# Finding Compact Proofs for Infinite-Data Parameterised Boolean Equation Systems

Thomas Neele, Tim A.C. Willemse, Jan Friso Groote

*Eindhoven University of Technology, The Netherlands*

**Abstract**

Parameterised Boolean Equation Systems (PBESs) can be used to represent many different kinds of decision problems. Most notably, model checking and equivalence problems can be encoded in a PBES. Traditional techniques to solve PBESs, such as instantiation techniques, cannot deal with PBESs with an infinite data domain. We propose an approach that can solve PBESs with infinite data by computing the bisimulation quotient of the underlying graph structure. Furthermore, we show how this technique can be improved by repeatedly searching for finite proofs. We also apply knowledge of intermediate solutions in an early termination heuristic. Unlike existing approaches, our technique is not restricted to subfragments of PBESs. Compared to similar procedures that operate on behavioural models, our technique is also more general: it is not restricted to model checking with finite action sets. Experimental results show that our ideas work well in practice and support a wider range of models and properties than state-of-the-art techniques.

*Keywords:* symbolic model checking, modal mu-calculus, parameterised Boolean equation system, bisimulation, infinite state system

## 1. Introduction

A *parameterised Boolean equation system* (PBES) [1] is a sequence of fixpoint equations over first-order logic formulae. Many different types of decision problems can be encoded in a PBES, for example model checking problems, as implemented by the toolsets CADP [2] and mCRL2 [3], and equivalence queries [4]. Model checking problems using the modal mu-calculus with data and time as well as CTL*/LTL formulas can be translated efficiently into PBESs. The answer to the encoded problem can be found by (partially) solving the PBES. In this way, PBESs and techniques to solve them are useful in the analysis of component systems.

Although finding the solution of a PBES is undecidable in general, in practice several efficient approaches to solve PBESs exist. Most notably, some PBESs can be solved efficiently by first simplifying them—if needed—using static analysis techniques [5, 6], instantiating them to finite *Boolean equation systems* (BESs) and subsequently solving these BESs [7]. However, for many types of problems, the corresponding PBES contains data taken from domains that are infinite. For example, a PBES encoding the mutual exclusion property for Lamport's bakery protocol requires data variables ranging over natural numbers. Similarly, PBESs encoding model checking problems for timed or hybrid systems, typically modelled by timed automata or hybrid automata, contain data variables that range over real numbers.

Several symbolic techniques have been proposed to deal with PBESs over infinite data domains [8, 9, 10], but their application is unfortunately limited to specific subclasses of PBESs. Typically, these fragments exclude PBESs in which both logical quantifiers occur; *i.e.* PBESs may only contain universal quantification or only existential quantification. Such constraints effectively limit the class of properties that can be encoded, excluding, *e.g.* most behavioural equivalence decision problems, but also many CTL* properties.

*Email addresses:* `T.S.Neele@tue.nl` (Thomas Neele), `T.A.C.Willemse@tue.nl` (Tim A.C. Willemse), `J.F.Groote@tue.nl` (Jan Friso Groote)

*Contributions.* The current paper extends our earlier work [11], where we presented an approach, called *PBES quotienting*, that is more general than the symbolic techniques discussed above: PBES quotienting is applicable to the *full* class of PBESs. Our approach is based on *minimal model generation* (MMG) [12], a similar procedure that operates on behavioural models, such as extended finite state machines or timed automata. PBES quotienting relies on a normal form for PBESs called *clustered recursive form* (CRF). This normal form facilitates reasoning about the dependencies between predicate variables in a PBES and enables capturing these in a *dependency graph.* A procedure based on quotienting can be used to compute a minimal reduced dependency graph from a symbolic representation of the dependency graph. PBES quotienting can be further improved by extracting finite partial solutions from PBESs that have an infinite minimal reduced dependency graph.

To validate the above, we performed a number of experiments with an implementation of our procedures and compared these to the solver of [9]. The results of this evaluation show that our technique is indeed capable of solving decision problems that existing approaches fail to solve so far. In particular, the experiments show that our technique is a promising generic approach for model checking of (timed) modal mu-calculus properties [13] on systems with infinite data domains and also equivalence checking of systems with infinite data domains. The current paper extends [11] as follows:

- We discuss the basic ideas of minimal model generation: quotienting in the setting of behavioural models. Using Lamport's bakery protocol as an example, we show the limitations of this technique.
- We propose a second optimisation for PBES quotienting that employs knowledge of an intermediate solution to identify when this intermediate result correctly represents the dependency graph. In this way, more nodes in the dependency graph are considered equivalent and the procedure can possibly terminate earlier.
- We provide full proofs for the correctness of our basic PBES quotienting procedure (Theorem 8) and each of the optimisations we propose (Theorems 10 and 11).
- We extend the experimental evaluation by including an implementation of our new optimisation. Furthermore, we experimentally compare the applicability and performance of MMG and our PBES quotienting approach.

The rest of the paper is structured as follows: Section 2 introduces the basic theoretical concepts and the minimal model generation procedure. Section 3 contains an example that shows how minimal model generation can be applied and what its shortcomings are. Then, Section 4 introduces PBESs and dependency graphs and Sections 5 and 6 show how the minimal model generation procedure can be adapted to the setting of PBESs. Two improvements to the procedure are presented in Sections 7 and 8 respectively. In Section 9, we perform experiments to compare minimal model generation with the new PBES procedure, and its optimisations. Finally, Section 10 gives an overview of related work and Section 11 presents a conclusion and suggestions for future work.

