Theoretical Computer Science 1038 (2025) 115181

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Unfolding state variables improves model checking performance ☆

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

Eindhoven University of Technology, the Netherlands

ARTICLE INFO

Keywords:
Model checking
Algebraic data types
Static analysis
Process algebra

ABSTRACT

When describing the behavior of systems, state variables are typically modeled using complex data types. This use of data types allows for concise models that are easy to read. However, model checking tools that aim to automatically establish the correctness of such models use static analyses of state variables to improve their performance. Therefore, the use of complex data types in behavioral models negatively affects the performance of model checking tools. To address this, in this article we revisit a technique by Groote and Lisser that can be used to replace a single state variable of a complex data type by multiple state variables of simpler data types. We introduce and study several extensions in the context of the process algebraic specification language mCRL2, and establish their correctness. We demonstrate that our technique typically reduces the verification times when using symbolic model checking, and show that sometimes it enables static analysis to reduce the underlying state space from infinite to finite.

1. Introduction

Most modern software is inherently concurrent. Concurrent systems consist of components that perform local computations, and that use protocols to communicate (or interact) with other components and the environment. As users, we expect the software to work correctly in all circumstances. However, in practice, this is generally not the case. This is due to the inherent difficulty in the development of concurrent systems: corner cases are easily overlooked, resulting in subtle errors during the use of the software.

Several solutions have been developed to improve the quality of software. One can, for instance, prove the correctness of software using techniques such as Hoare logic [1], separation logic [2,3] and process algebra [4]. These typically involve a significant manual verification effort. Data-flow analysis (see, e.g., [5]) can be used as a fully automated, abstract, but imprecise interpretation of programs. Model checking [6,7] aims to provide a precise, fully automated analysis of a (model of) a program.

Although there are model checkers that directly deal with implementations in high-level programming languages such as C or C++, e.g., Spin [8], DIVINE 4 [9], and LLBMC [10], most model checkers use abstract models of concurrent systems. Model checking abstract models is one of the few instruments that can be used to formally verify designs when the code is not (yet) available. Examples of model checkers that use this approach are CADP [11], Dezyne [12], FDR [13], and mCRL2 [14]. The modeling languages of these tools differ, but in essence, all of these tools describe states of the model using state variables. These state variables have means to describe the states of the model, in the form of variable declarations, and a way to describe transitions between states. State variables are used in expressions that appear, e.g., in conditions that control whether a given transition is enabled, as parameters to actions that label a transition, and they can be assigned a new value to describe the effect of a transition.

☆ This article belongs to Section B: Logic, semantics and theory of programming, Edited by Don Sannella.

* Corresponding author.
E-mail addresses: a.stramaglia@tue.nl (A. Stramaglia), j.j.a.keiren@tue.nl (J.J.A. Keiren), t.s.neele@tue.nl (T. Neele).

https://doi.org/10.1016/j.tcs.2025.115181
Received 12 July 2024; Received in revised form 14 February 2025; Accepted 10 March 2025

Available online 13 March 2025

Model checking suffers from the infamous state space explosion problem [15]. There are two key contributing factors to the large number of states in the state space of a system. First, concurrency results in the exponential growth of the state space. For instance, a system consisting of three components of ten states each can potentially be in $10^3 = 1000$ different states. The second factor is the use of data in the model. For instance, a controller that tracks n bits of information can already be in 2^n different states, due to the data alone. Together these result in state spaces easily exceeding 10^{100} states in practice [16].

Many techniques have been developed to counteract the state space explosion problem [17]. For instance, partial-order reduction [18,19] reduces the number of different interleavings that must be considered. Symbolic model checking [20,21] uses symbolic representations such as binary decision diagrams to store states and transition relations. Most model checkers furthermore use static analysis techniques such as constant propagation and dead variable analysis to reduce the data [22,23].

Both static analysis and modern symbolic model checkers such as LTSmin [24] analyze state variables and their dependencies. As such, they benefit from models in which state variables are fine grained. On the other hand, to facilitate modeling of realistic systems, modeling languages often allow the use of data types such as structures, records and lists. Their use leads to specifications that are easy to construct and understand for the modeler.

Contributions In this paper, we describe a general approach that allows to *unfold* state variables in a specification of a distributed system, a basic version of which was described by Groote and Lisser [22]. We assume that the description of the behavior uses state variables and updates their value. We use algebraic data types to characterize the data. By unfolding state variables, we retain the possibility for the modeler to construct high-level specifications, while also automatically generating a fine-grained model that is more amenable to static analysis and symbolic model checking.

More concretely, our approach consists of the following steps:

- Replacing a single state variable s by a number of variables $s_1, \ldots, s_n$.
- Replacing a term using state variable s by an equivalent term using the newly introduced state variables $s_1, \ldots, s_n$. We describe and compare two alternatives for this, referred to as *case placement* and *alternative case placement*.
- Replacing an update of a single state variable s by the corresponding updates to the new state variables $s_1, \ldots, s_n$. We simplify complex state updates by locally eliminating functions that are defined using *pattern matching*. We refer to this as *pattern match unfolding*.
- Extending the algebraic data types with the functions needed to facilitate these replacements.

To study the effect of the unfolding of state variables, we consider the mCRL2 language [25]. This is a process algebraic specification language where processes can be parameterized with data specified using algebraic data types. The language comes with an associated toolset to model, validate and verify complex systems [14]. Models in mCRL2 consist of a number of (communicating) parallel processes that are parameterized with data. As preprocessing for further analysis, the mCRL2 toolset transforms specifications 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 variables, and a number of condition-action-effect rules referred to as *summands*, in which the variables are used and updated in a manner that closely matches the previous high-level description.

Prior to the research presented in this article, the tool `lpsparunfold` in the mCRL2 toolset already implemented Groote and Lisser's parameter unfolding [22]. We have extended this implementation with alternative case placement and pattern match unfolding. In addition, mCRL2 allows the use of global variables; we extend the unfolding technique and its implementation to take into account such global variables. We prove that each of the transformations preserves strong bisimilarity of LPEs. This establishes correctness. Using experiments, we show that parameter unfolding typically speeds up symbolic model checking. Pattern match unfolding and the unfolding of global variables typically have a positive effect on the performance. Although theoretically alternative case placement can lead to an exponential blow-up of terms to which it is applied, this effect is not observed in our experiments: the performance of case placement and alternative case placement is comparable most of the time. On our running example alternative case placement is essential in order to transform an infinite state space into an equivalent but finite one.

This article is an extended version of [26]. Compared to [26] we have separated the presentation of the unfolding of state variables from the setting of mCRL2, emphasizing that this is a technique that is more generally applicable. Equivalence of terms and their unfolding is proven in this general setting. For correctness of the unfolding in the setting of mCRL2, we include a detailed discussion of the unfolding using global variables. Moreover, we present full proofs that show that strong bisimulation is preserved. We have also extended the experiments and present the results in more detail.

Related work The algebraic data types used in mCRL2 [25] and assumed in this article, have a *model-class semantics* [27]. In this semantic approach, the class consists of all algebras that satisfy the axioms, in contrast to the more common initial algebra semantics that is restricted to the isomorphism class of initial algebras. This approach is sometimes referred to as *loose semantics* [28,29]. Without further assumptions, the loose semantics allows for certain degenerate models, such as the one where every element is mapped to the same value in the domain. Such degenerate models can be excluded by taking the *non-degenerate loose semantics*, in which models satisfying $true = false$ are excluded. We refer to [30] for an accessible introduction to algebraic data types.

Our unfolding of state variables is most closely related to various analysis and transformation techniques for LPEs that have been developed in the setting of μCRL [31] and mCRL2 [14] over the years. Groote and Lisser [22] introduced static analysis for μCRL specifications, including a technique for flattening the structure of process parameters and implemented these in μCRL [31].

The latter technique is the core of our unfolding of state variables. However, in [22] alternatives for reconstructing parameters and pattern match unfolding are not considered, and global variables are not taken into account. Furthermore, no correctness proof is presented. A more advanced algorithm is *liveness analysis* [23], 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 analysis and transformation techniques have been developed for *Parameterized Boolean Equation Systems* (PBES) [32]. For example, redundant and constant parameter elimination for PBES is presented in [33], liveness analysis in [34]; a generalization of constant elimination occurs in [35].

The use of data flow analysis techniques to reduce the state space or improve the performance of model checking is not limited to mCRL2. For instance, manually resetting variables when they are no longer needed is supported through a dedicated keyword (`clear`) by Murphi [36]. Automated dead variable analysis has been studied for model checkers such as CADP [37] and UPPAAL [38]. Data flow analysis has also been studied for probabilistic models [39]. All of these analyses potentially benefit from a more fine-grained representation of variables, e.g., when only part of a more complex variable is dead.

Parameter unfolding could be beneficial for other techniques used in model checking as well. For instance, symmetry reduction [40], which is implemented in model checkers such as FDR [41], depends on an analysis of shared variables. Unfolding parameters can lead to more fine-grained information regarding such shared variables. Furthermore, symbolic model checkers use representations such as *list decision diagrams* (LDDs, a generalization of *binary decision diagrams*) [42], in which each variable is represented by a layer in the LDD. Parameter unfolding could change the LDD structure by having multiple simpler layers instead of a single more complex layer. This potentially reduces the size of the LDD representation. The implementation of symbolic reachability used in our experiments is based on the techniques from [43,44], and uses the list decision diagrams from Sylvan [45].

Instead of using data flow analysis to improve model checking, model checking has also been used to perform data flow analysis. For instance, Steffen uses a model checker to compute optimal placement of computations within a program [46]. Del Mar Gallardo et al. used model checkers as generic, on-the-fly data flow analyzers [47]. Data flow analysis for programming languages in general has been studied extensively in the literature. We refer to standard textbooks such as [5] for an in-depth description of data flow analysis of programming languages.

Structure Section 2 introduces a running example that is used throughout the paper. Next, an introduction to algebraic data types is provided in Section 3. In Section 4 we introduce the unfolding of state variables, and describe alternative case placement and pattern match unfolding. We describe how this can be used in mCRL2 in Section 5, and prove that unfolding preserves strong bisimilarity. Finally, we evaluate the approach using experiments in Section 6 and conclude in Section 7.

2. Motivating example

We first present a motivating example. To facilitate consistent use of syntax throughout the paper, we present the motivating example using the mCRL2 specification language. The techniques introduced in this paper are, however, generally applicable to specification languages that: (1) use algebraic data types for the specification of the data used, and (2) *declare* state variables, *use* them in terms, and *update* their value. Note that mCRL2 has standard data types for, e.g., Booleans and numeric data types. To present the motivating example independently from mCRL2, we here choose to give our own specification of all the relevant data types. In Section 5 we update the example to instead use the full power of mCRL2.

Our motivating example is a specification of a simple system inspired by the mCRL2 models generated from Open Interaction Language (OIL) specifications [48]. It describes a system that starts out uninitialized. If it is uninitialized, it can be initialized using a transition labelled *initialize*. The initialized system can be in either of two states: *off* or *on*, and can be toggled between these two states. Moreover, the initialized system has an IP address, which we model abstractly as a natural number. The IP address is only relevant when the state is *on*, and whenever the system switches from *off* to *on*, it gets assigned an arbitrary number as IP address.

$$
\begin{array}{ll}
\textbf{sort} & B; \\
\textbf{cons} & \mathit{true}, \mathit{false} : B; \\
\textbf{map} & \approx, \not\approx : B \times B \to B; \\
 & \neg : B \to B; \\
 & \wedge, \vee : B \times B \to B;
\end{array}
\qquad
\begin{array}{ll}
\textbf{var} & x, y : B; \\
\textbf{eqn} & x \approx x = \mathit{true}; \\
 & \mathit{true} \approx \mathit{false} = \mathit{false}; \\
 & \mathit{false} \approx \mathit{true} = \mathit{false}; \\
 & x \not\approx y = \neg(x \approx y); \\
 & \neg \mathit{true} = \mathit{false}; \\
 & \neg \mathit{false} = \mathit{true}; \\
 & x \wedge \mathit{true} = x; \\
 & \mathit{true} \wedge x = x; \\
 & x \wedge \mathit{false} = \mathit{false}; \\
 & \mathit{false} \wedge x = \mathit{false}; \\
 & x \vee \mathit{true} = \mathit{true}; \\
 & \mathit{true} \vee x = \mathit{true}; \\
 & x \vee \mathit{false} = x; \\
 & \mathit{false} \vee x = x;
\end{array}
\qquad
\begin{array}{ll}
\textbf{sort} & N; \\
\textbf{cons} & \mathit{zero} : N; \\
 & \mathit{succ} : N \to N; \\
\textbf{map} & \approx, \not\approx : N \times N \to B; \\
 & + : N \times N \to N; \\
\textbf{var} & n, m : N; \\
\textbf{eqn} & \mathit{zero} \approx \mathit{zero} = \mathit{true}; \\
 & \mathit{zero} \approx \mathit{succ}(n) = \mathit{false}; \\
 & \mathit{succ}(n) \approx \mathit{zero} = \mathit{false}; \\
 & \mathit{succ}(n) \approx \mathit{succ}(m) = n \approx m; \\
 & n \not\approx m = \neg(n \approx m); \\
 & \mathit{zero} + n = n; \\
 & n + \mathit{zero} = n; \\
 & \mathit{succ}(n) + m = \mathit{succ}(n + m); \\
 & n + \mathit{succ}(m) = \mathit{succ}(n + m);
\end{array}
$$

Fig. 1. Specification of the data types for Booleans and natural numbers.

sort	$State$;	**sort**	Sys;	**var**	$s, t : Sys; p_1, p_2 : State; n, m : N$;
cons	$p\_on, p\_off : State$;	**cons**	$uninit : Sys$;	**eqn**	$s \approx s = true$;
map	$\approx, \not\approx : State \times State \to B$;		$sys : State \times N \to Sys$;		$uninit \approx sys(p_1, n) = false$;
var	$x, y : State$;	**map**	$\approx, \not\approx : Sys \times Sys \to B$;		$sys(p_1, n) \approx uninit = false$;
eqn	$x \approx x = true$;		$get\_state : Sys \to State$;		$sys(p_1, n) \approx sys(p_2, m) = p_1 \approx p_2 \wedge n \approx m$;
	$p\_on \approx p\_off = false$;		$get\_ip : Sys \to N$;		$s \not\approx t = \neg(s \approx t)$;
	$p\_off \approx p\_on = false$;		$set\_state : Sys \times State \to Sys$;		$get\_state(sys(p_1, n)) = p_1$;
	$x \not\approx y = \neg(x \approx y)$;		$set\_ip : Sys \times N \to Sys$;		$get\_ip(sys(p_1, n)) = n$;
					$set\_state(sys(p_1, n), p_2) = sys(p_2, n)$;
					$set\_ip(sys(p_1, n), m) = sys(p_1, m)$;

Fig. 2. Specification of the data types *State* and *Sys*.

The specification uses four data types, see Figs. 1 and 2. The Booleans are described using sort B with constructors *true* and *false*. Standard operations such as equality $\approx$, negation $\neg$ as well as conjunction and disjunction $\wedge/\vee$ are defined. The operations are defined in an equational manner. In a similar way, natural numbers, represented using sort N, can be defined using *zero* and successor (*succ*). We restrict the definitions to the operators used in our specifications, and we illustrate the definition of $+$. They can be extended with additional operations such as multiplication in the obvious way.

The sort *State* represents the status of the system which can be set to *p_on* or *p_off*, see Fig. 2. They are defined to be distinct using a definition of $\approx$.

Finally, sort *Sys* has two constructors, $uninit : Sys$ and $sys : State \times N \to Sys$. For this, operations such as equality ($\approx$) and inequality ($\not\approx$) are defined, to ensure that, e.g., $sys(p, n) \not\approx uninit$ for all $p : State$, $n : N$. Also, the projection functions $get\_state : Sys \to State$ and $get\_ip : Sys \to N$ are defined such that, $get\_state(sys(p, n)) = p$ and $get\_ip(sys(p, n)) = n$. Similarly, we define functions *set_state* and *set_ip* to set the state and IP address. Note that these four functions are partially defined.

