}) \approx \mathit{p\_on}, s_1 \approx \mathit{p\_on})) \\
&\qquad \to \mathit{off} \cdot P(C(e, \overline{c}_{\mathit{uninit}}, \overline{c}_{\mathit{sys}}), C(e, \mathit{p\_on}, \mathit{p\_off}), C(e, 0, \mathit{dc2}))
\end{aligned}$$

We thus obtained more concise expressions than those in Example 2. In particular, this summand no longer contains any reference to unfolded parameter s_2. The same applies to the other two summands, hence parameter s_2 can be eliminated. As a result, the sum over n in the second summand can be eliminated as well, and the final LPE we obtain is:

$$\begin{aligned}
&\textbf{proc}\ P(e\colon U_{\mathit{Sys}}, s_1\colon \mathit{State}) \\
&\qquad = C(e, \mathit{true}, \mathit{false}) \\
&\qquad\qquad \to \mathit{initialize} \cdot P(\overline{c}_{\mathit{sys}}, \mathit{p\_off}) \\
&\qquad +(!C(e, \mathit{true}, \mathit{false}) \wedge C(e, \mathit{get\_state}(\mathit{uninit}) \approx \mathit{p\_off}, s_1 \approx \mathit{p\_off})) \\
&\qquad\qquad \to \mathit{on} \cdot P(C(e, \overline{c}_{\mathit{uninit}}, \overline{c}_{\mathit{sys}}), \mathit{p\_on}) \\
&\qquad +(!C(e, \mathit{true}, \mathit{false}) \wedge C(e, \mathit{get\_state}(\mathit{uninit}) \approx \mathit{p\_on}, s_1 \approx \mathit{p\_on})) \\
&\qquad\qquad \to \mathit{off} \cdot P(C(e, \overline{c}_{\mathit{uninit}}, \overline{c}_{\mathit{sys}}), C(e, \mathit{p\_on}, \mathit{p\_off})) \\
&\textbf{init}\ \ P(\overline{c}_{\mathit{uninit}}, \mathit{p\_on});
\end{aligned}$$

Note that the original state space before the unfolding is infinite while after unfolding with alternative case placement the state space has only three states.

5.2 Global Variables

Other static analysis techniques use global variables to more effectively simplify the process. For instance, when constant elimination observes that the only change to a parameter is assigning a global variable to that parameter, the global variable can be replaced by a constant. This is safe since all values for global variables lead to bisimilar processes.

When unfolding a process parameter, the value assigned to it in the initialization or recursion may be a global variable $dc \in \mathcal{X}_g$. Applying the unfoldings described so far results in $\mathsf{unfold}(dc)$, which contains expressions such as $det_D(dc)$ and $\pi^j_{f_i}(dc)$ that cannot be rewritten further. Other static analysis techniques cannot directly use such expressions, leaving the resulting LPE more complicated than it needs to be, resulting in longer verification times.

We illustrate the issue using an example that is based on the representation of the board in the specifications of games such as tic-tac-toe.

Example 4. Process P is initialized with a singleton list $[o]$ of sort $List(Piece)$ representing the board. It also has parameters p, keeping track of the player whose turn it is, and $done$ to indicate that the game ends. As long as $done$ is false, and l contains a piece of player p whose turn it is, p is updated to the next player. If l contains a piece of the other player, a τ transition is taken, the values of l and p are set to global variables, and $done$ is set to true. If $done$ is true, the process deadlocks. This resembles what happens in models of board games such as tic-tac-toe when the game ends.

$$
\begin{array}{ll}
\textbf{sort} & Piece = \textbf{struct}\ x \mid o;\\
\textbf{map} & other\colon Piece \to Piece;\\
\textbf{eqn} & other(x) = o; other(o) = x;\\
\textbf{act} & is : Piece;\\
\textbf{glob} & dc1\colon List(Piece); dc2\colon Piece;\\
\textbf{proc} & P(l\colon List(Piece), p\colon Piece, done\colon Bool)\\
 & \qquad = (\neg done \land l \approx [other(p)]) \to \tau.P(dc1, dc2, true)\\
 & \qquad +(\neg done \land l \approx [p]) \to is(p).P([p], other(p), done);\\
\textbf{init} & P([o], o, false);
\end{array}
$$

Unfolding parameter l yields the following LPE.