## 2. Preliminaries

In this paper, we work with abstract data types and denote their non-empty data sorts with the letters $D, E, \dots$ and their corresponding semantic domains by $\mathbb{D}, \mathbb{E}, \dots$ In addition, we use $B$ to denote the Booleans and $N$ to denote the natural numbers $\{0, 1, 2, \dots\}$, which have the semantic counterparts $\mathbb{B}$ and $\mathbb{N}$ respectively. We also have a singleton sort $D_\star = \{\star\}$ on which no operations are defined. Furthermore, we have a set of data variables $\mathcal{V}$. Syntactic variables are typically denoted with the letters $d$ and $e$, while semantic values are denoted with $v$ and $w$. To indicate that the type of variable $d$ is $D$, we write $d{:}D$. For expressions that contain variables, we have a data environment $\delta$ that maps each variable in $\mathcal{V}$ to an element of the corresponding sort. The semantics of an expression $f$ in the context of a data environment $\delta$ is denoted $[\![f]\!]\delta$. The set of all data environments is $\Delta$. Updates to an environment $\delta$ are denoted by $\delta[v/d]$, which is defined as $\delta[v/d](d) = v$ and $\delta[v/d](d') = \delta(d')$ for all variables $d, d'$ satisfying $d' \neq d$.

## *2.1. Processes and Transition Systems*

In the following definition, we assume the existence of a fixed set of actions $Act$ and a data domain $D_{par}$ (with semantic domain $\mathbb{D}_{par}$) from which action parameters are taken. The basic representation of behaviour we consider is a labelled transition system.

**Definition 1.** A *labelled transition system* (LTS) is a three-tuple $\mathcal{L} = (S, \rightarrow, \hat{s})$, where

- $S$ is a set of states, which we refer to as the *state space*;
- $\rightarrow \subseteq S \times (Act \times \mathbb{D}_{par}) \times S$ is the transition relation; and
- $\hat{s} \in S$ is the initial state.

Below, we will use the terms LTS and transition system interchangeably. We write $s \xrightarrow{a(v)} s'$ whenever $(s, (a, v), s') \in \rightarrow$. An LTS can be represented compactly with a *linear process.* Without loss of generality, in this paper, we only consider processes with a single parameter $d$ of type $D$. In our examples, we use (multi-parameter) processes ranging over the Booleans ($B$), the natural numbers ($N$), etc.

**Definition 2.** A *linear process specification* (LPS) is a tuple $L = (P, \hat{d})$, where $\hat{d}{:}D$ is the initial state, given by a closed expression of sort $D$, and $P$ is a recursive process of the shape

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

where $I$ is a finite (possibly empty) index set, $e_i$ is a summation variable ranging over the non-empty domain $E_i$, $c_i$ is a boolean condition, $a_i \in Act$ is an action that has the parameter $f_i(d, e_i){:}D_{par}$ and $g_i$ is an update expression.

We adopt the following conventions for the examples: we unfold the first sum operator by using the *choice operator* (notation $+$) and we omit the second sum operator when $e_i$ is not used. Intuitively, in every state represented by variable $d$, a linear process offers a (non-deterministic) choice to perform an action $a_i(f_i(d, e_i))$ that is enabled, *i.e.*, for which $c_i(d, e_i)$ is *true* for some $e_i$. After performing this action, the state is updated to $g_i(d, e_i)$. The semantics of an LPS is defined in terms of an LTS.

**Definition 3.** Let $L = (P, \hat{d})$ be an LPS and $\delta$ some data environment. Then the associated LTS is defined as $\mathcal{L}_L = (\mathbb{D}, \rightarrow, \hat{v})$, where $\hat{v}$ is the semantic value corresponding to $\hat{d}$ and $\rightarrow$ is the set satisfying for all $v \in \mathbb{D}$, $i \in I$, $v_i \in \mathbb{E}_i$: $(v, (a_i, [\![f_i(d, e_i)]\!]\delta[v/d, v_i/e_i]), [\![g_i(d, e_i)]\!]\delta[v/d, v_i/e_i]) \in \rightarrow$ if and only if $[\![c_i(d, e_i)]\!]\delta[v/d, v_i/e_i]$ holds.

The most common and straightforward way of analysing the behaviour specified by an LPS is to construct the corresponding transition system by means of *state space exploration.* Starting from the initial state, the exploration procedure computes outgoing transitions and stores new states it encounters. However, this technique is not complete: for processes with an infinite reachable state space, the procedure does not terminate. This is demonstrated in the following example.