The behavior of our example is defined abstractly as a *process* P, parameterized with a single state variable s of sort *Sys*. The definition uses actions *on*, *off*, and *initialize*. The behavior is defined using a set of (recursive) condition-action-effect rules. A condition-action-effect rule is of the shape '(condition) $\to$ action-effect' which can be read as 'if condition is true then do action and update the state with effect'. The operator $+$ denotes a nondeterministic choice among the different rules. Operator $\sum$, parameterized with a local variable, denotes a generalized nondeterministic choice over rules parameterized with that variable.

$$
\begin{aligned}
P(s : Sys) = {} & (s \approx uninit) \to initialize \cdot P(sys(p\_off, zero)) \\
& + \sum_{n : N} (s \not\approx uninit \wedge get\_state(s) \approx p\_off) \to on \cdot P(set\_state(set\_ip(s, n), p\_on)) \\
& + (s \not\approx uninit \wedge get\_state(s) \approx p\_on) \to off \cdot P(set\_state(set\_ip(s, zero), p\_off))
\end{aligned}
$$

The process P describes that when the system is uninitialized, captured using condition $s \approx uninit$, a transition labelled with action *initialize* is taken, and the value of variable s is updated to be $sys(p\_off, zero)$. For any natural number n, when the system is *off*, denoted by $s \not\approx uninit \wedge get\_state(s) \approx p\_off$, the transition labelled *on* can be taken, and in the next state, the IP address component of the state becomes n, and the state-component becomes *p_on*; this is denoted using $set\_state(set\_ip(s, n), p\_on)$. When the system is *on*, it can take a transition labelled *off*, and similar to the previous case, the IP address is set to *zero*, and the state component is updated to *p_off*.

Note that equivalently, in the second condition-action-effect rule, we could have set s in the next state to $sys(p\_on, n)$. In the same rule, the use of $\sum_{n : N}$ is shorthand for the following nondeterministic choice between infinitely many transitions:

$$
\begin{aligned}
& + (s \not\approx uninit \wedge get\_state(s) \approx p\_off) \to on \cdot P(set\_state(set\_ip(s, zero), p\_on)) \\
& + (s \not\approx uninit \wedge get\_state(s) \approx p\_off) \to on \cdot P(set\_state(set\_ip(s, succ(zero)), p\_on)) \\
& + \ldots
\end{aligned}
$$

The above process serves as a compact description of a *labelled transition system* (LTS). The LTS for $P(uninit)$ is shown in Fig. 3. Note that this LTS 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 when it is *on*, *i.e.*, it is in a state $sys(p\_on, n)$, is bisimilar for all values of n. Since the state contained in state variable s is used in the process, for instance to determine whether a transition is enabled, static analysis techniques that consider s as a single entity are not able to simplify the description. Yet, intuitively, as the IP address contained in s is not used significantly, it is desirable for static analysis to detect this, and remove this component altogether, leading to a finite, bisimilar description. To enable such static analyses, it is beneficial to split parameter s into multiple parameters, e.g., e_s, denoting whether the value of s is *uninit* or $sys(p, n)$ for some p and n, encoded as $\overline{c}_{uninit}$ and $\overline{c}_{sys}$, respectively, and parameters s^1_{sys} and s^2_{sys} storing the parameters p and n in case s is $sys(p, n)$. Such a process could look as follows. Note that we omit the details of the data types. These are discussed in more detail later in the paper.

$$
\begin{aligned}
P(e_s : U_{Sys}, s^1_{sys} : State, s^2_{sys} : N) = {} & (e_s \approx \overline{c}_{uninit}) \to initialize \cdot P(\overline{c}_{sys}, p\_off, zero) \\
& + \sum_{n : N} (e_s \not\approx \overline{c}_{uninit} \wedge p \approx p\_off) \to on \cdot P(e_s, p\_on, n) \\
& + (e_s \not\approx \overline{c}_{uninit} \wedge p \approx p\_on) \to off \cdot P(e_s, p\_off, zero)
\end{aligned}
$$

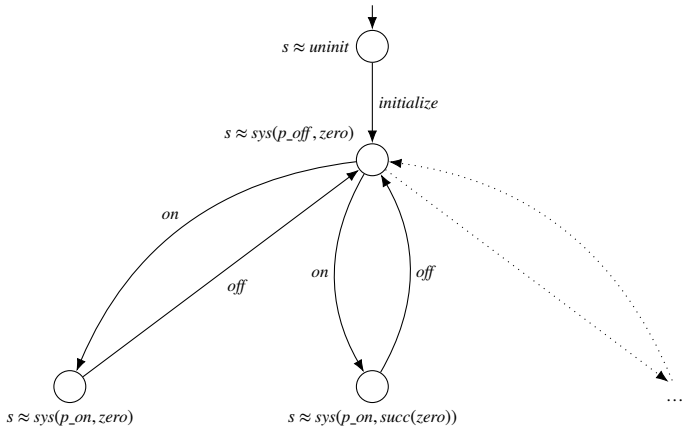

Fig. 3. LTS for process $P(init)$ of the running example.

The transformation we present in this paper produces a description similar to the process described above, obtained from our running example. The reader should note that s^2_{xys} is not used significantly in this description, so it (and as result also the locally bound variable n) can be removed using for instance the parameter elimination technique from [22]. This reduced description has an underlying LTS of only 3 states.

3. Algebraic data types

In this paper we work in a setting where data is defined using algebraic data types. We give a brief overview of the concepts relevant to this paper. A good textbook introduction to algebraic data types can be found in [30]. For detailed definitions of the data types used in mCRL2 we refer to [25].

We use many-sorted algebras to allow for the definition of several *sorts* of data. A *signature* is a triple $\Sigma = (S, C_S, \mathcal{M}_S)$ where S is the set of sorts, C_S and $\mathcal{M}_S$ are disjoint sets of function symbols over S, called *value constructors*, and *mappings*, respectively. We typically write *constructors* instead of value constructors. The set of sorts S consists of sort names and function sorts. Function sorts are of the form $D_0 \times \cdots \times D_n \to D$, for $D_i, D \in S$ for $0 \leq i \leq n$; sorts that are not function sorts are sort names. If $D = D_1 \times \cdots \times D_n \to D'$ we write $range(D)$ for its range D'. Function symbols in $C_S \cup \mathcal{M}_S$ are of the form $f : D_1 \times \cdots \times D_n \to D$. If $n = 0$, we say f is a constant. We assume every signature has a sort name B representing the Booleans.

Although they are syntactically not distinguished, the role of constructors and mappings is subtly different. Constructors are used to inductively define the elements of a sort, and introduce a means for pattern matching. Mappings define any other operation on an algebraic data type. As such, constructors play a crucial role when unfolding state variables in the technique that we propose. Note that not every sort is defined using constructors, the real numbers are an example of such a sort. We write $C_S(D) = \{f : D_1 \times \cdots \times D_n \to D' \in C_S \mid D' = D\}$ for the constructors of sort D. We assume a bijection ι_D between $C_S(D)$ and $0..|C_S(D)| - 1$ ordering the constructors, and write ι if D is clear from the context. For our examples we assume that ι is consistent with the order in which the constructors appear in the specification, and we leave its definition implicit. We say that D is a *constructor sort* if, and only if, $C_S(D) \neq \emptyset$. A constructor sort D is syntactically non-empty if there is a constructor $f : D_1 \times \cdots \times D_n \to D$ 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 syntactically non-empty, and for $f : D \in C_S$, $range(D)$ must not be a function sort. With every constructor sort D, we associate a unique default term, def_D. Such a term exists due to syntactic non-emptiness.

Example 1. The sort B, representing the Booleans, from the previous section has two constructors, *true* and *false* that together allow us to describe all Booleans. Formally $C_S(B) = \{true : B, false : B\}$. Likewise, sort N with constructors *zero* and *succ* allows to describe all natural numbers. Similar as before, we have $C_S(N) = \{zero : N, succ : N \to N\}$. Note that sorts B and N are sort names, and sort $N \to N$ is a function sort. Both B and N are constructor sorts.

Given a set $\mathcal{X}_S$ of S-sorted variables, where $x \in \mathcal{X}_S$ for $S \in S$ denotes that x is a variable of sort S, we can construct *terms*. Terms are syntactically described by the following grammar:

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

where $x \in \mathcal{X}_S$ are variables, $f \in C_S \cup \mathcal{M}_S$ are sorted function symbols, where we sometimes write $f : D_1 \times \cdots \times D_n \to D \in C_S$ if the sort of f is important, and $t(t, \ldots, t)$ describes the application of a term to its arguments. For term $t(t_1, \ldots, t_n)$, t is the head term and $t_1, \ldots, t_n$ are the arguments; if t is a function symbol, we typically refer to it as the head symbol. We use $fv(t)$ to denote the set of variables occurring in t, and we write $e[x := e']$ for the syntactic substitution of x with e' in e.

Equality of terms is defined using an *equational specification* $\mathcal{D} = (\Sigma, E)$, where Σ is a signature and E is a set of conditional equations of the form $\langle c \to t = u \rangle$, where c, t, u are terms over $\mathcal{X}_S$. We typically write $\langle t = u \rangle$, when $c = \mathit{true}$. Note that the mCRL2 toolset uses term rewriting, interpreting the equations in a strictly left to right fashion, to simplify terms.

The core ideas of our technique are independent of the precise semantics of the data types. For the sake of conciseness we use the *model class semantics* of the data types in mCRL2 [25]. The results carry over straightforwardly when using different semantics.

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, $M_B = \{\mathbf{true}, \mathbf{false}\}$, 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 equational specification into its semantic counterpart, that is, for all $f \in \mathcal{C}_S \cup \mathcal{M}_S$ of sort D, $[\![f]\!] \in M_D$. This is generalized to arbitrary terms as follows:

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

where $\sigma : \mathcal{X}_S \to \bigcup_{D \in \mathcal{S}} M_D$ is a valuation that ensures that $\sigma(x) \in M_D$ for all $x : D$. We write $\sigma[v/d]$ for the valuation that assigns v to d and otherwise behaves as σ. The model $\mathbb{M}$ of a equational 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 $f \in \mathcal{C}_S$ 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.

In the remainder, we use some (standard) properties of the semantics of algebraic data types. The first property states that the valuation of variables that do not appear in a term do not affect the semantics of the term.

Lemma 1. *For all terms t, and variables y such that $y \notin \mathit{fv}(t)$, for all values v, and valuations σ*

$$[\![t]\!]^{\sigma[v/y]} = [\![t]\!]^\sigma.$$

In case t is closed, that is $\mathit{fv}(t) = \emptyset$, we have $[\![t]\!]^\sigma = [\![t]\!]^{\sigma'}$ for all σ, σ', and we sometimes write $[\![t]\!]$.

Also, syntactic substitutions can be moved into the valuation by evaluating the right hand side in the context of the same valuation.

Lemma 2. *For all terms t and e, variables d and valuations σ*

$$[\![t[d := e]]\!]^\sigma = [\![t]\!]^{\sigma[[\![e]\!]^\sigma/d]}.$$

Finally, we remark on the fact that, if D is a constructor sort, every term t of sort D can be written in terms of a constructor application.

Lemma 3. *Let D be a constructor sort. Then for every term t of sort D, and valuation σ, we have*

$$[\![t]\!]^\sigma = [\![f_i(x_i, \ldots, x_{m_i})]\!]^{\sigma[v_1/x_i, \ldots, v_{m_i}/x_{m_i}]}$$

for some constructor $f_i : D_1 \times \cdots \times D_{m_i} \to D \in \mathcal{C}_S$, variables x_i of sort D_i and $v_i \in M_{D_i}$.

Proof. Fix D, t and σ as above. Note that $[\![t]\!]^\sigma = v$ for some $v \in M_D$. As D is a constructor sort, v is a constructor element, hence a constructor function $f_i \in \mathcal{C}_S$ exists of sort $D_1 \times \cdots \times D_{m_i} \to D$ such that $v = [\![f_i]\!](v_1, \ldots, v_{m_i})$. Now, choose $x_1, \ldots, x_{m_i} \in \mathcal{X}_S$ fresh, then according to the semantics, $[\![f_i]\!](v_1, \ldots, v_{m_i}) = [\![f_i(x_1, \ldots, x_{m_i})]\!]^{\sigma[v_1/x_1, \ldots, v_{m_i}/x_{m_i}]}$. □

4. Unfolding state variables

The unfolding of state variables was introduced in the context of μCRL by Groote and Lisser under the name `structelm` [22], and has later been implemented in the mCRL2 toolset in a tool called `lpsparunfold`. The main idea 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.

More concretely, if we look at the description of our running example, there are three different ways in which a state variable appears, and that therefore need to be taken into account when unfolding:

1. The variable is declared, and this declaration must be split into multiple declarations. This is formalized in Section 4.2.
2. A variable can be used in a term. The term must be replaced by an equivalent term using the newly declared variables instead of the variable that is replaced. This is formalized in Section 4.3.

3. A variable can be assigned to. This variable assignment must be split into assignments to the variables that it is replaced by. This is formalized in Section 4.4.

Example 2. Recall our motivating example from Section 2. The single variable s is replaced by three variables: $e_s : U_{Sys}$, $s^1_{sys} : State$ and $s^2_{sys} : N$, where e_s represents the constructor at the head of s, and s^1_{sys} and s^2_{sys} are the arguments that are used when the constructor is sys.

The occurrence of s in condition $c \approx uninit$ of the first summand (*i.e.*, the first condition-action-effect rule) can be replaced, for instance, by $if(e_s \approx \overline{c}_{uninit}, uninit, sys(s^1_{sys}, s^2_{sys}))$. Essentially, this uses e_s to determine which constructor was at the head of s, and based on that it returns the term that is equivalent to s. If e_s is $\overline{c}_{uninit}$, the result is $uninit$, otherwise the result is $sys(s^1_{sys}, s^2_{sys})$. We later generalize this idea by introducing *case functions* that facilitate reasoning about sorts with more than two constructors.

Finally, again in the first summand, the assignment of $sys(p\_off, zero)$ to s must be split into assignments of the terms $\overline{c}_{sys}$, p_off, and $zero$, to parameters e_s, s^1_{sys} and s^2_{sys}, respectively. Note that the assignments to these different parameters are independent.

To facilitate these three transformations, we first extend the specification of our algebraic data type, and subsequently use the new definitions to describe the necessary transformations.

4.1. Extending the algebraic data types

The core of our unfolding is based on Groote and Lisser's technique in [22]. In particular, the extension of the equational specification that we present here is similar to that in [22]. We first introduce the extension of data types using our running example, after which we recall the formal definitions.

When unfolding a sort D, a new equational specification is constructed that extends the equational specification 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 3. Recall the equational specification from Fig. 2. We unfold sort Sys. Note that $C_S(Sys) = \{sys : State \times N \to Sys, uninit : Sys\}$, that is it has two constructors, sys and $uninit$. The equational specification of the running example is extended with the following.

sort U_{Sys};
cons $\overline{c}_{sys}, \overline{c}_{uninit} : U_{Sys}$;
map $C_{Sys} : U_{Sys} \times Sys \times Sys \to Sys$
$det_{Sys} : Sys \to U_{Sys}$;
$\pi^1_{sys} : Sys \to State$;
$\pi^2_{sys} : Sys \to N$;
var $x, x_1, x_2 : Sys; e : U_{Sys}$;
$y_1 : State; y_2 : N$;

eqn $C_{Sys}(\overline{c}_{uninit}, x_1, x_2) = x_1$;
$C_{Sys}(\overline{c}_{sys}, x_1, x_2) = x_2$;
$C_{Sys}(e, x, x) = x$;
$det_{Sys}(uninit) = \overline{c}_{uninit}$;
$det_{Sys}(sys(y_1, y_2)) = \overline{c}_{sys}$;
$\pi^1_{sys}(uninit) = p\_on$;
$\pi^2_{sys}(uninit) = 0$;
$\pi^1_{sys}(sys(y_1, y_2)) = y_1$;
$\pi^2_{sys}(sys(y_1, y_2)) = y_2$;

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 a term of sort Sys from the unfolded parts, e.g., $C_{Sys}(\overline{c}_{sys}, uninit, sys(p\_on, 3)) = sys(p\_on, 3)$. The equation $C_{Sys}(e, x, x) = x$ 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 a term 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 a term 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.

var $x_1, x_2 : Sys; e : U_{Sys}; b : B$
eqn $\pi^1_{sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, \pi^1_{sys}(x_1), \pi^1_{sys}(x_2))$;
$\pi^1_{sys}(if(b, x_1, x_2)) = if(b, \pi^1_{sys}(x_1), \pi^1_{sys}(x_2))$;
$\pi^2_{sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, \pi^2_{sys}(x_1), \pi^2_{sys}(x_2))$;
$\pi^2_{sys}(if(b, x_1, x_2)) = if(b, \pi^2_{sys}(x_1), \pi^2_{sys}(x_2))$;
$det_{Sys}(C_{Sys}(e, x_1, x_2)) = C_{Sys}(e, det_{Sys}(x_1), det_{Sys}(x_2))$;
$det_{Sys}(if(b, x_1, x_2)) = if(b, det_{Sys}(x_1), det_{Sys}(x_2))$;

We now formally define the unfolding of a constructor sort D.