$$
\begin{array}{ll}
\textbf{proc} & P(e\colon U_{Piece}, l_p\colon Piece, l_l\colon List(Piece), p\colon Piece, done\colon Bool)\\
 & \quad = \neg done \land C_{List(Piece)}(e, [], l_p \triangleright l_l) \approx [other(p)]\\
 & \qquad \to \tau.P(det_{List(Piece)}(dc1), \pi^1_{\triangleright}(dc1), \pi^2_{\triangleright}(dc1), dc2, true)\\
 & \quad +(\neg done \land C_{List(Piece)}(e, [], l_p \triangleright l_l) \approx [p])\\
 & \qquad \to is(p).P(det_{List(Piece)}([p]), \pi^1_{\triangleright}([p]), \pi^2_{\triangleright}([p]), other(p));\\
\textbf{init} & P(det_{List(Piece)}([o]), \pi^1_{\triangleright}([o]), \pi^2_{\triangleright}([o]), o, false);
\end{array}
$$

The recursion in the first summand cannot be simplified further, and constant- and redundant parameter elimination cannot remove any parameters.

Since the behavior of a process is not affected by (the value of) a global variable, the individual arguments of the expression assigned to that global variable also do not affect the behavior of the process. Therefore, instead of applying projection functions to a global variable, fresh global variables can be introduced for each of the new process parameters when unfolding a global variable. We extend the definition of unfold from Definition 2 as follows.

Definition 5. *Let e be an expression of constructor sort D. Then*

$$\mathsf{unfold}_g(e) = \begin{cases} dc_e, dc^1_{f_0}, \dots, dc^{m_0}_{f_0}, \dots, dc^1_{f_n}, \dots, dc^{m_n}_{f_n} & \text{if } e \in \mathcal{X}_g \\ \mathsf{unfold}(e) & \text{otherwise} \end{cases}$$

where $dc_e, dc^1_{f_0}, \dots, dc^{m_0}_{f_0}, \dots dc^1_{f_n}, \dots, dc^{m_n}_{f_n}$ are fresh global variables, and m_i denotes the index of the last argument of constructor f_i.

The unfolded LPE taking global variables is obtained by using unfold_g instead of unfold in Definition 3.[1] We apply this improved definition to the specification in Example 4.

Example 5. Recall the specification from Example 4. When using unfold_g instead of unfold, the recursion in the first summand becomes $P(dc_e, dc_{lp}, dc_{ll}, dc2, true)$.

This allows further simplification using constant elimination and parameter elimination and simplification using rewriting to the LPE below.

proc $P(l_p\colon Piece, p\colon Piece, done\colon Bool)$
$\quad = (\neg done \wedge l_p \approx p) \to is(p).P(p, other(p), done)$
$\quad +(\neg done \wedge l_p \approx other(p)) \to \tau.P(dc_{lp}, dc2, true);$
init $P(o, o, false);$

In particular, all case functions, determinizers and projection functions are fully removed. The transformation now essentially replaced the (fixed-length) list in the original process by its individual elements.

5.3 Simplifications for Pattern Matching Rules

In the recursion $P(\mathsf{unfold}(g_i[d := \mathsf{reconstruct}(d)]))$, we regularly obtain expressions of the shape $det_D(h(a_1, \dots, a_n))$ or $\pi^l_{f_k}(h(a_1, \dots, a_n))$ for some function symbol h that is not a constructor. Both of these cannot be rewritten any further, often due to the fact that there is insufficient information to apply the pattern matching in the equations for h. Therefore, we propose a method to perform one unfolding of the function h, allowing us to achieve the necessary simplifications. Let us first consider an example.

Example 6. Suppose we have a function *plusone*, which is defined using pattern matching, that increments every element of a list. Our linear process P updates the elements of its argument $l : List(\mathbb{N})$ using this function as follows:

map $plusone : List(\mathbb{N}) \to List(\mathbb{N});$
var $x : \mathbb{N};\ \ xs : List(\mathbb{N});$
eqn $plusone(\texttt{[]}) = \texttt{[]};$
$\quad plusone(x \triangleright xs) = (x + 1) \triangleright plusone(xs);$
proc $P(l : List(\mathbb{N})) = a \cdot P(plusone(l));$
init $P([7]);$

[1] The definition using alternative case placement can be modified to take global variables into account in the same way.

If we unfold l (default case placement), we obtain as first argument update in the summand the expression $det_{List(\mathbb{N})}(plusone(C_{List(\mathbb{N})}(e, \texttt{[]}, s_1 \triangleright s_2)))$, which cannot be rewritten any further.