**Example 1.** We consider a model of a primitive coffee machine that accepts coins of 5 cents and 10 cents and never gives change. After ordering a coffee and inputting at least 15 cents, the machine can give coffee. A possible representation of this machine is the LPS $(\mathit{Machine}, (\mathit{false}, 0))$, where $\mathit{Machine}$ is the process defined as follows.

$$\begin{aligned} &\mathit{Machine}(\mathit{idle}{:}B, \mathit{balance}{:}N) = \\ &\qquad \mathit{idle} \rightarrow \mathit{order} \cdot \mathit{Machine}(\mathit{false}, \mathit{balance}) \\ &\qquad + \sum_{c:\mathit{Coin}} \neg \mathit{idle} \rightarrow \mathit{insert}(c) \cdot \mathit{Machine}(\mathit{false}, \mathit{balance} + \mathit{value}(c)) \\ &\qquad + (\neg \mathit{idle} \wedge \mathit{balance} \geq 15) \rightarrow \mathit{coffee} \cdot \mathit{Machine}(\mathit{true}, 0) \end{aligned}$$

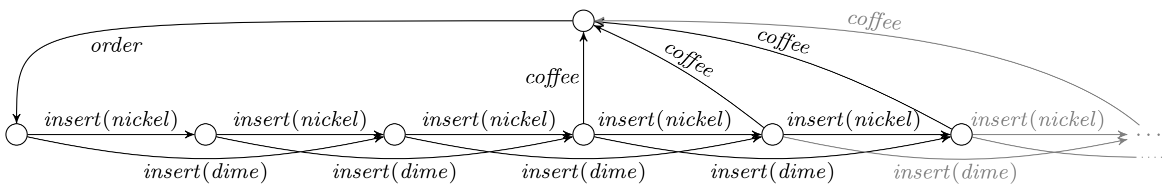

Figure 1: Infinite LTS of the coffee machine of Example 1.

Here, *idle* expresses whether the machine is idle or a transaction is in process and *balance* stores the amount of money inserted during the current transaction. Furthermore, *Coin* is a data sort containing the values *nickel* and *dime* and *value*: $Coin \to N$ computes the corresponding value of a coin. Part of the reachable state space of the associated LTS is depicted in Figure 1 Since there is no upper bound on the amount of money that can be inserted, the state space is infinite. □

To deal with instances like the coffee machine above, many symbolic approaches have been developed, one of which is *minimal model generation* (MMG) [12, 14], which we discuss below. Before we can introduce the MMG approach, we first introduce two more concepts that make up the underlying theory. First, we introduce the concept of a reduced LTS, which allows us to reason about certain infinite LTSs using a finite representation. Reduced LTSs rely on the notion of a *partition*: a set $A \subseteq 2^B$ is a partition of a set $B$ if and only if $\bigcup A = B$ and for distinct $a, a' \in A$, it holds that $a \cap a' = \emptyset$.

**Definition 4.** Let $\mathcal{L} = (S, \rightarrow, \hat{s})$ be an LTS. Then, $\mathcal{L}_r = (S_r, \rightarrow_r, b)$ is a *reduced LTS* iff:

- $S_r \subseteq 2^S$ is a partition of $S$;
- $\rightarrow_r = \{(b, (a, v), b') \mid \exists s \in b, t \in b'. \, s \xrightarrow{a(v)} t\}$.

We say $\mathcal{L}$ is the *base LTS* of $\mathcal{L}_r$.

We commonly refer to an element $b \in S_r$ as a *block*. Not all reduced LTSs are a meaningful representation of their base LTS. After all, any LTS can be represented by a reduced LTS with only one block that contains all states. To preserve interesting properties, every block should only contain states that are related by a certain equivalence relation. Here, we use *bisimulation* [15], which preserves most logic properties, such as those formulated in the *modal mu-calculus* [13].

**Definition 5.** Given an LTS $\mathcal{L} = (S, \rightarrow, \hat{s})$, a relation $\mathcal{R} \subseteq S \times S$ on states is a *bisimulation relation* iff for all $s$, $t$ such that $s\mathcal{R}t$, it holds that:

- For all transitions $s \xrightarrow{a(v)} s'$, there is a transition $t \xrightarrow{a(v)} t'$ such that $s'\mathcal{R}t'$.
- For all transitions $t \xrightarrow{a(v)} t'$, there is a transition $s \xrightarrow{a(v)} s'$ such that $s'\mathcal{R}t'$.

We say two states $s$ and $t$ are *bisimilar*, notation $s \leftrightarrows t$, iff they are related by some bisimulation relation. Two LTSs are bisimilar iff their initial states are bisimilar.

We call the reduced LTS that is minimal under bisimulation the *bisimulation quotient*, which we denote with $\mathcal{L}/\leftrightarrows$. The state space of $\mathcal{L}/\leftrightarrows$, denoted $S/\leftrightarrows$, consists of the equivalence classes induced by bisimulation, *i.e.*, states $s$ and $t$ are in the same block if and only if they are bisimilar. Observe that the bisimulation quotient is well-defined, since bisimilarity is an equivalence relation.

## *2.2. Partition refinement*

To compute the bisimulation quotient, we rely on *partition refinement*. In this procedure, a partition of the state space is iteratively refined until it becomes *stable* (a formal definition follows). We say a partition $\pi$ is *finer* than a partition $\pi'$ iff all blocks of $\pi$ are contained in some block of $\pi'$. The coarsest stable partition coincides with the equivalence classes under bisimulation.