Definition 1 *(Unfolding of sort D [22]).* Fix equational specification $\mathcal{D} = (\Sigma, E)$ with signature $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$. Let $D \in \mathcal{S}$ be a constructor sort.

The unfolding of D in $\mathcal{D}$ is the equational specification $\mathcal{D}' = (\Sigma', E')$, where $\Sigma' = (\mathcal{S}', \mathcal{C}_\mathcal{S}', \mathcal{M}_\mathcal{S}')$, defined as follows.

- $\mathcal{S}' = \mathcal{S} \cup \{U_D\}$, i.e., we add a fresh constructor sort U_D.
- $\mathcal{C}_\mathcal{S}' = \mathcal{C}_\mathcal{S} \cup \{\overline{c}_f \mid f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)\}$, i.e., we add one (unique, fresh) constant constructor $\overline{c}_f$ for every constructor $f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)$.
- $\mathcal{M}_\mathcal{S}' = \mathcal{M}_\mathcal{S} \cup \{C_D : U_D \times D \times \cdots \times D \to D, det_D : D \to U_D\} \cup \Pi$, with:
  - case function $C_D : U_D \times D \times \cdots \times D \to D$ with arity $|\mathcal{C}_\mathcal{S}(D)| + 1$;
  - determinizer function $det_D : D \to U_D$ that given a term of sort D determines the constructor of sort U_D that represents its head symbol; and
  - $\Pi = \bigcup_{f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)} \Pi_f$, where for every constructor $f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)$, the set of projection functions $\Pi_f = \{\pi^i_f : D \to D_i \mid 1 \le i \le arity(f)\}$ where π^i_f obtains the i^{th} argument, given a term of sort D with head symbol f.

  We assume that the mappings added here are fresh, i.e., they do not appear in $\mathcal{M}_\mathcal{S} \cup \mathcal{C}_\mathcal{S}$.
- $E' = E \cup E_{C_D} \cup E_{det_D} \cup E_\Pi \cup E_{dist}$ are the new equations for each of the mappings, defined as follows:

$$E_{C_D} = \{C_D(\overline{c}_f, x_1, \ldots, x_{|\mathcal{C}_\mathcal{S}(D)|}) = x_{\iota(f)} \mid f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)\} \cup \{C_D(e, x, \ldots, x) = x\}$$
$$E_{det_D} = \{det_D(f(y_1, \ldots, y_n)) = \overline{c}_f \mid f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)\}$$
$$E_\Pi = \bigcup_{f : D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}(D)} E_{\Pi_f}$$
$$E_{\Pi_f} = \{\pi^i_f(f(y_1, \ldots, y_n)) = y_i \mid 1 \le i \le n\} \cup \bigcup_{g : D'_1 \times \cdots \times D'_m \to D \in \mathcal{C}_\mathcal{S}(D), g \neq f} \{\pi^i_f(g(y_1, \ldots, y_m)) = \mathsf{def}_{D_i} \mid 1 \le i \le n\}$$

So, the case function, if provided with the constructor $\overline{c}_f$ that corresponds to f, returns the argument corresponding to index $\iota(f)$. Determinizer det_D, provided with a term that has f as head symbol, returns the constructor of sort U_D used to represent this head symbol. The set E_Π gives the equations to project the arguments of f. A projection function π^i_f returns default value def_{D_i} in case it is applied to a $g \neq f$. E_{dist} is the set of distribution laws:

$$\begin{aligned}
&\{\pi^i_f(C_D(x, x_1, \ldots, x_{|\mathcal{C}_\mathcal{S}(D)|})) = C_D(x, \pi^i_f(x_1), \ldots, \pi^i_f(x_{|\mathcal{C}_\mathcal{S}(D)|})) \mid \pi^i_f \in \Pi\} \\
&\cup \{\pi^i_f(if(b, x_1, x_2)) = if(b, \pi^i_f(x_1), \pi^i_f(x_2)) \mid \pi^i_f \in \Pi\} \\
&\cup \{det_D(C_D(x, x_1, \ldots, x_{|\mathcal{C}_\mathcal{S}(D)|})) = C_D(x, det_D(x_1), \ldots, det_D(x_{|\mathcal{C}_\mathcal{S}(D)|}))\} \\
&\cup \{det_D(if(b, x_1, x_2)) = if(b, det_D(x_1), det_D(x_2))\}
\end{aligned}$$

To ensure well-typedness of the distribution laws, equations analogous to those above and case functions $C_D : U_D \times D_i \times \cdots \times D_i \to D_i$ and $C_D : U_D \times U_D \times \cdots \times U_D \to U_D$, both with arity $|\mathcal{C}_\mathcal{S}(D)| + 1$, are also added as needed.

To avoid rendering the equational specification inconsistent (that is, we should not be able to derive $\mathbf{true} = \mathbf{false}$) we need to ensure that the new equational specification is a conservative extension. This means that using equations that are added in the unfolding, we should not be able to derive any new facts about the data types in the original equational specification. We remark on two aspects of our unfolding that together ensure the unfolded equational specification is a conservative extension.

First, the unfolding of sort D does not define any additional requirements regarding (in)equality of the constructors of sort U_D. This is motivated by the following example.

Example 4. Consider sort D with constructors $f : A \to S$ and $g : S$ such that $f(a) \equiv g$ for some $a \in A$. When unfolding sort D, we introduce sort U_D with constructors $\overline{c}_f, \overline{c}_g : U_D$. Now, suppose we would require these constructors to be distinct, e.g. by adding the following equations:

$$\overline{c}_f \approx \overline{c}_g = false;$$
$$\overline{c}_g \approx \overline{c}_f = false;$$

This would make the equational specification inconsistent, as shown by the following derivation.

$$\begin{aligned}
true &= \overline{c}_f \approx \overline{c}_f \\
&= \overline{c}_f \approx det_D(f(a)) \\
&=^\dagger \overline{c}_f \approx det_D(g) \\
&= \overline{c}_f \approx \overline{c}_g \\
&= false
\end{aligned}$$

where at † we use the assumption that $f(a) \equiv g$.

This shows that we cannot reuse existing data types such as Booleans to represent sort U_D.

To obtain a conservative extension, in addition, we need to impose a mild restriction on sorts that we unfold. A sort that can be unfolded is called *unfoldable* and is defined as follows.

Definition 2. Fix equational specification $\mathcal{D} = (\Sigma, E)$ with signature $\Sigma = (S, C_S, \mathcal{M}_S)$. Sort $D \in S$ is *unfoldable* if and only if it is a constructor sort, and for all constructors $f : D_1 \times \cdots \times D_n \to D \in C_S(D)$, and terms $t_1, \ldots, t_n, t'_1, \ldots, t'_n$, if $f(t_1, \ldots, t_n) \equiv f(t'_1, \ldots, t'_n)$ then $t_i \equiv t'_i$ for all i.

In the remainder of this paper, we implicitly assume that sorts that we unfold satisfy this restriction. The following example illustrates the need for this restriction.

Example 5. Consider sort D with constructor $f : A \to D$, and terms a and b of sort A such that it does not hold that $a \equiv b$, but $f(a) \equiv f(b)$. Using the equations introduced for the projection functions, we now obtain the following: $a \equiv \pi_f(f(a)) \equiv \pi_f(f(b)) \equiv b$. The unfolding of the equational specification allows us to derive new equivalences on the original sort D, so the new equational specification is not a conservative extension.

Unfolding of an unfoldable sort D yields a conservative extension. That is, using the new equations that result from unfolding sort D, we cannot derive any new facts about the original equational specification.

Lemma 4. *Let $\mathcal{D}$ be an equational specification with unfoldable sort D, and let $\mathcal{D}'$ be the unfolding of D in $\mathcal{D}$. Then $\mathcal{D}'$ is a conservative extension of $\mathcal{D}$.*

This follows from the definitions of the new equations, and the assumption that D is unfoldable. In particular, the only way to derive new facts about the original equational specification is through the application of projection functions, in which case the assumption guarantees that these 'new' facts were already present in the original specification.

In the remainder of this section we describe the three transformations needed to achieve the unfolding of process parameters.

4.2. Splitting variable declarations

When unfolding a state variable $d : D$ of unfoldable sort D, its declaration is split into a declaration $e_d : U_D$, capturing which constructor of sort D was applied, and for every constructor f_i of sort D, declarations of state variables for each of the parameters of f_i. This is defined using $\mathsf{params}(d)$. Its definition uses $\mathsf{params}(d, f_i)$ to introduce variables for the arguments of constructor f_i. This idea was described in [22]; we here formalize the idea.

Definition 3. Let $d : D$ be a variable of constructor sort D.

- Let $f_i : D_i^1 \times \cdots \times D_i^{m_i} \to D \in C_S(D)$, then

  $$\mathsf{params}(d, f_i) = d^1_{f_i} : D_i^1, \ldots, \ldots d^{m_i}_{f_i} : D_i^{m_i},$$

  where all $d^j_{f_i}$ are fresh. Note that if f_i is a constant, $\mathsf{params}(d, f_i)$ is the empty sequence.
- The variables introduced for d are defined as follows.

  $$\mathsf{params}(d) = e_d : U_D, \mathsf{params}(d, f_0), \ldots, \mathsf{params}(d, f_n)$$

  Note that e_d is fresh.

In Definition 3 we define how we split a variable declaration, by the use of params, defining new variables and their types. With a slight abuse of notation we will also use params to indicate the use of the newly introduced variables, in which case their sorts are omitted.

We illustrate the definition using an example.

Example 6. Recall our running example with state variable $s : Sys$. Sort Sys has two constructors, $uninit : Sys$ and $sys : State \times N \to Sys$. Note that $\mathsf{params}(s, uninit)$ is empty, and $\mathsf{params}(s, sys) = s^1_{sys} : State, s^2_{sys} : N$. We thus get $\mathsf{params}(s) = e_s : U_{Sys}, s^1_{sys} : State, s^2_{sys} : N$.

4.3. Reconstructing variable use in a term

Next we turn our attention to terms. Suppose we have a term t that contains occurrences of variable d that is being unfolded. We need to replace t with an equivalent term using $\mathsf{params}(d)$ instead of d. The straightforward idea described by [22] is to syntactically substitute d with an application of the case function to $\mathsf{params}(d)$.

We call this (default) *case placement*, and formalize it as follows.

Definition 4. Let t be an arbitrary term, and $d : D$ a variable of constructor sort D. The *case placement* is the term $\mathsf{cp}(t, d)$ defined as:

$$\mathsf{cp}(t, d) = t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]$$

Example 7. Recall our running example with state variable $s : Sys$. Every occurrence of s is replaced by $C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys}))$. So, if e_s is the constructor $\overline{c}_{uninit}$ of sort U_{Sys} that represents $uninit$, this term evaluates to $uninit$; if e_s is $\overline{c}_{sys}$, the term evaluates to sys applied to arguments s^1_{sys} and s^2_{sys}.

For condition $s \approx uninit$ of the first summand in our running example, we then obtain the following term in which the case function has been placed:

$$\begin{aligned}\mathsf{cp}(s \approx uninit, s) &= (s \approx uninit)[s := C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys}))] \\ &= C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit\end{aligned}$$

Alternative case placement In the standard definition of case placement, cp, case functions are placed at an *innermost* level. This can limit simplification using the equational specification; e.g., the term $C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit$ from Example 7 cannot be simplified since we have no equation that allows distributing the case function over $\approx$.

In many cases, placing the case function at an *outermost* level aids simplification and subsequent analysis. Formally, every term t now becomes $C_D(e_d, t[d := f_0(\mathsf{params}(d, f_0))], \ldots, t[d := f_n(\mathsf{params}(d, f_n))])$. However, this may lead to an exponential blow-up in the size of the terms if multiple parameter unfoldings are performed successively. Therefore, we propose a new intermediate approach that places case functions at the level where subterms 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. This is possible by the addition of case function $C_D : U_D \times B \times \cdots \times B \to B$ with arity $|C_S(D)| + 1$.

Definition 5. Given a term t and a variable $d : D$, the *alternative case placement* is the term $\mathsf{acp}(t, d)$, where acp is the recursive function:

$$\begin{aligned}\mathsf{acp}(b, d) &= C_D(e_d, b[d := f_0(\mathsf{params}(d, f_0))], \ldots, b[d := f_n(\mathsf{params}(d, f_n))]) \\ \mathsf{acp}(\neg\varphi, d) &= \neg\mathsf{acp}(\varphi, d) \\ \mathsf{acp}(\varphi \wedge \psi, d) &= \mathsf{acp}(\varphi, d) \wedge \mathsf{acp}(\psi, d) \\ \mathsf{acp}(\varphi \vee \psi, d) &= \mathsf{acp}(\varphi, d) \vee \mathsf{acp}(\psi, d) \\ \mathsf{acp}(\varphi \Rightarrow \psi, d) &= \mathsf{acp}(\varphi, d) \Rightarrow \mathsf{acp}(\psi, d)\end{aligned}$$

Here, φ and ψ are arbitrary terms and b is a term 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 equation $C_D(e_d, x, x) = x$. We have the following relation between cp and acp.

Lemma 5. *Let t be an arbitrary term, and d a variable, then*

$$\mathsf{cp}(t, d) \equiv \mathsf{acp}(t, d).$$

Proof. Follows by induction on t and a case analysis on e_d. □

We next discuss the benefits of alternative case placement on our running example.

Example 8. In Example 7 we established that $\mathsf{cp}(s \approx uninit, s) = C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit$. This case function cannot be simplified further, as the first argument e_s is a variable, and it cannot be matched to any of the equations in the equational specification; also, there are no equations 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 equations for $\approx$ can be used to simplify the individual arguments.

Concretely, we have the following:

$$\begin{aligned}\mathsf{acp}(s \approx uninit, s) &= C_{Sys}(e_s, (s \approx uninit)[s := uninit], (s \approx uninit)[s := sys(s^1_{sys}, s^2_{sys})]) \\ &= C_{Sys}(e_s, uninit \approx uninit, sys(s^1_{sys}, s^2_{sys}) \approx uninit) \\ &= C_{Sys}(e_s, true, false)\end{aligned}$$

Observe that the term has been simplified further. In particular, there are now no references to s^1_{sys} and s^2_{sys}.

4.4. Splitting variable assignments

The final case we need to consider when unfolding state variables is an assignment $d := e$. If d is replaced by $\mathsf{params}(d)$, we also need to calculate the appropriate assignments to the new variables from the single term e. Groote and Lisser [22] show how to achieve this using the determinizer and projection functions. We formalize this as follows.

Definition 6. Let t be a term of constructor sort D, with $C_S(D) = \{f_0, \ldots, f_n\}$. We define the following.

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

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

Example 9. Recall our running example. The first summand performs the assignment $s := sys(p\_off, zero)$. When unfolding s, instead the variables become e_s, s^1_{sys}, s^2_{sys}, and the values that are assigned are obtained by using $\mathsf{unfold}(sys(p\_off, zero))$, which is calculated as follows:

$$\begin{aligned}\mathsf{unfold}(sys(p\_off, zero)) &= det_{Sys}(sys(p\_off, zero)), \pi^1_{sys}(sys(p\_off, zero)), \pi^2_{sys}(sys(p\_off, zero)) \\ &= \overline{c}_{sys}, p\_off, zero\end{aligned}$$

Note that in the calculation we use the definitions of det_{Sys} and π^i_{sys} described in Example 3.

Simplifications for pattern matching rules As a result of unfolding variable assignments, we regularly obtain terms of the shape $det_D(h(t_1, \ldots, t_n))$ or $\pi^l_{f_k}(h(t_1, \ldots, t_n))$ for some mapping h (*i.e.*, h is not a constructor). Both of these cannot be simplified any further, often due to the fact that there is insufficient information to apply the pattern matching in the equations for h. To alleviate this, we propose a new method to perform one unfolding of the function h, allowing us to achieve the necessary simplifications. Let us first consider an example.