Intuitively, since $det_{List(\mathbb{N})}$ considers only its argument's constructor, and *plusone* does not modify the constructor, $det_{List(\mathbb{N})}(l) = det_{List(\mathbb{N})}(plusone(l))$ for all l. However, due to the pattern matching nature of *plusone*, we can only eliminate the application of $det_{List(\mathbb{N})}$ by means of term rewriting if l is of the shape `[]` or $x \triangleright xs$. Thus, the tools are not able to deduce that the update in the example above is in fact equal to e, and that the summand does not modify e. To facilitate further static analysis in the above example, it would be helpful to have a general technique for further simplification in such situations.

Our approach is to compute a single non-pattern-matching rewrite rule for each mapping that is equivalent to its original pattern-matching-based definition. The pattern matching logic will instead be encoded in a tree of case functions. We will apply the new singly-defined rule in selected places in order to eliminate determinizer and projection functions by means of ordinary rewriting. At its core, our transformation is based on the following observation, which follows by case analysis on the top-level constructor in a_i.

Lemma 1. *Let $h : D_1 \times \dots D_n \to D$ be a mapping and $a_1, \dots, a_n$ arbitrary expressions. Then we have for any σ and any $1 \leq i \leq n$:*

$$\begin{aligned}
\llbracket h(a_1, \dots, a_n) \rrbracket^\sigma = \llbracket & C_{D_i}(det_{D_i}(a_i), \\
& h(a_1, \dots, a_{i-1}, f_1(\pi^1_{f_1}(a_i), \dots, \pi^{n_1}_{f_1}(a_i)), a_{i+1}, \dots, a_n), \dots, \\
& h(a_1, \dots, a_{i-1}, f_{|\mathcal{C}_\mathcal{S}(D)|}(\pi^1_{f_{|\mathcal{C}_\mathcal{S}(D)|}}(a_i), \dots, \pi^{n_{|\mathcal{C}_\mathcal{S}(D)|}}_{f_{|\mathcal{C}_\mathcal{S}(D)|}}(a_i)), a_{i+1}, \dots, a_n)) \rrbracket^\sigma
\end{aligned}$$

We repeatedly apply this equality until each application of h can be rewritten, leading to nested case function applications. Furthermore, we add the rewrite rule $C_D(e, \overline{c}_{f_1}, \dots, \overline{c}_{f_{|\mathcal{C}_\mathcal{S}(D)|}}) = e$ to aid simplification. Using the distribution laws, the surrounding determinizer/projection functions can often be eliminated.

Example 7. We revisit the expression $det_{List(\mathbb{N})}(plusone(C_{List(\mathbb{N})}(e, \texttt{[]}, s_1 \triangleright s_2)))$ obtained from unfolding in Example 6. Applying Lemma 1 on *plusone*, we obtain the following expression that can be rewritten to just e.

$$\begin{aligned}
& det_{List(\mathbb{N})}(C_{List(\mathbb{N})}(det_{List(\mathbb{N})}(C_{List(\mathbb{N})}(e, \texttt{[]}, s_1 \triangleright s_2)), \\
& \quad plusone(\texttt{[]}), \\
& \quad plusone(\pi^1_{List(\mathbb{N})}(C_{List(\mathbb{N})}(e, \texttt{[]}, s_1 \triangleright s_2)) \triangleright \pi^2_{List(\mathbb{N})}(C_{List(\mathbb{N})}(e, \texttt{[]}, s_1 \triangleright s_2)))))
\end{aligned}$$

6 Experiments

The original parameter unfolding technique from [6] has been available in the tool `lpsparunfold` in the mCRL2 toolset [3] for over a decade. We have extended the C++ implementation with the ideas described. The tool allows selecting

Table 1. Experimental results for symbolic reachability, reporting size of the underlying labeled transition system, and the mean total time of each of the tool executions out of 10 runs.