**Procedure 1:** Minimal model generation for LPSs

```
Input: LPS $L = (P, \hat{d})$, initial partition $\pi_0$
1 $i := 0$;
2 while $\pi_i$ is not stable do
3     $i := i + 1$;
4     $\pi_i := (\pi_{i-1} \setminus \{b\}) \cup \{split(b, b', a(v)), co\text{-}split(b, b', a(v))\}$ for some $b, b' \in \pi_{i-1}$, $a \in Act$, $v \in \mathbb{D}_{par}$ such that
          $split(b, b', a(v))$ and $co\text{-}split(b, b', a(v))$ are non-empty;
5     $\rightarrow_i := \{(b, (a, v), b') \mid \exists s \in b, t \in b'.\, s \xrightarrow{a(v)} t\}$;
6     $\pi_i := \{b \mid \hat{b} \rightarrow_i^* b\}$ where $[\![\hat{d}]\!] \in \hat{b}$;
7 return $(\pi_i, \rightarrow_i, \hat{b})$ where $[\![\hat{d}]\!] \in \hat{b}$;
```

Procedure 1 shows how to perform partition refinement on an LTS $\mathcal{L} = (\mathbb{D}, \rightarrow, \hat{v})$ that underlies the LPS $L$. The initial partition has one block containing all states, *i.e.*, $\pi_0 = \{\mathbb{D}\}$. In every iteration, we find two blocks $b, b' \in \pi_i$ and a combination of action and parameter $a(v)$ and split $b$ with respect to $b'$ in the following way:

$$split(b, b', a(v)) = \{s \in b \mid \exists t \in b'.\, s \xrightarrow{a(v)} t\}$$
$$co\text{-}split(b, b', a(v)) = b \setminus split(b, b', a(v))$$

Then we update the partition and transition relation to reflect this split (lines 4 and 5). Next, we use a simple forward exploration on blocks to compute which blocks are reachable from the initial block, and discard blocks that are not reachable (line 6). Here, $\rightarrow_i^*$ is the reflexive transitive closure of the transition relation. Strictly speaking, after discarding one or more blocks, $\pi_i$ is no longer a partition of the complete state space, but only of some over-approximation of the reachable state space. Note that each partition $\pi_{i+1}$ is finer than partition $\pi_i$.

If a block $b$ cannot be split with respect to a block $b'$ for all actions $a(v)$, we say $b$ is *stable* (under bisimulation) with respect to $b'$. Block $b$ is stable with respect to a set of blocks $K$ iff it is stable with respect to all the blocks in $K$. A partition $\pi$ is stable (with respect to itself) iff all of the blocks in $\pi$ are stable with respect to $\pi$. The partition refinement procedure terminates when $\pi$ is stable (line 2). The reachable part of the bisimulation quotient can be constructed from the stable partition using Definition 4. Remark that termination is not guaranteed as not every infinite LTS has a finite bisimulation quotient. Consequently, Procedure 1, and also the other procedures we present below, is a semi-decision procedure.

Since our goal is to enable reasoning about LTSs with an infinite state space, we cannot store blocks by storing each of their constituent states explicitly. Instead, we represent each block with a *characteristic function.*

**Definition 6.** Let $L$ be an LPS and $b$ be a set of states in the associated LTS. The corresponding *characteristic function* $\mathbb{K}_b : \mathbb{D} \rightarrow \mathbb{B}$ is defined as:

$$\mathbb{K}_b(v) = \begin{cases} true & \text{if } v \in b \\ false & \text{otherwise} \end{cases}$$

Henceforth, we represent the semantic function $\mathbb{K}_b$ for block $b$ with a syntactic Boolean expression $k_b$. With this representation, we can implement Procedure 1 symbolically. Firstly, the initial partition $\pi_0$ is represented by $\{\lambda w \in \mathbb{D}.\, true\}$. Secondly, $split(k_b, k_{b'}, a(v))$ and $co\text{-}split(k_b, k_{b'}, a(v))$ are the respective symbolic implementations of $split(b, b', a(v))$ and $co\text{-}split(b, b', a(v))$. In the following definitions, $\delta$ is an arbitrary data environment and $k_b$ and $k_{b'}$ are the characteristic functions corresponding to blocks $b$ and $b'$,

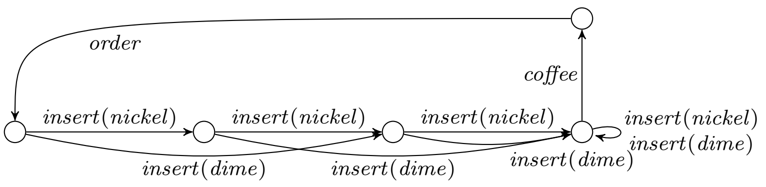

Figure 2: The bisimulation quotient of the coffee machine with an infinite state space (Example 1) as computed by Procedure 1.

respectively.

$$\begin{aligned}
&\mathit{split}(k_b,k_{b'},a(v)) = \\
&\qquad \lambda w \in \mathbb{D}.\, [\![k_b(d) \wedge \bigvee_{i\in I} a = a_i \wedge \big(\exists e_i{:}E_i.\, c_i(d,e_i) \wedge e = f_i(d,e_i) \wedge k_{b'}(g_i(d,e_i))\big)]\!]\delta[w/d, v/e] \\
&\mathit{co\text{-}split}(k_b,k_{b'},a(v)) = \\
&\qquad \lambda w \in \mathbb{D}.\, [\![k_b(d) \wedge \neg \bigvee_{i\in I} a = a_i \wedge \big(\exists e_i{:}E_i.\, c_i(d,e_i) \wedge e = f_i(d,e_i) \wedge k_{b'}(g_i(d,e_i))\big)]\!]\delta[w/d, v/e]
\end{aligned}$$

A similar symbolic implementation is necessary to compute the transition relation on line 5 of Procedure 1, where we replace $\exists s \in b, t \in b'.\, s \xrightarrow{a(v)} t$ by the following:

$$[\![\exists d{:}D.\, k_b(d) \wedge \bigvee_{i\in I} a = a_i \wedge \big(\exists e_i{:}E_i.\, c_i(d,e_i) \wedge e = f_i(d,e_i) \wedge k_{b'}(g_i(d,e_i))\big)]\!]\delta[v/e]$$