Example 10. Suppose we have a definition of lists of natural numbers, with a function *plusone*, which is defined using pattern matching, that increments every element of a list.

sort $ListN$;
cons $[] : ListN$;
$\triangleright : N \times ListN \to ListN$;
map $plusone : ListN \to ListN$;
var $x : N$; $xs : ListN$;
eqn $plusone([]) = []$;
$plusone(x \triangleright xs) = (x + 1) \triangleright plusone(xs)$;

Suppose we have a state variable l of sort $ListN$ with an assignment $l := plusone(l)$. The first argument of $\mathsf{params}(l)$ is $e_l : U_{ListN}$, and the first argument update obtained from $\mathsf{unfold}(plusone(l))$ is the term $det_{ListN}(plusone(C_{ListN}(e_l, [], s^1_{\triangleright} \triangleright s^2_{\triangleright})))$, which cannot be simplified any further. Here the term $det_{ListN}(plusone(C_{ListN}(e_l, [], s^1_{\triangleright} \triangleright s^2_{\triangleright})))$ is found as follows: first, by definition of cp, l is replaced in $plusone(l)$ by $C_{ListN}(e_l, [], s^1_{\triangleright} \triangleright s^2_{\triangleright})$, then, unfold is applied to $plusone(C_{ListN}(e_l, [], s^1_{\triangleright} \triangleright s^2_{\triangleright}))$ such that $det_{ListN}(plusone(C_{ListN}(e_l, [], s^1_{\triangleright} \triangleright s^2_{\triangleright})))$ is obtained.

Intuitively, since det_{ListN} considers only its argument's constructor, and *plusone* does not modify the constructor, $det_{ListN}(l) = det_{ListN}(plusone(l))$ for all l. However, due to the pattern matching nature of *plusone*, we can only eliminate the application of det_{ListN} by means of term rewriting if l is of the shape $[]$ or $x \triangleright xs$. Thus, we are not able to automatically deduce that the update in the example above is in fact equal to e_l, and that the assignment does not modify e_l. 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 equation 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.

Lemma 6. *Let $h : D_1 \times \cdots \times D_n \to D$ be a mapping and $t_1, \ldots, t_n$ arbitrary terms. Then we have for any σ and any $1 \leq i \leq n$:*

$$\begin{aligned}[\![h(t_1,\ldots,t_n)]\!]^\sigma = [\![&C_{D_i}(det_{D_i}(t_i),\\ &h(t_1,\ldots,t_{i-1},f_1(\pi^1_{f_1}(t_i),\ldots,\pi^{m_1}_{f_1}(t_i)),t_{i+1},\ldots,t_n),\\ &\ldots,\\ &h(t_1,\ldots,t_{i-1},f_{|\mathcal{C}_S(D_i)|}(\pi^1_{f_{|\mathcal{C}_S(D_i)|}}(t_i),\ldots,\pi^{m_{|\mathcal{C}_S(D_i)|}}_{f_{|\mathcal{C}_S(D_i)|}}(t_i)),t_{i+1},\ldots,t_n))]\!]^\sigma\end{aligned}$$

where m_k denotes the index of the last argument of constructor f_k.

Proof. Let h, $t_1,\ldots,t_n$ and i be as above. By Lemma 3, let $[\![t_i]\!]^\sigma = [\![f_k(u_1,\ldots,u_{m_k})]\!]^\sigma$ for some $1 \leq k \leq |\mathcal{C}_S(D_i)|$ and some terms $u_1,\ldots,u_{m_k}$. After unfolding the semantics, we can apply the equations

$$det_{D_i}(f_k(u_1,\ldots,u_{m_k})) = \overline{c}_{f_k}$$
$$C_{D_i}(\overline{c}_{f_k},x_1,\ldots,x_k,\ldots,x_{|\mathcal{C}_S(D_i)|}) = x_k$$

to obtain the desired equality. □

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

Example 11. We revisit the term $det_{ListN}(plusone(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})))$ obtained from unfolding in Example 10. Applying Lemma 6 on $plusone(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright}))$, we obtain the following term:

$$\begin{aligned}&det_{ListN}(C_{ListN}(det_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})),\\ &\quad plusone([\,]),\\ &\quad plusone(\pi^1_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})) \triangleright \pi^2_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})))))\end{aligned}$$

By the definition of *plusone*, the above term is logically equivalent to

$$\begin{aligned}&det_{ListN}(C_{ListN}(det_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})),\\ &\quad [\,],\\ &\quad (\pi^1_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})) + 1) \triangleright plusone(\pi^2_{ListN}(C_{ListN}(e_l, [\,], s^1_{\triangleright} \triangleright s^2_{\triangleright})))))\end{aligned}$$

Thus, we now managed to eliminate outermost occurrences of *plusone*. After repeated distribution of det_{ListN} over C_{ListN}, this term can ultimately be rewritten to simply e_l.

4.5. Properties of unfold and case placement

The definitions of unfold and case placement work closely together in the following sense. Given a term t, unfold replaces a variable d with a case function over $\mathsf{params}(d)$. If we originally assigned a term e to d, $\mathsf{unfold}(e)$ determines the terms that need to be assigned to the new parameters in order to obtain an equivalent term.

Lemma 7. *For all constructor sorts D, variables d, terms e of sort D and valuations σ, we have*

$$[\![e]\!]^\sigma = [\![C_D(det_D(e), f_0(\pi^1_{f_0}(e),\ldots,\pi^{m_0}_{f_0}(e)),\ldots,f_n(\pi^1_{f_n}(e),\ldots,\pi^{m_n}_{f_n}(e)))]\!]^\sigma$$

Proof. Fix constructor sort D with constructors $\mathcal{C}_S(D) = \{f_0,\ldots,f_n\}$, variable d and term e of sort D and valuation σ. According to Lemma 3, there exist $f_i \in \mathcal{C}_S(D)$, $e_1,\ldots e_{m_i}$ and fresh variables $x_1,\ldots x_{m_i}$ such that $[\![e]\!]^\sigma = [\![f_i(x_1,\ldots x_{m_i})]\!]^{\sigma[e_1/x_1,\ldots,e_{m_i}/x_{m_i}]}$. Pick such f_i, $e_1,\ldots e_{m_i}$ and $x_1,\ldots x_{m_i}$. We now derive the following.

$$\begin{aligned}&[\![e]\!]^\sigma\\ &= \{\text{Lemma 3}\}\\ &[\![f_i(x_1,\ldots x_{m_i})]\!]^{\sigma[e_1/x_1,\ldots,e_{m_i}/x_{m_i}]}\\ &= \{\text{Definition of } \pi^j_{f_i}\}\\ &[\![f_i(\pi^1_{f_i}(f_i(x_1,\ldots x_{m_i})),\ldots,\pi^{m_i}_{f_i}(f_i(x_1,\ldots x_{m_i})))]\!]^{\sigma[e_1/x_1,\ldots,e_{m_i}/x_{m_i}]}\end{aligned}$$

$$
\begin{aligned}
&= \{\text{Definition of case function } C_D\}\\
&[\![C_D(\overline{c}_{f_i}, f_0(\pi^1_{f_0}(f_i(x_1,\dots x_{m_i})),\dots,\pi^{m_0}_{f_0}(f_i(x_1,\dots x_{m_i}))),\dots,\\
&\qquad f_n(\pi^1_{f_n}(f_i(x_1,\dots x_{m_i})),\dots,\pi^{m_n}_{f_n}(f_i(x_1,\dots x_{m_i}))))]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}\\
&= \{\text{Definition of } det_D\}\\
&[\![C_D(det_D(f_i(x_1,\dots x_{m_i})), f_0(\pi^1_{f_0}(f_i(x_1,\dots x_{m_i})),\dots,\pi^{m_0}_{f_0}(f_i(x_1,\dots x_{m_i}))),\dots,\\
&\qquad f_n(\pi^1_{f_n}(f_i(x_1,\dots x_{m_i})),\dots,\pi^{m_n}_{f_n}(f_i(x_1,\dots x_{m_i}))))]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}\\
&= \{\text{Semantics}\}\\
&[\![C_D]\!]([\![det_D]\!]([\![f_i(x_1,\dots x_{m_i})]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}),\\
&\qquad [\![f_0]\!]([\![\pi^1_{f_0}]\!]([\![f_i(x_1,\dots x_{m_i})]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}),\dots,[\![\pi^{m_0}_{f_0}]\!]([\![f_i(x_1,\dots x_{m_i})]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}),\dots,\\
&\qquad [\![f_n]\!]([\![\pi^1_{f_n}]\!]([\![f_i(x_1,\dots x_{m_i})]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}),\dots,[\![\pi^{m_n}_{f_n}]\!]([\![f_i(x_1,\dots x_{m_i})]\!]^{\sigma[e_1/x_1,\dots,e_{m_i}/x_{m_i}]}))\\
&= \{\text{Lemma 3}\}\\
&[\![C_D]\!]([\![det_D]\!]([\![e]\!]^\sigma),\\
&\qquad [\![f_0]\!]([\![\pi^1_{f_0}]\!]([\![e]\!]^\sigma),\dots,[\![\pi^{m_0}_{f_0}]\!]([\![e]\!]^\sigma)),\dots,\\
&\qquad [\![f_n]\!]([\![\pi^1_{f_n}]\!]([\![e]\!]^\sigma),\dots,[\![\pi^{m_n}_{f_n}]\!]([\![e]\!]^\sigma)))\\
&= \{\text{Semantics}\}\\
&[\![C_D(det_D(e), f_0(\pi^1_{f_0}(e),\dots,\pi^{m_0}_{f_0}(e)),\dots,f_n(\pi^1_{f_n}(e),\dots,\pi^{m_n}_{f_n}(e)))]\!]^\sigma \quad \square
\end{aligned}
$$

We use this to establish correctness of case placement in a term as established by the following lemma.

Lemma 8. *For all variables d and terms e of constructor sort D, terms t, and valuations σ such that $\sigma(d) = [\![e]\!]^\sigma$ and $\sigma(\mathsf{params}(d)) = [\![\mathsf{unfold}(e)]\!]^\sigma$, it holds that*

$$[\![t]\!]^\sigma = [\![\mathsf{cp}(t,d)]\!]^\sigma.$$

Proof. Fix d and e and σ as above. The proof proceeds by induction on the structure of t.

- $t \in \mathcal{X}_{\mathcal{S}}$. So, t is a variable. If $t \neq d$, then $\mathsf{cp}(t,d) = t$, and the result follows immediately. So, assume that $t = d$. We argue as follows.

$$
\begin{aligned}
&[\![d]\!]^\sigma\\
&= \{\text{Semantics}\}\\
&\sigma(d)\\
&= \{\text{Valuation } \sigma\}\\
&[\![e]\!]^\sigma\\
&= \{\text{Lemma 7}\}\\
&[\![C_D(det_D(e), f_0(\pi^1_{f_0}(e),\dots,\pi^{m_0}_{f_0}(e)),\dots,f_n(\pi^1_{f_n}(e),\dots,\pi^{m_n}_{f_n}(e)))]\!]^\sigma\\
&= \{\text{Substitution, definition of } \mathsf{params}(d,f_i)\}\\
&[\![C_D(e_d, f_0(\mathsf{params}(d,f_0)),\dots,f_n(\mathsf{params}(d,f_n)))\\
&\qquad [e_d := det_D(e), d^1_{f_0} := \pi^1_{f_0}(e),\dots,d^{m_0}_{f_0} := \pi^{m_0}_{f_0}(e)),\dots,d^1_{f_n} := \pi^1_{f_n}(e),\dots,d^{m_n}_{f_n} := \pi^{m_n}_{f_n}(e))]\!]^\sigma\\
&= \{\text{Definition of } \mathsf{params}(d) \text{ and } \mathsf{unfold}(e)\}\\
&[\![C_D(e_d, f_0(\mathsf{params}(d,f_0)),\dots,f_n(\mathsf{params}(d,f_n)))[\mathsf{params}(d) := \mathsf{unfold}(e)]]\!]^\sigma\\
&= \{\text{Lemma 2, assumption on } \sigma\}\\
&[\![C_D(e_d, f_0(\mathsf{params}(d,f_0)),\dots,f_n(\mathsf{params}(d,f_n)))]\!]^\sigma\\
&= \{\text{Definition of } \mathsf{cp}\}\\
&[\![\mathsf{cp}(d,d)]\!]^\sigma
\end{aligned}
$$

- $t = f \in \mathcal{C}_S \cup \mathcal{M}_S$. Then $\mathsf{cp}(t,d) = \mathsf{cp}(f,d) = f = t$ and the result is immediate.
- $t = t'(t_1,\ldots,t_n)$. We reason as follows.

$$
\begin{aligned}
&[\![t'(t_1,\ldots,t_n)]\!]^\sigma \\
&= \{\text{Semantics}\} \\
&[\![t']\!]^\sigma([\![t_1]\!]^\sigma,\ldots,[\![t_n]\!]^\sigma) \\
&= \{\text{IH}\} \\
&[\![\mathsf{cp}(t',d)]\!]^\sigma([\![\mathsf{cp}(t_1,d)]\!]^\sigma,\ldots,[\![\mathsf{cp}(t_n,d)]\!]^\sigma) \\
&= \{\text{Definition of cp}\} \\
&[\![t'[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))]]\!]^\sigma \\
&\quad([\![t_1[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))]]\!]^\sigma,\ldots, \\
&\qquad [\![t_n[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))]]\!]^\sigma) \\
&= \{\text{Semantics}\} \\
&[\![t'[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))] \\
&\quad(t_1[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))],\ldots, \\
&\qquad t_n[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))])]\!]^\sigma \\
&= \{\text{Definition of substitution}\} \\
&[\![(t'(t_1,\ldots,t_n)[d := C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))])]\!]^\sigma \\
&= \{\text{Definition of cp}\} \\
&[\![(\mathsf{cp}(t'(t_1,\ldots,t_n),d)]\!]^\sigma \quad \square
\end{aligned}
$$

Since d and $\mathsf{params}(d)$ do not appear on one side of the equation, we also immediately get the following corollary using Lemma 1. This shows the precise interplay between the new parameters generated by $\mathsf{params}(d)$ and the application of case placement, as well as the unfolding of term e.

Corollary 1. *For all variables d and terms e of constructor sort D, terms t and valuations σ*

$$[\![t]\!]^{\sigma[[\![e]\!]^\sigma/d]} = [\![\mathsf{cp}(t,d)]\!]^{\sigma[[\![\mathsf{unfold}(e)]\!]^\sigma/\mathsf{params}(d)]}.$$

Proof. Fix d and e and σ as above. The proof proceeds by induction on the structure of t.

- $t \in \mathcal{X}_S$. So, t is a variable. If $t \neq d$, then $\mathsf{cp}(t,d) = t$, and the result follows immediately. So, assume that $t = d$. We argue as follows.

$$
\begin{aligned}
&[\![d]\!]^{\sigma[[\![e]\!]^\sigma/d]} \\
&= \{\text{Lemma 2}\} \\
&[\![d[d := e]]\!]^\sigma \\
&= \{\text{Substitution}\} \\
&[\![e]\!]^\sigma \\
&= \{\text{Lemma 7}\} \\
&[\![C_D(det_D(e), f_0(\pi^1_{f_0}(e),\ldots,\pi^{m_0}_{f_0}(e)),\ldots,f_n(\pi^1_{f_n}(e),\ldots,\pi^{m_n}_{f_n}(e)))]\!]^\sigma \\
&= \{\text{Substitution, definition of } \mathsf{params}(d,f_i)\} \\
&[\![C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n))) \\
&\qquad [e_d := det_D(e), d^1_{f_0} := \pi^1_{f_0}(e),\ldots,d^{m_0}_{f_0} := \pi^{m_0}_{f_0}(e)),\ldots,d^1_{f_n} := \pi^1_{f_n}(e),\ldots,d^{m_n}_{f_n} := \pi^{m_n}_{f_n}(e))]\!]^\sigma \\
&= \{\text{Definition of } \mathsf{params}(d) \text{ and } \mathsf{unfold}(e)\} \\
&[\![C_D(e_d, f_0(\mathsf{params}(d,f_0)),\ldots,f_n(\mathsf{params}(d,f_n)))[\mathsf{params}(d) := \mathsf{unfold}(e)]]\!]^\sigma
\end{aligned}
$$

$= \{\text{Lemma 2}\}$

$[\![C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]\!]^{\sigma[[\![\mathsf{unfold}(e)]\!]^\sigma/\mathsf{params}(d)]}$

$= \{\text{Definition of cp}\}$

$[\![\mathsf{cp}(d, d)]\!]^{\sigma[[\![\mathsf{unfold}(e)]\!]^\sigma/\mathsf{params}(d)]}$

- $t = f \in \mathcal{C}_S \cup \mathcal{M}_S$. Then $\mathsf{cp}(t, d) = \mathsf{cp}(f, d) = f = t$ and the result is immediate.
- $t = t'(t_1, \ldots, t_n)$. The result follows immediately from the semantics and the induction hypothesis. □

This result also immediately extends to the vectors of expressions obtained by unfolding a term.