Models	Sizes (# states)				Times (s)			
	`standard`	`original`	`new.def`	`new.alt`	`standard`	`original`	`new.def`	`new.alt`
cylinder	1 593 209	1 593 209	1 593 209	1 593 209	30.2	16.8	16.9	17.4
fourinarow3-4	12 305	12 305	12 305	12 305	63.9	59.2	1.5	1.5
fourinarow3-5	t-o	171 243	171 243	171 243	t-o	1 437.6	10.5	10.5
fourinarow4-3	6 214	6 214	6 214	6 214	15.3	32.3	1.0	1.0
fourinarow4-4	t-o	187 928	187 928	187 928	t-o	1 790.0	10.6	10.4
fourinarow4-5	t-o	t-o	5 464 759	5 464 759	t-o	t-o	350.5	350.1
fourinarow5-3	44 131	44 131	44 131	44 131	832.8	400.7	3.4	3.3
fourinarow5-4	t-o	t-o	2 788 682	2 788 682	t-o	t-o	166.0	164.9
fourinarow5-5	t-o	t-o	t-o	t-o	t-o	t-o	t-o	t-o
onoff	t-o	t-o	t-o	3	t-o	t-o	t-o	0.1
sla7	7 918	7 918	7 918	7 918	2.0	2.5	2.6	2.4
sla10	238 931	238 931	238 931	238 931	31.9	19.1	18.8	15.6
sla13	t-o	6 693 054	6 693 054	6 693 054	t-o	432.7	418.3	324.2
swp2-2	14 064	14 064	14 064	14 064	1.2	1.2	1.2	1.2
swp2-4	140 352	140 352	140 352	140 352	2.4	2.5	2.4	2.4
swp2-6	598 320	598 320	598 320	598 320	3.2	3.5	3.1	3.2
swp2-8	1 731 840	1 731 840	1 731 840	1 731 840	4.1	4.8	3.9	4.0
swp4-2	2 589 056	2 589 056	2 589 056	2 589 056	5.9	9.8	7.4	7.3
swp4-4	292 878 336	292 878 336	292 878 336	292 878 336	132.4	173.6	110.4	111.7
swp4-6	5 729 304 960	5 729 304 960	5 729 304 960	5 729 304 960	3 072.5	1 146.3	716.9	725.5
swp4-8	t-o	t-o	50 128 191 488	50 128 191 488	t-o	t-o	2 968.2	3 010.4
swp8-2	t-o	t-o	t-o	t-o	t-o	t-o	t-o	t-o
tictactoe3-3	5 479	5 479	5 479	5 479	14.1	9.6	1.5	1.5
wms	155 034 776	155 034 776	155 034 776	155 034 776	17.0	17.0	16.8	16.2

which parameters to unfold, and the number of times that parameter should be unfolded using command-line options. Multiple parameters can be unfolded in a single run; this is achieved by iterating the unfolding of a single parameter.

To evaluate the effect of our improvements on further analysis of LPEs and the generation of the underlying state space using symbolic reachability, we compare the following workflows:

- `standard`: standard static analysis workflow: instantiate finite summations, eliminate constant and redundant parameters and superfluous summation variables [6] (tools `lpssuminst`, `lpsconstelm`, `lpsparelm` and `lpssumelm`). Finally, perform symbolic reachability (`lpsreach`). No parameter unfolding.
- `original`: before `standard`, perform the original parameter unfolding.
- `new.def`: before `standard`, perform parameter unfolding with our extension for global variables and pattern matching rules with default case placement.
- `new.alt`: same as `new.def` but use alternative case placement.

The workflows are executed on various models translated to mCRL2, including our running example (onoff). Models of two-player games, often used to teach formal methods: *four-in-a-row*, with varying numbers of rows and columns and *tic-tac-toe* on a standard 3x3 board, in which the board is encoded using fixed length lists of lists. First, the board is unfolded, and then each of the rows resulting from this first unfolding. Models of a *sliding window protocol* [5], that forms the basis of the TCP protocol used for reliable in-order delivery of packets, as it occurs in [7], with window size n and m messages (swp-n-m) for different values of n and m. The send and receive windows are unfolded. Moreover, we include models based on industrial applications: a UML state machine diagram of an industrial pneumatic cylinder (cylinder) [15]; the protocol negotiating a *service level agreement* (sla) between two parties communicating via message passing along reliable channels encoded using fixed length lists [9]; and a model of the Workload Management System (wms) of the DIRAC Community Grid Solution for the LHCb experiment at CERN [14]. Note that the use of complex data structures for industrial case studies is wide-spread, allowing the creation of concise and elegant models.

All experiments were run 10 times, on a machine with 4 Intel 6136 CPUs and 3TB of RAM, running Ubuntu 20.04. A reproduction package is available from `https://github.com/astramaglia/lpsparunfold-experiments`. The results are presented in Table 1. We used a time-out of 1 hour (3600 seconds), and a memory limit of 64GB. We report the size of the explored state space (in number of states, or 't-o' in case of a time-out) and the mean total running time of ten runs in seconds. For most of the experiments, the standard deviation is below 10% of the mean.[2] If workflows result in the same state space for a model, we report the size only once in the table.

The experiments show that our improvements typically reduce the total running time of the verification. In particular, our extension for global variables reduces the running time for *four-in-a-row* and *tic-tac-toe*. The simplifications for pattern matching rules show a reduction in the running time for the sliding window protocol (swp). Alternative case placement reduces the infinite state space of our running example (onoff) to only three states; for the service-level-agreement protocol (sla) it reduces the total running time.