**Example 2.** We revisit the coffee machine with an infinite state space from Example 1. The bisimulation quotient generated by Procedure 1 is depicted in Figure 2. All states reached by inserting more than 15 cents are collapsed into one (the state with the self loop), since they are all bisimilar. A characteristic function for this block can be $\lambda \mathit{idle} \in \mathbb{B}, \mathit{balance} \in \mathbb{N}.\, \neg\mathit{idle} \wedge \mathit{balance} \geq 15$. □

Although characteristic functions help to deal with an infinite state space, MMG still does not deal well with infinite action sets or an infinite data domain for action parameters. The next section presents an example that shows how these limitations impact the ability to perform model checking using MMG.

## 3. Motivating Example

To show how MMG can be used for model checking and illustrate its limitations, we introduce a slightly larger example in this section. This example will also serve as a running example throughout the rest of the paper. The model we consider is a simplified version of Lamport's bakery protocol [16]. In our setting, there are only two processes (customer 0 and customer 1) and all writes and reads are atomic. When customer $i$ enters the bakery, he/she does not have a number ($n = 0$). At any point, the customer can pick a number, which is one larger than the number of the other customer. If both customers are waiting, the customer with the smallest number can enter the critical section. When leaving the critical section, the number is discarded ($n$ is reset to 0). See Figure 3.

The LPS $L = (\mathit{Bakery}, (\mathit{idle}, 0, \mathit{idle}, 0))$ represents the behaviour of the two customers, where $\mathit{Bakery}$ is the following linear process.

$$\begin{aligned}
&\mathit{Bakery}(s_0{:}\mathit{State}, n_0{:}N, s_1{:}S, n_1{:}N) = \\
&\qquad\quad (s_0 = \mathit{idle}) \to \mathit{pick}_0(n_1+1) \cdot \mathit{Bakery}(\mathit{waiting}, n_1+1, s_1, n_1) \\
&\qquad + (s_0 = \mathit{waiting} \wedge (n_1 = 0 \vee n_0 < n_1)) \to \mathit{enter}_0 \cdot \mathit{Bakery}(\mathit{cs}, n_0, s_1, n_1) \\
&\qquad + (s_0 = \mathit{cs}) \to \mathit{leave}_0 \cdot \mathit{Bakery}(\mathit{idle}, 0, s_1, n_1) \\
&\qquad + (s_1 = \mathit{idle}) \to \mathit{pick}_1(n_0+1) \cdot \mathit{Bakery}(s_0, n_0, \mathit{waiting}, n_0+1) \\
&\qquad + (s_1 = \mathit{waiting} \wedge (n_0 = 0 \vee n_1 < n_0)) \to \mathit{enter}_1 \cdot \mathit{Bakery}(s_0, n_0, \mathit{cs}, n_1) \\
&\qquad + (s_1 = \mathit{cs}) \to \mathit{leave}_1 \cdot \mathit{Bakery}(s_0, n_0, \mathit{idle}, 0)
\end{aligned}$$

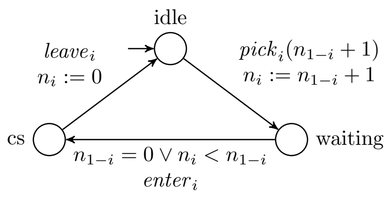

Figure 3: Process $i$ from the simplified bakery protocol.

In this encoding, $s_i$ and $n_i$ represent the state and number of customer $i$, respectively. Furthermore, the states of a single process are encoded in the sort *State*.

On this model, we would like to check the property "regardless of which number customer 0 picks, it can always enter the critical section in a finite time". This can be formalised with the modal mu-calculus formula $\nu X.([-]X \wedge \forall m{:}N.[pick_0(m)]\mu Y.([\overline{enter_0}]Y \wedge \langle - \rangle true))$. Here, the fixpoint variable $X$ ranges over the whole state space and $Y$ holds if and only if for all paths, $enter_0$ occurs within a finite number of steps. However, the state space of the LPS is infinite and also the set of actions is infinite, due to the parameter of the $pick_i$ actions, which is a natural number. Therefore, neither classical state space exploration nor minimal model generation can compute an LTS on which the formula can be evaluated. An alternative approach is to encode this model checking question in a *parameterised Boolean equation system.*

## 4. Parameterised Boolean Equation Systems

A parameterised Boolean equation system is a sequence of fixpoint equations over predicate formulae. We confine ourselves to giving a cursory overview of the syntax and semantics of the relevant theory and refer the interested reader to [1] for a more in-depth treatment and additional examples.