Corollary 2. *For all variables d, terms e and t, all of which are of constructor sort D and for all σ*

$$[\![\mathsf{unfold}(t)]\!]^{\sigma[[\![e]\!]^\sigma/d]} = [\![\mathsf{unfold}(\mathsf{cp}(t, d))]\!]^{\sigma[[\![\mathsf{unfold}(e)]\!]^\sigma/\mathsf{params}(d)]}.$$

Finally, the order in which unfolding and case placement are performed to a term does not matter. This is formalized in the following lemma.

Lemma 9. *Let t be a term, and d a variable of constructor sort D with $\mathcal{C}_S(D) = \{f_0, \ldots, f_n\}$, then*

$$\mathsf{unfold}(\mathsf{cp}(t, d)) = \mathsf{cp}(\mathsf{unfold}(t), d).$$

Proof. Let t and d be as above. We derive the following.

$\mathsf{unfold}(\mathsf{cp}(t, d))$

$= \{\text{Definition of cp}\}$

$\mathsf{unfold}(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))])$

$= \{\text{Definition of unfold}\}$

$det_D(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]),$

$\quad \pi^1_{f_0}(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]), \ldots,$

$\quad \pi^{m_0}_{f_0}(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]), \ldots,$

$\quad \pi^1_{f_n}(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]), \ldots,$

$\quad \pi^{m_n}_{f_n}(t[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))])$

$= \{\text{Property of substitution}\}$

$(det_D(t), \pi^1_{f_0}(t), \ldots, \pi^{m_0}_{f_0}(t), \ldots, \pi^1_{f_n}(t), \ldots, \pi^{m_n}_{f_n}(t))$

$\quad [d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]$

$= \{\text{Definition of unfold}\}$

$\mathsf{unfold}(t)[d := C_D(e_d, f_0(\mathsf{params}(d, f_0)), \ldots, f_n(\mathsf{params}(d, f_n)))]$

$= \{\text{Definition of cp}\}$

$\mathsf{cp}(\mathsf{unfold}(t), d)$ □

5. Unfolding parameters in mCRL2

In the remainder of this paper we show how the generic unfolding technique from the previous section can be applied in mCRL2. The mCRL2 language is a modeling language based on process algebra with data [25]. In general, the language allows for the specification of the behavior of communicating, parallel processes. However, the first step in any automated analysis using the mCRL2 toolset [14] 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). We apply the unfolding of variables to such LPEs.

5.1. Data specification

In mCRL2, the equational specification defined in Section 3 is referred to as *data specification*. The algebraic data types in mCRL2 can be specified using a richer syntax than we introduced in Sections 2 and 3. In particular, mCRL2 has default definitions of common data types such as Booleans (*Bool*), and numeric data types such as natural numbers (*Nat*), as well as container sorts such as lists ($List(D)$). In the remainder of the paper we assume that the standard data types from mCRL2 and their standard operations such as $\wedge$ and $+$ are part of the signature. For any sort D, we assume sort $List(D)$ is defined, with constructors $[]$ for the empty list, and $\triangleright$ for the constructor that adds an element in front of a list.

In addition to the standard data types, for every data type D that is introduced in mCRL2, the mappings for comparisons ($\approx$, $\not\approx$, $<$, $\leq$, $>$, $\geq$) as well as an if-then-else $if : Bool \times D \times D \to D$ are introduced by default. The default specification is only partial, and contains, e.g. $x \approx x = true$. For the standard data types these are extended to their full definition.

Finally, mCRL2 allows for the definition of *structured sorts*, which essentially are sorts with value constructors, and associated recognizer and projection functions. For such sorts, the comparison operators are fully defined. This allows for a much more compact and convenient definition of algebraic data types. We illustrate the use of structured sorts by redefining the *State* and *Sys* sorts from Fig. 2 in this manner.

Example 12. Recall the algebraic data types *State* and *Sys* from Fig. 2. Using structured sorts, these can be defined in mCRL2 as follows:

sort $State = \mathbf{struct}\ p\_on \mid p\_off;$
$Sys = \mathbf{struct}\ uninit?is\_uninit \mid sys(get\_state : State, get\_ip : Nat)?is\_sys;$
map $set\_state : Sys \times State \to Sys;$
$set\_ip : Sys \times Nat \to Sys;$
var $p_1, p_2 : State, n, m : Nat;$
eqn $set\_state(sys(p_1, n), p_2) = sys(p_2, n);$
$set\_ip(sys(p_1, n), m) = sys(p_1, m);$

The unfolding of sort *Sys* is identical to the unfolding in Example 3.

Structured sorts can be translated into equivalent sorts with an explicit constructor definition. For our running example, the result would be the definitions from Section 2, extended with full definitions of the comparisons and recognizer functions. The semantics of algebraic data types in mCRL2 follows the model class semantics as outlined in Section 3. For a complete overview of mCRL2's data types and their semantics see [25].

5.2. 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. The examples from Section 2, in fact, are LPEs.

An LPE is defined in the context of a data specification $\mathcal{D}$, that specifies algebraic data types, and a set of global variables $\mathcal{X}_g$. These global variables are parameters to the LPE, but their value is immaterial to the behavior described by the LPE. Global variables can thus be seen as "do not care" values; later we discuss this in more detail. The combination of an LPE with a data specification and its global variables is a Linear Process Specification (LPS).

Definition 7. 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 terms 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:

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

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 conditions, 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 local variables $\vec{e}_i$ of sort $\vec{E}_i$ ($\vec{e}_j$ of sort $\vec{E}_j$).

The operational semantics of LPEs induces a *labelled transition system*, see [25] for its definition. The definition in [25] assumes that every value $v \in M_D$ in the data types has a syntactic denotation as closed terms t_v [25, Definition 15.2.17]. In the remainder of this paper we also use this assumption.

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, and we generally assume processes whose parameters we unfold have a single

parameter, and summands with a single local variable; the generalization to multiple parameters is straightforward. Of course, the resulting process will have more than one parameter.

Example 13. We recall our motivating example from Section 2. So far, we have reset the IP-address to *zero* when the state is *p_off*. We can make the fact that we do not care about the value of the IP-address explicit by, instead, using a global variable when changing the state to *p_off*. Let $\mathcal{D}$ be the data specification described in Example 12. We use mCRL2 syntax to describe the global variables (**glob**), LPE (**proc**) and initialization (**init**).

glob $dc1, dc2 : Nat;$
proc $P(s : Sys) =$
$(s \approx uninit) \to initialize \cdot P(sys(p\_off, dc1))$
$+ \sum_{n : Nat}(s \not\approx uninit \wedge get\_state(s) \approx p\_off) \to on \cdot P(set\_state(set\_ip(s, n), p\_on))$
$+(s \not\approx uninit \wedge get\_state(s) \approx p\_on) \to off \cdot P(set\_state(set\_ip(s, dc2), p\_off));$
init $P(uninit);$

Transformations of LPEs are correct if they are behavior preserving. For this, we use a generalization of strong bisimulation to linear processes [22]. 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 $P(e) \leftrightarroweq P'(e')$. For ease of definition, we assume that the process parameters and summation variables in the processes are disjoint (since this can easily be achieved by renaming, this does not affect generality). Formally, strong bisimulation of LPEs is defined as follows.

Definition 8 *(Strong bisimulation of LPEs [22]).* Let $\mathcal{D} = (\Sigma, E)$ be a data specification, and let $\vec{D} = D_1, \ldots, D_n$, $\vec{D'} = D'_1, \ldots, D'_m$. Consider the following two LPEs.

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

$$Q(\vec{d'} : \vec{D'}) = \sum_{i \in I'} \sum_{\vec{e'}_i : \vec{E'}_i} c'_i \to a'_i(f'_i) \cdot Q(g'_i)$$

Relation $R \subseteq (M_{D_1} \times \cdots \times M_{D_n}) \times (M_{D'_1} \times \cdots \times M_{D'_m})$ is a *strong bisimulation* iff for all terms $\vec{e}$, $\vec{e'}$ and valuations σ and σ', if $[\![\vec{e}]\!]^\sigma \; R \; [\![\vec{e'}]\!]^{\sigma'}$, then:

- for all $i \in I$, $\vec{w}_i \in M_{\vec{E}_i}$, such that $[\![c_i]\!]^{\sigma[[\![\vec{e}]\!]^\sigma/\vec{d}, \vec{w}_i/\vec{e}_i]} = \mathbf{true}$ there is some $i' \in I'$ and $\vec{w}'_{i'}$ such that
  - $[\![c'_{i'}]\!]^{\sigma'[[\![\vec{e'}]\!]^\sigma/\vec{d'}, \vec{w}_{i'}/\vec{e}_{i'}]} = \mathbf{true}$,
  - $a_i = a'_{i'}$,
  - $[\![f_i]\!]^{\sigma[[\![\vec{e}]\!]^\sigma/\vec{d}, \vec{w}_i/\vec{e}_i]} = [\![f'_{i'}]\!]^{\sigma'[[\![\vec{e'}]\!]^\sigma/\vec{d'}, \vec{w}_{i'}/\vec{e}_{i'}]}$, and
  - $[\![g_i]\!]^{\sigma[[\![\vec{e}]\!]^\sigma/\vec{d}, \vec{w}_i/\vec{e}_i]} \; R \; [\![g'_{i'}]\!]^{\sigma'[[\![\vec{e'}]\!]^\sigma/\vec{d'}, \vec{w}_{i'}/\vec{e}_{i'}]}$.
- vice versa.

Process terms $P(\vec{t})$ and $Q(\vec{t'})$ are strongly bisimilar w.r.t. data specification D, denoted $P(\vec{t}) \leftrightarroweq Q(\vec{t'})$, iff for all D-models $\mathbb{M}$ and valuations σ and σ' there is a bisimulation relation R such that $[\![\vec{t}]\!]^\sigma \; R \; [\![\vec{t'}]\!]^{\sigma'}$. It is well known that the composition of strong bisimulation relations is again a strong bisimulation relation.

Note that Groote and Lisser adapted the standard definition of strong bisimulation [49] to LPEs: if process P can do an action $a_i(f_i)$, since its condition c_i is true, process Q can do the same action, and the target states are related by the strong bisimulation relation. If the LPEs do not refer to global variables, the valuations are fully defined by the assignment of values to process parameters and sum variables. As a consequence, their version of strong bisimulation is an equivalence relation (in particular, it is a reflexive relation).

The fact that we allow for global variables in the definition of processes means that strong bisimulation is no longer reflexive. However, global variables are generally assumed to not have any significant effect on the behavior of a process. This is captured by the reflexivity property (**Refl**).

(**Refl**) Let $L = (\mathcal{D}, \mathcal{X}_g, P, \vec{e})$ be an LPS. Then $P(\vec{e}) \leftrightarroweq P(\vec{e})$.

By the definition of strong bisimulation, this implies that for every pair of valuations σ, σ' there exists a strong bisimulation relation R such that $[\![\vec{e}]\!]^\sigma \; R \; [\![\vec{e}]\!]^{\sigma'}$. As $\vec{e}$ is a vector of terms over $\mathcal{X}_g$, the only (potentially) relevant difference is in the assignment to global variables, but any assignment to the variables leads to bisimilar processes. When discussing the correctness of our transformations, we implicitly assume that the input LPS satisfies the (**Refl**) property.

5.3. Unfolding process parameters in an LPE

The basic definition of the unfolding of process parameters using cp, and without pattern match unfolding, was described by Groote and Lisser [22].

Definition 9 *(Unfolding of process parameters [22]).* Let $L = (\mathcal{D}, \mathcal{X}_g, P, e)$ be an LPS, where P is the following LPE.

$$P(d : D) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to a_i(f_i) \cdot P(g_i)$$

The result of unfolding process parameter $d : D$ in 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:

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

So, essentially, unfolding parameter d replaces d by the vector $\mathsf{params}(d)$. In recursive calls to P, the term g_i assigned to the unfolded parameter is also unfolded using $\mathsf{unfold}(g_i)$. Similarly, using $\mathsf{unfold}(e)$, the initial process is unfolded. Finally, in the right hand side of the equation, default case placement is used to replace every occurrence of d by an application of the corresponding case function.

We illustrate the combined application of all transformations on our running example.

Example 14. Recall our example with global variables from Example 13, for which we have described the unfolding of sort *Sys* in the data specification in Example 3. If we unfold parameter s, we get the LPE and initialization shown below.

$$\begin{aligned}
&\textbf{glob} \quad dc1, dc2 : Nat;\\
&\textbf{proc} \quad P(e_s : U_{Sys}, s^1_{sys} : State, s^2_{sys} : Nat) =\\
&\quad (C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit)\\
&\qquad \to initialize \cdot P(det_{Sys}(sys(p_off, dc1)), \pi^1_{sys}(sys(p_off, dc1)), \pi^2_{sys}(sys(p_off, dc1)))\\
&\quad + \textstyle\sum_{n : Nat}(\neg(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit) \wedge get_state(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys}))) \approx p_off)\\
&\qquad \to on \cdot P(det_{Sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), n), p_on)),\\
&\qquad\qquad \pi^1_{sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), n), p_on)),\\
&\qquad\qquad \pi^2_{sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), n), p_on)))\\
&\quad + (\neg(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})) \approx uninit) \wedge get_state(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys}))) \approx p_on)\\
&\qquad \to off \cdot P(det_{Sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), dc2), p_off)),\\
&\qquad\qquad \pi^1_{sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), dc2), p_off)),\\
&\qquad\qquad \pi^2_{sys}(set_state(set_ip(C_{Sys}(e_s, uninit, sys(s^1_{sys}, s^2_{sys})), dc2), p_off)));\\
&\textbf{init} \quad P(det_{Sys}(uninit), \pi^1_{sys}(uninit), \pi^2_{sys}(uninit));
\end{aligned}$$

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

Using the equations for det_{Sys}, π^1_{sys} and π^2_{sys} this can be simplified slightly. The recursion of the first summand then becomes $P(\overline{c}_{sys}, p\_off, dc1)$ and the initialization becomes **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_{sys} and s^2_{sys} 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.

Alternative case placement If, in Definition 9, we use acp instead of cp, the changed level of placement of case functions dramatically affects the simplifications that are allowed after the transformation. We show the result of the unfolding using alternative case placement.