Even when the size of the state space is not changed, our improvements often reduce the running time of symbolic reachability. This is due to the simplification of data in the processes, and the reduction of dependencies between process parameters. Although in theory alternative case placement could lead to an exponential blow-up of the expressions in the LPE, this is not observed in our experiments.

7 Conclusion

In this paper we described and revisited the static analysis technique for flattening the structure of process parameters in LPEs, in the context of mCRL2. The

[2] The SDs for the only cases where it exceeds 10% of the mean are: *fourinarow4-3* `standard` 1.7, *sla7* `new.def`: 0.3, *tictactoe3-3* `standard`: 2.0, *wms* `standard`: 2.5, `original`: 2.2, `new.def`: 1.9

extensions improve the effectiveness of parameter unfolding in two ways. First, it improves the effectiveness of other static analysis tools. Our experiments show that this can result in large reductions of the underlying state space, directly improving explicit-state model checking. Second, for symbolic model checking, and symbolic reachability in particular, our improvements reduce the execution times even if the size of the state space is not reduced.

We believe the effect of `lpsparunfold` should be investigated in relation to other static analysis techniques such as dead variable analysis. Together these have the potential to speed up the model checking of industrial systems, e.g., described by OIL models [4] and Cordis models [15] using mCRL2. Furthermore, the effect of `lpsparunfold` could be investigated in the context of PBESs.

Acknowledgements Michel Reniers and Frank Stappers previously described Groote and Lisser's original definition of parameter unfolding in an unpublished note. Some of our notation is inspired by their note.

References

1. Blom, S., Fokkink, W., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: µCRL: A toolset for analysing algebraic specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. pp. 250–254. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_23
2. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer (2008). https://doi.org/10.1007/978-3-540-85762-4_6
3. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E., Wesselink, J.W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
4. Bunte, O., van Gool, L.C.M., Willemse, T.A.C.: Formal verification of OIL component specifications using mCRL2. In: ter Beek, M.H., Ničković, D. (eds.) Formal Methods for Industrial Critical Systems. pp. 231–251. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_10
5. Cerf, V., Kahn, R.: A protocol for packet network intercommunication. IEEE Transactions on Communications **22**(5), 637–648 (1974). https://doi.org/10.1109/TCOM.1974.1092259
6. Groote, J.F., Lisser, B.: Computer assisted manipulation of algebraic process specifications. Tech. Rep. SEN-R0117, CWI (Jan 2001), `https://ir.cwi.nl/pub/4326/`
7. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. The MIT Press (2014)
8. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theoretical Computer Science **343**(3), 332–369 (2005). https://doi.org/10.1016/j.tcs.2005.06.016
9. Groote, J.F., Willemse, T.A.C.: A symmetric protocol to establish service level agreements. Log. Methods Comput. Sci. **16**(3) (2020). https://doi.org/10.23638/LMCS-16(3:19)2020

10. Keiren, J.J.A., Wesselink, W., Willemse, T.A.C.: Liveness Analysis for Parameterised Boolean Equation Systems. In: ATVA 2014. LNCS, vol. 8837, pp. 219–234 (2014). https://doi.org/10.1007/978-3-319-11936-6_16
11. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing. pp. 204–219. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014)
12. Neele, T.: (Re)moving Quantifiers to Simplify Parameterised Boolean Equation Systems. In: ARQNL 2022. vol. 3326, pp. 64–80. CEUR-WS (2022)
13. Orzan, S., Wesselink, W., Willemse, T.A.C.: Static Analysis Techniques for Parameterised Boolean Equation Systems. In: TACAS 2009. LNCS, vol. 5505, pp. 230–245 (2009). https://doi.org/10.1007/978-3-642-00768-2_22
14. Remenska, D., Willemse, T.A.C., Verstoep, K., Templon, J., Bal, H.: Using model checking to analyze the system behavior of the LHC production grid. Future Generation Computer Systems **29**(8), 2239–2251 (2013). https://doi.org/10.1016/j.future.2013.06.004
15. Stramaglia, A., Keiren, J.J.A.: Formal verification of an industrial UML-like model using mCRL2. In: Groote, J.F., Huisman, M. (eds.) FMICS 2022. pp. 86–102. LNCS, Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_7
16. van de Pol, J., Timmer, M.: State Space Reduction of Linear Processes Using Control Flow Reconstruction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. pp. 54–68. LNCS, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_5
17. Van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int J Softw Tools Technol Transfer **19**(6), 675–696 (Nov 2017). https://doi.org/10.1007/s10009-016-0433-2