**Definition 7.** A *predicate formula* is defined by the following grammar:

$$\phi ::= b \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \exists e{:}E.\, \phi \mid \forall e{:}E.\, \phi \mid X(f)$$

where $b$ is an expression of sort $B$, $e$ is a variable of sort $E$, $X$ is a predicate variable of sort $D \to B$, which is taken from some set $\mathcal{X}$ of sorted predicate variables and argument $f$ is an expression of sort $D$. The interpretation of a predicate formula $\phi$ in the context of a *predicate environment* $\eta : \mathcal{X} \to 2^{\mathbb{D}}$, providing an interpretation for predicate variables from $\mathcal{X}$, and a data environment $\delta$ is denoted by $[\![\phi]\!]\eta\delta$ and inductively defined as follows:

$$\begin{array}{rcl@{\qquad}rcl}
[\![b]\!]\eta\delta & \Leftrightarrow & [\![b]\!]\delta & [\![X(f)]\!]\eta\delta & \Leftrightarrow & [\![f]\!]\delta \in \eta(X) \\
[\![\varphi \wedge \psi]\!]\eta\delta & \Leftrightarrow & [\![\varphi]\!]\eta\delta \text{ and } [\![\psi]\!]\eta\delta \text{ hold} & [\![\varphi \vee \psi]\!]\eta\delta & \Leftrightarrow & [\![\varphi]\!]\eta\delta \text{ or } [\![\psi]\!]\eta\delta \text{ hold} \\
[\![\varphi \Rightarrow \psi]\!]\eta\delta & \Leftrightarrow & [\![\varphi]\!]\eta\delta \text{ holds implies that } [\![\psi]\!]\eta\delta \text{ holds} \\
[\![\forall e{:}E.\, \varphi]\!]\eta\delta & \Leftrightarrow & \text{for all } v \in \mathbb{E},\ [\![\varphi]\!]\eta\delta[v/e] \text{ holds} \\
[\![\exists e{:}E.\, \varphi]\!]\eta\delta & \Leftrightarrow & \text{for some } v \in \mathbb{E},\ [\![\varphi]\!]\eta\delta[v/e] \text{ holds}
\end{array}$$

A predicate formula is *syntactically monotone* iff all its subformulae of the form $\varphi \Rightarrow \psi$ are such that $\varphi$ contains no predicate variables. As with LPSs, in this paper we only consider parameterised Boolean equation systems where each equation carries the same single parameter of a given data sort $D$. This does not affect the generality of the theory we develop. The examples may contain equations with multiple parameters.

**Definition 8.** A *parameterised Boolean equation system* (PBES) is a sequence of equations as defined by the following grammar:

$$\mathcal{E} ::= \emptyset \mid (\nu X(d{:}D) = \varphi)\mathcal{E} \mid (\mu X(d{:}D) = \varphi)\mathcal{E}$$

where $\emptyset$ is the empty PBES, $\mu$ and $\nu$ denote the least and greatest fixpoint operator, respectively, and $X \in \mathcal{X}$ is a predicate variable of sort $D \to B$. The right-hand side $\varphi$ is a syntactically monotone predicate formula. Lastly, $d \in \mathcal{V}$ is a parameter of sort $D$.

We use $\mathsf{bnd}(\mathcal{E})$ to denote the predicate variables bound in $\mathcal{E}$, *i.e.*, those variables occurring at the left-hand side of an equation. For an equation for $X$, $d_X$ denotes its parameter and $\varphi_X$ denotes its right-hand side predicate formula. We omit the trailing $\emptyset$. We say a PBES is *closed* when it does not contain free variables, *i.e.*, all data variables that occur in a right-hand side $\varphi_X$ are either bound by a quantifier or as a data parameter of $X$, whereas all predicate variables belong to $\mathsf{bnd}(\mathcal{E})$. A PBES $\mathcal{E}$ is called a *Boolean equation system* (BES) iff all predicate variables bound by $\mathcal{E}$ have type $D_\star \to B$ and every right-hand side only contains the operators $\wedge$ and $\vee$, constants *true* and *false* and $X(\star)$. We say that a PBES $\mathcal{E}$ is *well-formed* iff for every $X \in \mathsf{bnd}(\mathcal{E})$ there is exactly one equation in $\mathcal{E}$. In the remainder of the paper we only reason about well-formed, closed PBESs.

**Definition 9.** The *solution* $[\![\mathcal{E}]\!]\eta\delta$ of a PBES $\mathcal{E}$ in the context of a predicate environment $\eta$ and a data environment $\delta$, is a predicate environment that is defined inductively:

$$\begin{aligned}
[\![\,\emptyset\,]\!]\eta\delta &= \eta \\
[\![(\mu X(d{:}D) = \varphi_X)\mathcal{E}]\!]\eta\delta &= [\![\mathcal{E}]\!]\eta[\mu T_X/X]\delta \\
[\![(\nu X(d{:}D) = \varphi_X)\mathcal{E}]\!]\eta\delta &= [\![\mathcal{E}]\!]\eta[\nu T_X/X]\delta
\end{aligned}$$

with $T_X(R) = \{v \in \mathbb{D} \mid [\![\varphi_X]\!]([\![\mathcal{E}]\!]\eta[R/X]\delta)\delta[v/d]\}$.

Intuitively, the solution of a PBES gives priority to fixpoints that occur early in the PBES, while satisfying the equalities that are specified by each equation. The monotonicity of the transformer $T_X : 2^{\mathbb{D}} \to 2^{\mathbb{D}}$, which follows from syntactic monotonicity of $\varphi_X$, guarantees the existence of the least fixpoint $\mu T_X$ and greatest fixpoint $\nu T_X$ in the complete lattice $(2^{\mathbb{D}}, \subseteq)$. Also, note that the solution of a bound variable in a closed PBES does not depend on the environments $\eta$ and $\delta$. For this reason, we often omit $\eta$ and $\delta$ and simply write $[\![\mathcal{E}]\!]$ instead of $[\![\mathcal{E}]\!]\eta\delta$. Finally, for a PBES $\mathcal{E}$ and some $X \in \mathsf{bnd}(\mathcal{E})$ we sometimes say that (the solution to) $X(v)$ is *true* iff $v \in [\![\mathcal{E}]\!](X)$.