Example 15. Recall our example with global variables from Example 13 which we have unfolded using cp in Example 14. We now instead transform terms using alternative case placement (acp).

$$
\begin{aligned}
&\textbf{glob}\quad dc1, dc2 : Nat;\\
&\textbf{proc}\quad P(e_s : U_{Sys}, s^1_{sys} : State, s^2_{sys} : Nat) =\\
&\quad (C_{Sys}(e_s, uninit \approx uninit, sys(s^1_{sys}, s^2_{sys}) \approx uninit)\\
&\qquad \rightarrow initialize \cdot P(det_{Sys}(sys(p\_off, dc1)), \pi^1_{sys}(sys(p\_off, dc1)), \pi^2_{sys}(sys(p\_off, dc1)))\\
&\quad + \textstyle\sum_{n : Nat}(\neg C_{Sys}(e_s, uninit \approx uninit, sys(s^1_{sys}, s^2_{sys}) \approx uninit) \wedge\\
&\qquad C_{Sys}(e_s, get\_state(uninit) \approx p\_on, get\_state(sys(s^1_{sys}, s^2_{sys})) \approx p\_off))\\
&\qquad \rightarrow on \cdot P(C_{Sys}(e_s, det_{Sys}(set\_state(set\_ip(uninit, n), p\_on)),\\
&\qquad\qquad det_{Sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), n), p\_on))),\\
&\qquad\qquad C_{Sys}(e_s, \pi^1_{sys}(set\_state(set\_ip(uninit, n), p\_on)),\\
&\qquad\qquad\qquad \pi^1_{sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), n), p\_on))),\\
&\qquad\qquad C_{Sys}(e_s, \pi^2_{sys}(set\_state(set\_ip(uninit, n), p\_on)),\\
&\qquad\qquad\qquad \pi^1_{sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), n), p\_on))))\\
&\quad + (\neg C_{Sys}(e_s, uninit \approx uninit, sys(s^1_{sys}, s^2_{sys}) \approx uninit) \wedge\\
&\qquad C_{Sys}(e_s, get\_state(uninit) \approx p\_on, get\_state(sys(s^1_{sys}, s^2_{sys})) \approx p\_on))\\
&\qquad \rightarrow off \cdot P(C_{Sys}(e_s, det_{Sys}(set\_state(set\_ip(uninit, dc2), p\_off)),\\
&\qquad\qquad det_{Sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), dc2), p\_off))),\\
&\qquad\qquad C_{Sys}(e_s, \pi^1_{sys}(set\_state(set\_ip(uninit, dc2), p\_off)),\\
&\qquad\qquad\qquad \pi^1_{sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), dc2), p\_off))),\\
&\qquad\qquad C_{Sys}(e_s, \pi^2_{sys}(set\_state(set\_ip(uninit, dc2), p\_off)),\\
&\qquad\qquad\qquad \pi^2_{sys}(set\_state(set\_ip(sys(s^1_{sys}, s^2_{sys}), dc2), p\_off))));\\
&\textbf{init}\quad P(det_{Sys}(uninit), \pi^1_{sys}(uninit), \pi^2_{sys}(uninit));
\end{aligned}
$$

As explained in Example 8, the case functions appear at a higher level, such that the case functions can be simplified further using the equations for $\approx$, det_{Sys}, π^1_{sys}, π^2_{sys}, set_ip and set_state. Using these, the last summand is simplified to:

$$
\begin{aligned}
&(\neg C_{Sys}(e_s, true, false) \wedge C_{Sys}(e_s, get\_state(uninit) \approx p\_on, s^1_{sys} \approx p\_on))\\
&\qquad \rightarrow off \cdot P(C_{Sys}(e_s, \overline{c}_{uninit}, \overline{c}_{sys}), C_{Sys}(e_s, p\_on, p\_off), C_{Sys}(e_s, 0, dc2))
\end{aligned}
$$

We thus obtained more concise terms than those in Example 14. In particular, this summand no longer contains any reference to unfolded parameter s^2_{sys}. The same applies to the other two summands, hence parameter s^2_{sys} can be eliminated using static analysis techniques [22]. As a result, the sum over n in the second summand is not used and can be eliminated as well. The final LPE we obtain is:

$$
\begin{aligned}
&\textbf{proc}\quad P(e_s : U_{Sys}, s^1_{sys} : State) =\\
&\quad C_{Sys}(e_s, true, false)\\
&\qquad \rightarrow initialize \cdot P(\overline{c}_{sys}, p\_off)\\
&\quad + (\neg C_{Sys}(e_s, true, false) \wedge C_{Sys}(e_s, get\_state(uninit) \approx p\_off, s^1_{sys} \approx p\_off))\\
&\qquad \rightarrow on \cdot P(C_{Sys}(e_s, \overline{c}_{uninit}, \overline{c}_{sys}), p\_on)\\
&\quad + (\neg C_{Sys}(e_s, true, false) \wedge C_{Sys}(e_s, get\_state(uninit) \approx p\_on, s^1_{sys} \approx p\_on))\\
&\qquad \rightarrow off \cdot P(C_{Sys}(e_s, \overline{c}_{uninit}, \overline{c}_{sys}), C_{Sys}(e_s, p\_on, p\_off))\\
&\textbf{init}\quad P(\overline{c}_{uninit}, p\_on);
\end{aligned}
$$

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.

5.4. Global variables

Some static analysis techniques in mCRL2 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 (by Property (**Refl**)). The technique from [22] does not give special treatment to such global variables. We first describe why global variables need special treatment, and subsequently describe how they should be treated.

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 terms such as $det_D(dc)$ and $\pi^j_{f_i}(dc)$ that cannot be simplified further. These more complex terms cannot be used directly for simplification in static analysis, leaving the resulting LPE more complicated than it needs to be. This results in longer verification times. We illustrate the issue using an example that is based on board games such as tic-tac-toe, which often represent the board using (lists of) lists.

Example 16. 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.

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

Unfolding parameter l yields the following LPE.

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

The recursion in the first summand cannot be simplified further, and no parameters can be removed during static analysis.

Since the behavior of a process is not affected by (the value of) a global variable, the individual arguments of the term 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 6 as follows.

Definition 10. Let e be a term of constructor sort D. Then

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

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

The unfolded LPE taking global variables into account is obtained using unfold_g instead of unfold in Definition 9. However, we need to take care that any other occurrences of the same global variable that is being replaced are updated consistently. This results in the following definition.[1]

Definition 11 *(Unfolding of process parameters with global variable replacement).* Let $L = (\mathcal{D}, \mathcal{X}_g, P, e)$ be an LPS, where P is the following LPE.

$$
P(d : D) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to a_i(f_i) \cdot P(g_i)
$$

The result of unfolding process parameter $d : D$ in L is the LPS

$$
L' = (\mathcal{D}', \mathcal{X}'_g, P', \mathsf{unfold}_g(e)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r})
$$

where $\mathcal{D}'$ is data specification $\mathcal{D}$ in which sort D is unfolded, and LPE P' is as follows:

$$
\begin{aligned}
& P'(\mathsf{params}(d)) \\
& = \sum_{i \in I} \sum_{e_i : E_i} \mathsf{cp}(c_i, d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r} \\
& \qquad \to a_i(\mathsf{cp}(f_i, d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}) \\
& \qquad\qquad \cdot P'(\mathsf{cp}(\mathsf{unfold}_g(g_i), d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r})
\end{aligned}
$$

where $\mathcal{X}_r \subseteq \mathcal{X}_g$ is the set of all global variables $dc : D$ that have been replaced by a vector of fresh global variables $\mathsf{unfold}_g(dc)$. $\mathcal{X}'_g$ is the set $\mathcal{X}_g$ extended with these fresh global variables.

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

We apply this improved definition to the specification in Example 16.

Example 17. Recall the specification from Example 16. When using unfold_g instead of unfold, the recursion in the first summand becomes $P(dc1_e, dc1_{lp}, dc1_{ll}, dc2, true)$.

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

$$\begin{aligned}&\textbf{proc} \quad P(l_p : Piece, p : Piece, done : Bool)\\ &\qquad = (\neg done \wedge l_p \approx p) \rightarrow is(p).P(p, other(p), done)\\ &\qquad +(\neg done \wedge l_p \approx other(p)) \rightarrow \tau.P(dc1_{lp}, dc2, true);\\ &\textbf{init} \quad P(o, o, false);\end{aligned}$$

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.

When unfolding parameters in other examples, for instance board games such as tic-tac-toe or four in a row, replacing global variables in the way described proves essential for eliminating all lists from the specification. In our experiments in Section 6 we will demonstrate that this results in a dramatic performance increase for symbolic reachability.

5.5. Correctness

We describe correctness of the unfolding with standard placement of case functions. A similar result was given, without proof, in [22]. The result in [22] does not allow for global variables in an LPE.

Theorem 1. *Let $\mathcal{D} = (\Sigma, E)$, and consider LPE $L = (\mathcal{D}, \mathcal{X}_g, P, e)$, where P is defined as:*

$$P(d : D) = \sum_{i \in I} \sum_{e_i : E_i} c_i \rightarrow a_i(f_i) \cdot P(g_i)$$

Also consider the result $L' = (\mathcal{D}', \mathcal{X}_g, P', \mathsf{unfold}(e))$ of unfolding parameter d, as in Definition 9,

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

Then $P(e) \leftrightarroweq P'(\mathsf{unfold}(e))$.

Proof. We need to show that for all valuations σ' and σ, there exists a bisimulation relation $R_{P,P'}$ such that $[\![e]\!]^{\sigma'} \; R_{P,P'} \; [\![\mathsf{unfold}(e)]\!]^{\sigma}$. Fix σ' and σ. As $P(e) \leftrightarroweq P(e)$, there exists a bisimulation relation $R_{P,P}$ such that $[\![e]\!]^{\sigma'} \; R_{P,P} \; [\![e]\!]^{\sigma}$. So, it suffices to prove there exists a bisimulation relation R such that $[\![e]\!]^{\sigma} \; R \; [\![\mathsf{unfold}(e)]\!]^{\sigma}$. It then follows that $R_{P,P'} = R_{P,P} \circ R$ is a bisimulation relation such that $[\![e]\!]^{\sigma'} \; R_{P,P'} \; [\![\mathsf{unfold}(e)]\!]^{\sigma}$.

Define relation R as follows:

$$R = \{([\![t]\!]^{\sigma}, [\![\mathsf{unfold}(t)]\!]^{\sigma}) \mid t \text{ is a term of sort } D\}$$

We prove R is a strong bisimulation relation. So, fix arbitrary term t such that $[\![t]\!]^{\sigma} \; R \; [\![\mathsf{unfold}(t)]\!]^{\sigma}$, and fix arbitrary $i \in I$, value $w_i \in M_{E_i}$ such that $[\![c_i]\!]^{\sigma[[\![t]\!]^{\sigma}/d, w_i/e_i]} = \mathbf{true}$. That is, the condition of summand i is satisfied. We need to show there exists a summand $i' \in I'$ and value $w'_{i'} \in M_{E'_{i'}}$ such that the condition of summand i' is satisfied, the action and its parameters match those of summand i, and the target states are related. We prove that this is witnessed by summand i and value w_i.

First, we show that $[\![\mathsf{cp}(c_i, d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^{\sigma}/\mathsf{params}(d), w_i/e_i]} = [\![c_i]\!]^{\sigma[[\![t]\!]^{\sigma}/d, w_i/e_i]}$. As $[\![c_i]\!]^{\sigma[[\![t]\!]^{\sigma}/d, w_i/e_i]} = \mathbf{true}$, it then follows that $[\![\mathsf{cp}(c_i, d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^{\sigma}/\mathsf{params}(d), w_i/e_i]} = \mathbf{true}$. The derivation is as follows.

$$\begin{aligned}&[\![\mathsf{cp}(c_i, d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^{\sigma}/\mathsf{params}(d), w_i/e_i]}\\ &= \{\mathsf{params}(d) \text{ fresh}\}\\ &[\![\mathsf{cp}(c_i, d)]\!]^{\sigma[w_i/e_i][[\![\mathsf{unfold}(t)]\!]^{\sigma}/\mathsf{params}(d)]}\\ &= \{\text{Corollary 1}\}\\ &[\![c_i]\!]^{\sigma[w_i/e_i][[\![t]\!]^{\sigma}/d]}\\ &= \{\mathsf{params}(d) \text{ fresh}\}\\ &[\![c_i]\!]^{\sigma[[\![t]\!]^{\sigma}/d, w_i/e_i]}\end{aligned}$$

By construction of the unfolded LPE, $a_i = a_i$. The proof that $[\![\mathsf{cp}(f_i, d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^{\sigma}/\mathsf{params}(d), w_i/e_i]} = [\![f_i]\!]^{\sigma[[\![t]\!]^{\sigma}/d, w_i/e_i]}$ is analogous to the case for c_i.

Finally, we prove that $[\![g_i]\!]^{\sigma[[\![t]\!]^\sigma/d,w_i/e_i]}\ R\ [\![\mathsf{cp}(\mathsf{unfold}(g_i),d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^\sigma/\mathsf{params}(d),w_i/e_i]}$.

$[\![g_i]\!]^{\sigma[[\![t]\!]^\sigma/d,w_i/e_i]}$

$=$ {By assumption, every w_i has closed term t_i s.t. $[\![t_i]\!]=w_i$; $[\![t_i]\!]=[\![t_i]\!]^\sigma$ since t_i closed}

$[\![g_i]\!]^{\sigma[[\![t]\!]^\sigma/d,[\![t_i]\!]^\sigma/e_i]}$

$=$ {Lemma 2}

$[\![g_i[d:=t,e_i:=t_i]]\!]^\sigma$

R {Definition of R}

$[\![\mathsf{unfold}(g_i[d:=t,e_i:=t_i])]\!]^\sigma$

$=$ {Definitions of substitution, unfold}

$[\![\mathsf{unfold}(g_i)[d:=t,e_i:=t_i]]\!]^\sigma$

$=$ {Lemma 2}

$[\![\mathsf{unfold}(g_i)]\!]^{\sigma[[\![t]\!]^\sigma/d,[\![t_i]\!]^\sigma/e_i]}$

$=$ {Corollary 2}

$[\![\mathsf{unfold}(\mathsf{cp}(g_i,d))]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^\sigma/\mathsf{params}(d),[\![t_i]\!]^\sigma/e_i]}$

$=$ {$[\![t_i]\!]^\sigma=w_i$, see first step in derivation}

$[\![\mathsf{unfold}(\mathsf{cp}(g_i,d))]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^\sigma/\mathsf{params}(d),w_i/e_i]}$

$=$ {Lemma 9}

$[\![\mathsf{cp}(\mathsf{unfold}(g_i),d)]\!]^{\sigma[[\![\mathsf{unfold}(t)]\!]^\sigma/\mathsf{params}(d),w_i/e_i]}$

This concludes the first direction of the proof that both processes are strongly bisimilar. The other direction is symmetric. □

Corollary 3. *Consider the LPEs from Theorem 1. Then it holds that* $P'(\mathsf{unfold}(e)) \leftrightarroweq P'(\mathsf{unfold}(e))$.

Proof. Theorem 1 shows that $P(e) \leftrightarroweq P'(\mathsf{unfold}(e))$. Using similar arguments we can also show that $P'(\mathsf{unfold}(e)) \leftrightarroweq P(e)$. The result then follows from transitivity of strong bisimulation. □

We next show that also the variant where global variables are replaced with fresh global variables preserves strong bisimulation.

Theorem 2. *Let* $\mathcal{D}=(\Sigma,E)$*, and consider LPE* $L=(\mathcal{D},\mathcal{X}_g,P,e)$*, where* P *is defined as:*

$$P(d:D)=\sum_{i\in I}\sum_{e_i:E_i} c_i \to a_i(f_i)\cdot P(g_i)$$

Also consider the result of unfolding parameter d *using the global variables optimization from Definition 11*

$$L'=(\mathcal{D}',\mathcal{X}'_g,P',\mathsf{unfold}_g(e)[dc:=C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]_{dc\in\mathcal{X}_r})$$

where P' *is defined as*

$$\begin{aligned}&P'(\mathsf{params}(d))\\&=\sum_{i\in I}\sum_{e_i:E_i}\mathsf{cp}(c_i,d)[dc:=C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]_{dc\in\mathcal{X}_r}\\&\qquad\to a_i(\mathsf{cp}(f_i,d)[dc:=C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]_{dc\in\mathcal{X}_r})\\&\qquad\qquad\cdot P'(\mathsf{cp}(\mathsf{unfold}_g(g_i),d)[dc:=C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]_{dc\in\mathcal{X}_r})\end{aligned}$$

Then $P(e)\leftrightarroweq P'(\mathsf{unfold}_g(e)[dc:=C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]_{dc\in\mathcal{X}_r})$.

Proof. Fix arbitrary valuation σ. Let $\mathcal{X}_r$ be as in Definition 11. We define valuation σ_r as follows:

$$\sigma_r=\sigma[[\![C_D(dc_e,f_0(dc^1_{f_0},\dots,dc^{m_0}_{f_0}),\dots,f_n(dc^1_{f_n},\dots,dc^{m_n}_{f_n}))]\!]^\sigma/dc]_{dc\in\mathcal{X}_r}$$

Also, define relation R such that for every term t of sort D,

$$[\![t]\!]^{\sigma_r} \; R \; [\![\mathsf{unfold}_g(t)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma}$$

Before we continue our proof, note that it follows immediately from Lemma 2 and the definition of σ_r that

$$[\![\mathsf{unfold}_g(t)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma} = [\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}$$

So, an equivalent definition of R is, for any t of sort D,

$$[\![t]\!]^{\sigma_r} \; R \; [\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}$$

We prove that R is a strong bisimulation relation.

To this end, fix arbitrary term t such that $[\![t]\!]^{\sigma_r} \; R \; [\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}$, and fix arbitrary $i \in I$, value $w_i \in M_{E_i}$ such that $[\![c_i]\!]^{\sigma_r[[\![t]\!]^{\sigma_r}/d, w_i/e_i]} = \mathbf{true}$. So, the condition of summand i is satisfied. We show that summand i and w_i witness the transfer condition.

First, we show that

$$\begin{aligned}&[\![\mathsf{cp}(c_i, d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma_r[[\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}/\mathsf{params}(d), w_i/e_i]}\\ =&[\![c_i]\!]^{\sigma_r[[\![t]\!]^{\sigma_r}/d, w_i/e_i]}\end{aligned}$$

It then follows immediately that the left hand side of this equality is **true** as well.

For the sake of brevity, in the remainder of the proof, we write $\sigma_{r,i}$ for $\sigma_r[[\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}/\mathsf{params}(d), w_i/e_i]$. The derivation is as follows.

$$\begin{aligned}&[\![\mathsf{cp}(c_i, d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma_{r,i}}\\ &= \{\text{Lemma 2}\}\\ &[\![\mathsf{cp}(c_i, d)]\!]^{\sigma_{r,i}[[\![C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]\!]^{\sigma_{r,i}}/dc]_{dc \in \mathcal{X}_r}}\\ &= \{\mathsf{params}(d) \text{ and } e_i \text{ are not global variables}\}\\ &[\![\mathsf{cp}(c_i, d)]\!]^{\sigma_{r,i}[[\![C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]\!]^{\sigma_r}/dc]_{dc \in \mathcal{X}_r}}\\ &= \{\text{Fresh global variables are not in } \mathcal{X}_r\}\\ &[\![\mathsf{cp}(c_i, d)]\!]^{\sigma_{r,i}[[\![C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]\!]^{\sigma}/dc]_{dc \in \mathcal{X}_r}}\\ &= \{\text{Definition of } \sigma_r;\ dc \text{ not in } \mathsf{params}(d) \text{ or } e_i\}\\ &[\![\mathsf{cp}(c_i, d)]\!]^{\sigma_{r,i}}\\ &= \{\mathsf{params}(d) \text{ fresh, Corollary 1, analogous to case } c_i \text{ in the proof of Theorem 1}\}\\ &[\![c_i]\!]^{\sigma_r[[\![t]\!]^{\sigma_r}/d, w_i/e_i]}\end{aligned}$$

The proofs for a_i and f_i are analogous to the proof of Theorem 1 and that of c_i above. So we finally need to prove that

$$\begin{aligned}&[\![g_i]\!]^{\sigma_r[[\![t]\!]^{\sigma_r}/d, w_i/e_i]}\\ &R\\ &[\![\mathsf{cp}(\mathsf{unfold}_g(g_i), d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma_{r,i}}\end{aligned}$$

Using a similar line of reasoning as the case for c_i, it follows that

$$\begin{aligned}&[\![\mathsf{cp}(\mathsf{unfold}_g(g_i), d)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma_{r,i}}\\ &= [\![\mathsf{cp}(\mathsf{unfold}_g(g_i), d)]\!]^{\sigma_{r,i}}\end{aligned}$$

So, using $\sigma_{r,i} = \sigma_r[[\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}/\mathsf{params}(d), w_i/e_i]$, it remains to show that

$$[\![g_i]\!]^{\sigma_r[[\![t]\!]^{\sigma_r}/d, w_i/e_i]} \; R \; [\![\mathsf{cp}(\mathsf{unfold}_g(g_i), d)]\!]^{\sigma_r[[\![\mathsf{unfold}_g(t)]\!]^{\sigma_r}/\mathsf{params}(d), w_i/e_i]}$$

The proof of this is analogous to that of the case for g_i in the proof of Theorem 1.

This concludes the first direction of the proof that R is a strong bisimulation relation. The other direction is symmetric.

Finally we show that for all valuations σ' and σ, there exists a bisimulation relation $R_{P,P'}$ such that $[\![e]\!]^{\sigma'} \; R_{P,P'} \; [\![\mathsf{unfold}_g(e)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma}$. Using σ_r and R as defined above, we have that

$$[\![e]\!]^{\sigma_r} \; R \; [\![\mathsf{unfold}_g(e)[dc := C_D(dc_e, f_0(dc^1_{f_0}, \ldots, dc^{m_0}_{f_0}), \ldots, f_n(dc^1_{f_n}, \ldots, dc^{m_n}_{f_n}))]_{dc \in \mathcal{X}_r}]\!]^{\sigma}$$

As $P(e) \leftrightarroweq P(e)$, it also follows that there is a bisimulation relation $R_{P,P}$ such that $[\![e]\!]^{\sigma}\ R_{P,P}\ [\![e]\!]^{\sigma_r}$. From this it follows that $R_{P,P'} = R_{P,P} \circ R$ is a strong bisimulation. This concludes the proof. □

Using similar arguments as before, it follows that, after unfolding it still remains the case that global variables do not affect the behavior of the processes.

Corollary 4. *Consider the LPEs from Theorem 2. Then it holds that*

$$P'(\mathsf{unfold}_g(e)[dc := C_D(dc_e, f_0(dc_{f_0}^1, \ldots, dc_{f_0}^{m_0}), \ldots, f_n(dc_{f_n}^1, \ldots, dc_{f_n}^{m_n}))]_{dc \in \mathcal{X}_r})$$

$$\leftrightarroweq$$

$$P'(\mathsf{unfold}_g(e)[dc := C_D(dc_e, f_0(dc_{f_0}^1, \ldots, dc_{f_0}^{m_0}), \ldots, f_n(dc_{f_n}^1, \ldots, dc_{f_n}^{m_n}))]_{dc \in \mathcal{X}_r})$$

6. Experiments

The original parameter unfolding technique from [22] has been available in the tool `lpsparunfold` in the mCRL2 toolset [14] for over a decade. We have extended the C++ implementation with the ideas described in this article. The tool allows selecting which parameters to unfold, and the number of times a 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.

In previous experiments reported in [26], we compared the original definition of parameter unfolding from Groote and Lisser [22] to our description in which distribution laws, pattern match unfolding and the global variables optimization were always enabled. We compared the effect of default and alternative case placement in this setting. In this article we extend the experiment, and focus on the effect of (default vs alternative) case placement, pattern match unfolding and global variables replacement.[2] We run all eight combinations of these options to allow studying the effectiveness of the single optimizations.

By default, the tool `lpsparunfold` performs parameter unfolding with distribution laws, pattern match unfolding and global variables replacement using default case placement. Command line arguments `-x` can be used to switch off pattern match unfolding, and `-g` disables replacement of global variables. 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 nine workflows:

- `standard`: standard static analysis workflow: instantiate finite summations, eliminate constant and redundant parameters and superfluous summation variables [22] (using the mCRL2 tools `lpssuminst`, `lpsconstelm`, `lpsparelm` and `lpssumelm`). Finally, perform symbolic reachability (`lpsreach`). No parameter unfolding is applied.
- `cp-x-g`: perform parameter unfolding with default case placement (`cp`), where pattern matching functions are not unfolded (`-x`) and global variables are not replaced (`-g`). After that, apply the steps from `standard`.
- `cp-x`: perform parameter unfolding with our extension for global variables with default case placement (`cp`), where pattern matching functions are not unfolded (`-x`). After that, apply the steps from `standard`.
- `cp-g`: perform parameter unfolding with pattern matching rules with default case placement (`cp`), where global variables are not replaced (`-g`). After that, apply the steps from `standard`.
- `cp`: perform parameter unfolding with our extension for global variables and pattern matching rules with default case placement (`cp`). After that, apply the steps from `standard`.
- `acp-x-g`, `acp-x`, `acp-g`, and `acp`: these are the same as the workflows for `cp`, but use alternative case placement instead of default case placement.

The workflows are executed on various mCRL2 specifications, including our running example (onoff). We consider 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. The *sliding window protocol* [50], that forms the basis of the TCP protocol used for reliable in-order delivery of packets, as it occurs in [25], with window size n and m messages (swp-n-m) for different values of n and m is a representative of communication protocols. For this 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) [51] and of an industrial lift (left-lift); the protocol negotiating a *service level agreement* (sla) between two parties communicating via message passing along reliable channels encoded using fixed length lists [52]; a model of the Workload Management System (wms) of the DIRAC Community Grid Solution for the LHCb experiment at CERN [53]; two configurations of the model of session setup of the IEEE 11073-20601 standard for communication between personal health devices, with two unidirectional buffers of size n for communication (ieee-11073-n) [54]. Note that the use of complex data structures for industrial case studies is wide-spread, allowing the creation of concise and elegant models.

[2] Experiments run one single time over each of the specifications show that adding distribution laws never negatively effects the running time, we hence always include them in our experiments.

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.

Model	Size (# states)	Time (s)								
		`standard`	`cp-x-g`	`cp-x`	`cp-g`	`cp`	`acp-x-g`	`acp-x`	`acp-g`	`acp`
cylinder	1 593 209	27.0	**15.9**	**15.6**	**15.9**	**15.6**	**15.9**	**15.8**	**16.0**	**15.8**
fourinarow3-4	12 305	62.4	**1.4**	**1.4**	**1.4**	**1.4**	**1.4**	**1.4**	**1.4**	**1.4**
fourinarow3-5	(*)171 243	t-o	**9.0**	**9.1**	**9.0**	**9.0**	**9.2**	**9.2**	**9.2**	**9.2**
fourinarow4-3	6 214	14.5	**1.0**	**1.1**	**1.1**	**1.0**	**1.1**	**1.1**	**1.1**	**1.1**
fourinarow4-4	(*)187 928	t-o	**9.3**	**9.3**	**9.3**	**9.3**	**9.6**	**9.6**	**9.4**	**9.4**
fourinarow4-5	(*)5 464 759	t-o	**312.6**	**311.8**	**313.0**	**312.4**	**316.1**	**315.2**	**316.5**	**316.0**
fourinarow5-3	44 131	842.4	**3.1**	**3.1**	**3.1**	**3.1**	**3.2**	**3.1**	**3.1**	**3.2**
fourinarow5-4	(*)2 788 682	t-o	**146.2**	**146.3**	**145.9**	**146.1**	**149.4**	**148.6**	**148.8**	**147.8**
onoff	(*)3	t-o	t-o	t-o	t-o	t-o	**0.0**	**0.0**	**0.0**	0.1
sla7	7 918	**2.0**	2.5	2.5	2.6	2.5	2.6	2.5	2.5	2.5
sla10	238 931	30.4	17.2	17.1	16.5	16.4	**14.2**	**14.2**	**14.3**	**14.3**
sla13	(*)6 693 054	t-o	383.7	385.5	375.4	375.5	**301.0**	**295.9**	**304.9**	**291.9**
swp2-2	14 064	**1.2**	**1.3**	1.4	**1.3**	**1.3**	**1.3**	**1.3**	**1.2**	**1.3**
swp2-4	140 352	**2.3**	**2.6**	**2.6**	**2.4**	**2.5**	**2.6**	**2.6**	**2.4**	**2.4**
swp2-6	598 320	**3.4**	3.6	3.7	**3.3**	**3.2**	3.6	3.6	**3.3**	**3.2**
swp2-8	1 731 840	**4.1**	4.8	4.8	**4.0**	**4.3**	4.8	4.8	**4.0**	**4.0**
swp4-2	2 589 056	**5.8**	9.5	9.5	7.2	7.4	9.8	9.4	7.2	7.6
swp4-4	292 878 336	130.9	163.1	161.5	**100.2**	**100.8**	162.5	162.1	**100.2**	**100.5**
swp4-6	5 729 304 960	3 040.8	1 071.1	1 071.4	**669.4**	**668.5**	1 075.1	1 073.4	**669.5**	**671.0**
swp4-8	(*)50 128 191 488	t-o	t-o	t-o	**2 746.3**	**2 740.6**	t-o	t-o	**2 754.5**	**2 745.7**
tictactoe3-3	5 479	12.3	8.0	**1.5**	4.7	**1.4**	2.3	**1.4**	2.3	**1.4**
wms	155 034 776	**17.4**	**17.7**	**17.5**	**17.6**	**17.6**	**17.5**	**17.5**	**17.4**	**17.6**
ieee-11073-2	9 874	**3.7**	**3.9**	**3.8**	**3.8**	**3.8**	**3.9**	**3.9**	**3.9**	**3.9**
ieee-11073-3	54 147	12.7	**8.1**	**8.0**	**8.4**	**8.1**	**7.8**	**7.9**	**7.8**	**7.8**
left-lift	13 212 954 983	**2 145.4**	2 780.2	2 778.2	2 770.0	2 783.1	2 780.9	2 764.4	2 844.3	2 770.6

A reproduction package including all tool versions and mCRL2 specifications used is available from https://doi.org/10.5281/zenodo.12705700, also in [55]. The used mCRL2 version is 202307.1.

6.1. Results

All experiments were run 10 times, on a machine with 4 Intel 6136 CPUs and 3TB of RAM, running Ubuntu 20.04. The results are presented in Table 1. We used a time-out of 1 hour (3600 seconds) and a memory limit of 64 GB. Every experiment is limited to the use of a single thread. We report the size of the explored state space in number of states and the mean total running time of 10 runs in seconds. The reported running time is only for symbolic reachability. The reason for this is that the running time for standard static analysis tools and parameter unfolding are insignificant compared to that of symbolic reachability. For each model we report the size, in terms of the number of states, only once in the table. This is because, for a single model, the workflows that do not timeout result in the same state space. For all models, apart from *onoff*, parameter unfolding does not enable other static analysis tools to achieve a reduction of the state space size. Therefore, the size of the state space is the same for all the workflows that terminate. If a workflow times out, 't-o', no size for the state space is reported. With the (*) symbol we indicate that the reported size is for the workflows that did not result in a timeout. For example, the *fourinarow3-5* model has a state space of size 171 243 for all workflows but `standard`, for the latter no size is reported since the workflow times out.

For each model, we highlight the fastest runs as follows. Let m be the running time of the fastest run. We highlight in bold all running times that are at most 10% higher than m. For most of the experiments, the standard deviation is below 10% of the mean.[3]

6.2. Discussion

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 *tic-tac-toe*, i.e., in Table 1 workflows `cp-x`, `cp`, `acp-x` and `acp` have a lower running time than the other workflows. The simplifications for pattern matching rules show a reduction in the running time for the sliding window protocol (swp). For model *swp4-8*, in Table 1, workflows `cp-g`, `cp`, `acp-g` and `acp` have a running time of ~ 45 minutes while the other workflows result in a timeout. 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, mostly for larger configurations as it is shown by the results for *sla13*.

[3] The cases where the standard deviation exceeds 10% of the mean, with their standard deviation, are: *sla-13* `acp-g`: 37.8, *swp2-6* `standard`: 0.6, *swp2-8* `cp`: 0.8, *swp4-2* `cp`: 0.8, `acp-x-g`: 1.3, `acp`: 1.4, *11073-3* `acp-g`: 1.2.

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 terms in the LPE, this is not observed in our experiments.

In some cases, our parameter unfolding techniques do not manage to improve the results of static analysis. This typically happens when the unfolded data structures do not have a fixed size. In Table 1, models *ieee-11073-n* and *left-lift* have data structures with a dynamic size which, as clearly shown by the results of *left-lift*, negatively affects our parameter unfolding techniques. We demonstrate this in the below example, which is inspired by the *ieee-11073* model.

Example 18. Consider the following process that models a buffer that can store up to two natural numbers:

$$\begin{aligned}&\textbf{proc}\quad Buf(l : List(Nat), broken : Bool)\\ &\qquad = \Sigma_{n:Nat}(\#l \leq 2 \wedge \neg broken) \rightarrow receive(n).Buf(l \lhd n, broken)\\ &\qquad +(l \not\approx [\,] \wedge \neg broken) \rightarrow send(head(l)).Buf(tail(l), broken)\\ &\qquad +destroy.Buf([\,], true);\\ &\textbf{init}\quad Buf([\,], false);\end{aligned}$$

Here, $\lhd$ is a mapping that appends a single element to the back of a list and $\#l$ is the length of list l (the corresponding equations in the data specification are straightforwardly defined using recursion). This buffer operates in a *first-in first-out* manner: when a number is received it is placed at the back of the list and the number at the head of the list can be sent. In case the buffer is destroyed in an accident, it ceases all operations.