**Example 3.** Consider the following PBES $\mathcal{E}$ consisting of an equation for $X$ and an equation for $Y$, both carrying a single parameter. Furthermore, the equation for $X$ has a least fixpoint, and the equation for $Y$ has a greatest fixpoint.

$$\begin{aligned}
\mu X(n{:}N) &= (\exists m{:}N.\, m \geq n \wedge X(m)) \wedge Y(\mathit{false}) \\
\nu Y(b{:}B) &= Y(\neg b)
\end{aligned}$$

By applying the semantics of predicate formulae, we can derive the predicate transformer for $Y$ as follows:

$$\begin{aligned}
T_Y(R) &= \{v \in \mathbb{B} \mid [\![Y(\neg b)]\!]([\![\emptyset]\!]\eta[R/Y]\delta)\delta[v/b]\} \\
&= \{v \in \mathbb{B} \mid [\![Y(\neg b)]\!]\eta[R/Y]\delta[v/b]\} \\
&= \{v \in \mathbb{B} \mid \neg v \in R\}
\end{aligned}$$

The largest set that satisfies $T_Y(R) = R$ is $\mathbb{B}$, hence $\nu T_Y = \mathbb{B}$. We can apply a similar reasoning to $X$ to obtain its predicate transformer.

$$\begin{aligned}
T_X(R) &= \{v \in \mathbb{N} \mid [\![(\exists m{:}N.\, m \geq n \wedge X(m)) \wedge Y(\mathit{false})]\!]([\![\nu Y(b{:}B) = Y(\neg b)]\!]\eta[R/X]\delta)\delta[v/n]\} \\
&= \{v \in \mathbb{N} \mid \exists v' \in \mathbb{N}.\, v' \geq v \wedge v' \in R\}
\end{aligned}$$

We derive that $\mu T_X = \emptyset$. The application of Definition 9 yields $[\![\mathcal{E}]\!]\eta\delta = \eta[\mu T_X/X][\nu T_Y/Y]$. The solution of $\mathcal{E}$ thus satisfies $[\![\mathcal{E}]\!](X) = \emptyset$ and $[\![\mathcal{E}]\!](Y) = \mathbb{B}$. Note that since this particular example is not mutually recursive, the order of the equations does not influence the solution. □

$$\begin{tikzcd}
(X,0) \arrow[r] \arrow[d] & (X,5) \arrow[loop right] \arrow[dl] \\
(Y,\mathit{false}) \arrow[r, bend left] & (Y,\mathit{true}) \arrow[l, bend left]
\end{tikzcd}$$

(a) A positive dependency graph

$$\begin{tikzcd}
(X,0) \arrow[loop below] \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left] & (X,1) \arrow[r] \arrow[d] & (X,2) \arrow[r] \arrow[dl] & \cdots \arrow[dll] \\
 & (Y,\mathit{false}) \arrow[r, bend left] & (Y,\mathit{true}) \arrow[l, bend left] &
\end{tikzcd}$$

(b) A negative dependency graph

Figure 4: Dependency graphs for the PBES from Example 3.

## *4.1. Dependency graphs and proof graphs*

The theory of this paper is built on the notion of *dependency graphs* and *proof graphs* explored in [17]. Intuitively, a proof graph is a witness providing an operational explanation for a (partial) solution of a PBES. Before we introduce these graphs formally, we need some additional concepts.

First, $\mathsf{sig}(\mathcal{E})$ is the signature of $\mathcal{E}$, defined as $\mathsf{sig}(\mathcal{E}) = \{(X, v) \mid X \in \mathsf{bnd}(\mathcal{E}), v \in \mathbb{D}\}$. For a given set $S \subseteq \mathsf{sig}(\mathcal{E})$, the predicate environment $\mathsf{env}(S, \mathit{true})$ that follows from it is defined as $\mathsf{env}(S, \mathit{true})(X) = \{v \in \mathbb{D} \mid (X, v) \in S\}$. Dually, we define $\mathsf{env}(S, \mathit{false})(X) = \mathbb{D} \setminus \mathsf{env}(S, \mathit{true})(X)$. Furthermore, every predicate variable bound in $\mathcal{E}$ is assigned a *rank*, where $\mathsf{rank}_{\mathcal{E}}(X) \leq \mathsf{rank}_{\mathcal{E}}(Y)$ if $X$ occurs before $Y$ in $\mathcal{E}$, and $\mathsf{rank}_{\mathcal{E}}(X)$ is even if and only if $X$ is labelled with a greatest fixed point. We assume every PBES has a fixed $\mathsf{rank}$ function.

**Definition 10.** Let $\mathcal{E}$ be a PBES and $G = (V, E)$ be a directed graph, where $V \subseteq \mathsf{sig}(\mathcal{E})$. We say $G$ is a *dependency graph* for $r \in \mathbb{B}$ iff for every $(X, v) \in V$ and for all $\delta$, $[\![\varphi_X]\!]\eta(\delta[v/d_X]) = r$ with $\eta = \mathsf{env}((X, v)E, r)$, where $sE$ denotes the successor set of a node, defined as $sE = \{t \mid s\,E\,t\}$.