We unfold the parameter $l : List(Nat)$ twice with alternative case placement and pattern match unfolding. The new process now has the following parameters:

$$Buf'(e_l^1, e_l^2 : U_{List(Nat)}, p_1, p_2 : Nat, q : List(Nat), broken : Bool)$$

Parameters e_l^1 and e_l^2 indicate whether the first and second positions of the original list are occupied, respectively. The corresponding values are stored in p_1 and p_2, while the remainder of the list is stored in parameter q. Note that $q \approx [\,]$ is an invariant of Buf', since the original list never grows beyond size 2.

The difficulty of deducing this invariant lies in the first summand, where the list is extended. After unfolding and rewriting, this summand is as follows (for conciseness we refer to $C_{List(Nat)}$ simply as C).

$$\begin{aligned}&\Sigma_{n:Nat}(C(e_l^2, true, C(e_l^1, true, false)) \wedge \neg broken) \rightarrow receive(n).\\ &\quad Buf'(\overline{c}_{\triangleright}, e_l^1, C(e_l^1, n, p_1), C(e_l^2, C(e_l^1, 0, n), C(e_l^1, 0, p_2)), C(e_l^2, [\,], C(e^1, [\,], q \lhd n)))\end{aligned}$$

The invariant can only be deduced if we can show that $[\![C(e_l^2, true, C(e_l^1, true, false)) \wedge \neg broken]\!]^\sigma$ implies $[\![C(e_l^2, [\,], C(e^1, [\,], q \lhd n)) \approx [\,]]\!]^\sigma$ for all σ. This requires a careful analysis of the equations in the data specification, something our static analysis tools are currently not capable of.

Similar to the example, in the case of *ieee-11073* our static analysis tools are not able to deduce invariants of the form $q \approx [\,]$; the case of *left-lift* is comparable. Despite this unused potential, parameter unfolding still helps speed up symbolic exploration in the case of *ieee-11073* with buffer size 3.

Overall, the results show that generally pattern match unfolding and the unfolding of global variables have a positive effect on the performance. Our experiments show that pattern match unfolding and the unfolding of global variables are safe to be used by default. They never have a significant negative effect on the performance. Case placement and alternative case placement are often close in terms of performance, where alternative case placement is potentially more powerful. The models where alternative case placement is clearly beneficial are those where distribution of function symbols over case functions can be exploited for simplification. In particular when there are many comparisons, e.g., using $\approx, <, >, \ldots$, with a constant, alternative case placement can speed up the running time, or, as in our running example, reduce the size of the state space. The presence of (many) such comparisons can be deduced by inspecting the structure of the specification. Since alternative case placement is susceptible to exponential blow-up, even though we did not observe such blow-up in our experiments, we keep it as an option to the tool, but refrain from making it the default. Unfortunately, it is not possible to detect *a priori* which specification would lead to a blow-up.

7. Conclusion

In this article we have described a general approach to unfold state variables in a specification of a distributed system. We have presented our technique, based on Groote and Lisser's parameter unfolding [22], for models that describe the behavior of a system using state variables and (terms over) algebraic data types. In particular, we have presented (alternative) case placement and pattern match unfolding in detail. In the context of mCRL2, we have added global variables unfolding. We have proven the correctness of each of the transformations.

We have experimentally evaluated the effect of case placement and alternative case placement, pattern match unfolding and global variables unfolding in mCRL2. In general, we observe that, even if the size of the state space is not reduced, the unfolding of state

variables improves the performance of symbolic reachability. Pattern match unfolding and global variables unfolding typically have a positive effect; the performance of case placement and alternative case placement are mostly comparable.

We believe the effect of `lpsparunfold` should be investigated in relation to other static analysis techniques such as dead variable analysis [23]. Together these have the potential to speed up the model checking of industrial systems, e.g., described by OIL models [48] and Cordis models [51] using mCRL2. The effect of `lpsparunfold` could also be investigated in the context of PBESs.

The general nature of our techniques also warrants further study in the context of other formalisms that use algebraic data types. Examples are *constrained Horn clause* (CHC) solvers [56] and compilers for functional programming languages.

CRediT authorship contribution statement

Anna Stramaglia: Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Jeroen J.A. Keiren:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis. **Thomas Neele:** Writing – review & editing, Writing – original draft, Software, Investigation, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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.

This research was supported by the MACHINAIDE project (ITEA3, No. 18030), and the National Growth Fund through the Dutch 6G flagship project "Future Network Services".

References

[1] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580, https://doi.org/10.1145/363235.363259.
[2] S.S. Ishtiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, in: POPL, ACM, 2001, pp. 14–26, https://doi.org/10.1145/360204.375719.
[3] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: LICS, IEEE Computer Society, 2002, pp. 55–74, https://doi.org/10.1109/LICS.2002.1029817.
[4] J.C.M. Baeten, T. Basten, M.A. Reniers, Process Algebra: Equational Theories of Communicating Processes, Cambridge Tracts in Theoretical Computer Science, vol. 50, Cambridge University Press, Cambridge, New York, 2010.
[5] U. Khedker, A. Sanyal, B. Sathe, Data Flow Analysis: Theory and Practice, CRC Press, Boca Raton, 2017, https://doi.org/10.1201/9780849332517.
[6] C. Baier, J.P. Katoen, Principles of Model Checking, MIT Press, 2008.
[7] E.M. Clarke, O. Grumberg, D. Kroening, D.A. Peled, H. Veith, Model Checking, 2nd edition, MIT Press, 2018.
[8] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, 4th edition, Addison-Wesley, Boston, Mass. Munich, 2008.
[9] Z. Baranová, J. Barnat, K. Kejstová, T. Kucera, H. Lauko, J. Mrázek, P. Rockai, V. Still, Model checking of C and C++ with DIVINE 4, in: ATVA, in: Lecture Notes in Computer Science, vol. 10482, Springer, 2017, pp. 201–207, https://doi.org/10.1007/978-3-319-68167-2_14.
[10] F. Merz, S. Falke, C. Sinz, LLBMC: bounded model checking of C and C++ programs using a compiler IR, in: VSTTE, in: Lecture Notes in Computer Science, vol. 7152, Springer, 2012, pp. 146–161, https://doi.org/10.1007/978-3-642-27705-4_12.
[11] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: a toolbox for the construction and analysis of distributed processes, Int. J. Softw. Tools Technol. Transf. 15 (2) (2013) 89–107, https://doi.org/10.1007/S10009-012-0244-Z.
[12] R. van Beusekom, J.F. Groote, P.F. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, T.A.C. Willemse, Formalising the Dezyne modelling language in mCRL2, in: FMICS-AVoCS, in: Lecture Notes in Computer Science, vol. 10471, Springer, 2017, pp. 217–233, https://doi.org/10.1007/978-3-319-67113-0_14.
[13] T. Gibson-Robinson, P.J. Armstrong, A. Boulgakov, A.W. Roscoe, FDR3 - a modern refinement checker for CSP, in: TACAS, in: Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 187–201, https://doi.org/10.1007/978-3-642-54862-8_13.
[14] O. Bunte, J.F. Groote, J.J.A. Keiren, M. Laveaux, T. Neele, E.P. de Vink, W. Wesselink, A. Wijs, T.A.C. Willemse, The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: TACAS (2), in: Lecture Notes in Computer Science, vol. 11428, Springer, 2019, pp. 21–39, https://doi.org/10.1007/978-3-030-17465-1_2.
[15] E.M. Clarke, My 27-year quest to overcome the state explosion problem, in: LICS, IEEE Computer Society, 2009, p. 3, https://doi.org/10.1109/LICS.2009.42.
[16] Y. Hwong, J.J.A. Keiren, V.J.J. Kusters, S.J.J. Leemans, T.A.C. Willemse, Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider, Sci. Comput. Program. 78 (12) (2013) 2435–2452, https://doi.org/10.1016/J.SCICO.2012.11.009.
[17] A. Valmari, The state explosion problem, in: W. Reisig, G. Rozenberg (Eds.), Lectures on Petri Nets I: Basic Models: Advances in Petri Nets, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 429–528, https://doi.org/10.1007/3-540-65306-6_21.
[18] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems - an Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032, Springer, 1996, https://doi.org/10.1007/3-540-60761-7.
[19] D.A. Peled, All from one, one for all: on model checking using representatives, in: CAV, in: Lecture Notes in Computer Science, vol. 697, Springer, 1993, pp. 409–423, https://doi.org/10.1007/3-540-56922-7_34.
[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inf. Comput. 98 (2) (1992) 142–170, https://doi.org/10.1016/0890-5401(92)90017-A.
[21] K.L. McMillan, Symbolic Model Checking, Springer US, Boston, MA, 1993, https://doi.org/10.1007/978-1-4615-3190-6.
[22] J.F. Groote, B. Lisser, Computer assisted manipulation of algebraic process specifications, Tech. Rep. SEN-R0117, CWI, Jan. 2001, https://ir.cwi.nl/pub/4326/.
[23] J. van de Pol, M. Timmer, State space reduction of linear processes using control flow reconstruction, in: ATVA, in: Lecture Notes in Computer Science, vol. 5799, Springer, 2009, pp. 54–68, https://doi.org/10.1007/978-3-642-04761-9_5.

[24] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, T. van Dijk, LTSmin: high-performance language-independent model checking, in: TACAS, in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 692–707, https://doi.org/10.1007/978-3-662-46681-0_61.
[25] J.F. Groote, M.R. Mousavi, Modeling and Analysis of Communicating Systems, MIT Press, 2014.
[26] A. Stramaglia, J.J.A. Keiren, T. Neele, Simplifying process parameters by unfolding algebraic data types, in: ICTAC, in: Lecture Notes in Computer Science, vol. 14446, Springer, 2023, pp. 399–416, https://doi.org/10.1007/978-3-031-47963-2_24.
[27] M. Wirsing, Structured algebraic specifications: a kernel language, Theor. Comput. Sci. 42 (1986) 123–249, https://doi.org/10.1016/0304-3975(86)90051-4.
[28] J.V. Guttag, The specification and application of programming of abstract data types, Ph.D. thesis, University of Toronto, Toronto, Sep. 1975, http://archive.org/details/technicalreportc59univ.
[29] J.V. Guttag, J.J. Horning, The algebraic specification of abstract data types, Acta Inform. 10 (1978) 27–52, https://doi.org/10.1007/BF00260922.
[30] D. Sannella, A. Tarlecki, Foundations of Algebraic Specification and Formal Software Development, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2012.
[31] S. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, J. van de Pol, μCRL: a toolset for analysing algebraic specifications, in: CAV, in: Lecture Notes in Computer Science, vol. 2102, Springer, 2001, pp. 250–254, https://doi.org/10.1007/3-540-44585-4_23.
[32] J.F. Groote, T.A.C. Willemse, Parameterised Boolean equation systems, Theor. Comput. Sci. 343 (3) (2005) 332–369, https://doi.org/10.1016/J.TCS.2005.06.016.
[33] S. Orzan, W. Wesselink, T.A.C. Willemse, Static analysis techniques for parameterised Boolean equation systems, in: TACAS, in: Lecture Notes in Computer Science, vol. 5505, Springer, 2009, pp. 230–245, https://doi.org/10.1007/978-3-642-00768-2_22.
[34] J.J.A. Keiren, W. Wesselink, T.A.C. Willemse, Liveness analysis for parameterised Boolean equation systems, in: ATVA, in: Lecture Notes in Computer Science, vol. 8837, Springer, 2014, pp. 219–234, https://doi.org/10.1007/978-3-319-11936-6_16.
[35] T. Neele, (Re)moving quantifiers to simplify parameterised Boolean equation systems, in: ARQNL@IJCAR, in: CEUR Workshop Proceedings, vol. 3326, 2022, pp. 64–80, CEUR-WS.org.
[36] R. Melton, D. Dill, C. Ip, U. Stern, Murphi annotated reference manual, release 3.1, https://github.com/melver/cmurphi/blob/master/doc/User.Manual, 1996.
[37] H. Garavel, W. Serwe, State space reduction for process algebra specifications, Theor. Comput. Sci. 351 (2) (2006) 131–145.
[38] G.H. Slomp, Reducing UPPAAL models through control flow analysis, MSc thesis, University of Twente, Enschede, 2010, https://essay.utwente.nl/60021/.
[39] C. Dubslaff, A. Morozov, C. Baier, K. Janschek, Reduction methods on probabilistic control-flow programs for reliability analysis, CoRR, arXiv:2004.06637, 2020.
[40] E.M. Clarke, E.A. Emerson, S. Jha, A.P. Sistla, Symmetry reductions in model checking, in: CAV, in: Lecture Notes in Computer Science, vol. 1427, Springer, 1998, pp. 147–158.
[41] T. Gibson-Robinson, G. Lowe, Symmetry reduction in CSP model checking, Int. J. Softw. Tools Technol. Transf. 21 (5) (2019) 567–605, https://doi.org/10.1007/S10009-019-00516-4.
[42] T. van Dijk, J. van de Pol, Sylvan: multi-core decision diagrams, in: TACAS, in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 677–691, https://doi.org/10.1007/978-3-662-46681-0_60.
[43] S. Blom, J. van de Pol, Symbolic reachability for process algebras with recursive data types, in: ICTAC, in: Lecture Notes in Computer Science, vol. 5160, Springer, 2008, pp. 81–95, https://doi.org/10.1007/978-3-540-85762-4_6.
[44] J. Meijer, G. Kant, S. Blom, J. van de Pol, Read, write and copy dependencies for symbolic model checking, in: Haifa Verification Conference, in: Lecture Notes in Computer Science, vol. 8855, Springer, 2014, pp. 204–219, https://doi.org/10.1007/978-3-319-13338-6_16.
[45] T. van Dijk, J. van de Pol, Sylvan: multi-core framework for decision diagrams, Int. J. Softw. Tools Technol. Transf. 19 (6) (2017) 675–696, https://doi.org/10.1007/S10009-016-0433-2.
[46] B. Steffen, Data flow analysis as model checking, in: TACS, in: Lecture Notes in Computer Science, vol. 526, Springer, 1991, pp. 346–365.
[47] M. Gallardo, C. Joubert, P. Merino, On-the-fly data flow analysis based on verification technology, in: COCV@ETAPS, in: Electronic Notes in Theoretical Computer Science, vol. 190, Elsevier, 2007, pp. 33–48.
[48] O. Bunte, L.C.M. van Gool, T.A.C. Willemse, Formal verification of OIL component specifications using mCRL2, in: FMICS, in: Lecture Notes in Computer Science, vol. 12327, Springer, 2020, pp. 231–251, https://doi.org/10.1007/978-3-030-58298-2_10.
[49] D.M.R. Park, Concurrency and automata on infinite sequences, in: Theoretical Computer Science, in: Lecture Notes in Computer Science, vol. 104, Springer, 1981, pp. 167–183, https://doi.org/10.1007/BFB0017309.
[50] V.G. Cerf, R.E. Kahn, A protocol for packet network intercommunication, IEEE Trans. Commun. 22 (5) (1974) 637–648, https://doi.org/10.1109/TCOM.1974.1092259.
[51] A. Stramaglia, J.J.A. Keiren, Formal verification of an industrial UML-like model using mCRL2, in: FMICS, in: Lecture Notes in Computer Science, vol. 13487, Springer, 2022, pp. 86–102, https://doi.org/10.1007/978-3-031-15008-1_7.
[52] J.F. Groote, T.A.C. Willemse, 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.
[53] D. Remenska, T.A.C. Willemse, K. Verstoep, J. Templon, H.E. Bal, Using model checking to analyze the system behavior of the LHC production grid, Future Gener. Comput. Syst. 29 (8) (2013) 2239–2251, https://doi.org/10.1016/J.FUTURE.2013.06.004.
[54] J.J.A. Keiren, M. Klabbers, Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 53 (2012), https://doi.org/10.14279/TUJ.ECEASST.53.793.
[55] A. Stramaglia, J.J.A. Keiren, T. Neele, Artifact for 'Unfolding State Variables Improves Model Checking', https://doi.org/10.5281/zenodo.12705700, 2024.
[56] N.S. Bjørner, A. Gurfinkel, K.L. McMillan, A. Rybalchenko, Horn clause solvers for program verification, in: L.D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, W. Schulte (Eds.), Fields of Logic and Computation II, in: LNCS, vol. 9300, Springer, 2015, pp. 24–51, https://doi.org/10.1007/978-3-319-23534-9_2.