We distinguish *positive dependency graphs*, where $r$ is *true*, from *negative dependency graphs*, where $r$ is *false*. Intuitively, in a positive dependency graph, $\eta = \mathsf{env}((X, v)E, \mathit{true})$ is a predicate environment that maps all successors of $(X, v)$ to *true* and all other nodes to *false*. Then, the requirement is that $\varphi_X$ (and thus $X(v)$) is *true* under $\eta$ and a data environment that maps $d_X$ to $v$. In other words, the successors of a node $(X, v)$ being *true* must imply that $(X, v)$ is *true* as well. Dually, a negative dependency graph indicates a node $(X, v)$ is *false*, when its successors are all *false*.

**Example 4.** Recall the PBES from Example 3. Figure 4 depicts a positive and a negative dependency graph for this PBES. We focus on node $(X, 0)$ in the positive dependency graph of Figure 4(a). Its successors are $(X, 5)$ and $(Y, \mathit{false})$. The environment $\eta$ induced by these successors is given by $\mathsf{env}((X, 0)E, \mathit{true})$, which sets these successors to *true*; *i.e.*, $\eta$ is such that $\eta(X) = \{5\}$ and $\eta(Y) = \{\mathit{false}\}$. When we evaluate the right-hand side of the equation for $X$ in the context of $\eta$ and parameter $n$ set to 0, we obtain $[\![(\exists m{:}N.\, m \geq n \wedge X(m)) \wedge Y(\mathit{false})]\!]\eta(\delta[0/n]) = \mathit{true}$. Therefore, the positive dependency graph condition is satisfied for node $(X, 0)$. A similar reasoning applies to the other nodes, showing that the dependency graph condition is satisfied by each of them.

Note that nodes $(Y, \mathit{false})$ and $(Y, \mathit{true})$ are dependent on each other in both dependency graphs. Furthermore, in the negative case, $(X, 0)$ needs no dependency on $(Y, \mathit{false})$ as long as it depends on all $(X, i)$ with $i \in \mathbb{N}$, which is the case in the negative dependency graph of Figure 4(b). Hence, this particular dependency graph is infinite. An alternative, finite negative dependency graph for $(X, 0)$ is $(X, 0) \longrightarrow (Y, \mathit{false}) \rightleftarrows (Y, \mathit{true})$ □

A dependency graph captures the logical structure of a PBES; it does not include the fixpoint semantics. If we want to reason about the actual solution of a PBES, we need an additional restriction on the infinite paths in a dependency graph. Dependency graphs that meet these restrictions are called *proof graphs*.

**Definition 11.** Let $G = (V, E)$ be a positive (respectively negative) dependency graph for a PBES $\mathcal{E}$. Then $G$ is a *positive proof graph* (respectively *negative proof graph*) iff for all infinite paths $\pi$ in $G$, the number $\min\{\mathsf{rank}_{\mathcal{E}}(X) \mid X \in V^{\infty}(\pi)\}$ is even (respectively odd), where $V^{\infty}(\pi)$ is the set of predicate variables that occur infinitely often along $\pi$.

$$\begin{tikzcd}
(Y, \mathit{false}) \arrow[r, bend left] & (Y, \mathit{true}) \arrow[l, bend left]
\end{tikzcd}$$

(a) A positive proof graph

$$\begin{tikzcd}
(X,0) \arrow[r] \arrow[loop below] \arrow[rr, bend left] \arrow[rrr, bend left] & (X,1) \arrow[r] \arrow[loop below] \arrow[rr, bend left] & (X,2) \arrow[r] \arrow[loop below] & \cdots
\end{tikzcd}$$

(b) A negative proof graph

Figure 5: Proof graphs for the PBES from Example 3.

Observe that predicate variables with a lower rank dominate those with a higher rank. This reflects the fact that fixpoint symbols that occur early in an equation system take priority over later ones (cf. Definition 9).

**Example 5.** Recall again the PBES from Example 3. In this PBES, the rank of $X$ is 1, and the rank of $Y$ is 2. Figure 5 depicts a positive and a negative proof graph for this PBES. Note that Figure 5(a) depicts the smallest positive proof graph proving that $Y(\mathit{false})$ is *true*. Larger proof graphs can be obtained by adding a self loop to $(Y, \mathit{false})$ or $(Y, \mathit{true})$. Similarly, the proof graph in Figure 5(b) is the smallest negative proof graph explaining that $X(0)$ is *false*. However, there is a smaller negative proof graph showing that $X(1) = \mathit{false}$, *viz.* the graph that does not include $(X, 0)$. Note that for every $i \in \mathbb{N}$, the proof graph for $(X, i)$ is infinite, since $(X, i)$ depends on all $(X, j)$ with $j \geq i$. □

The next theorem formally states the relationship between proof graphs and the solution of a PBES.

**Theorem 1 ([17]).** *Let $\mathcal{E}$ be a PBES with $X \in \mathsf{bnd}(\mathcal{E})$. Then $v \in [\![\mathcal{E}]\!](X)$ iff there is a positive proof graph $(V, E)$ such that $(X, v) \in V$. Dually, $v \notin [\![\mathcal{E}]\!](X)$ iff there is a negative proof graph containing $(X, v)$.*

In [17], proof graphs were introduced mainly to formalise the concept of witnesses and counterexamples, as implemented in [18]. Instead, we rely on the above theorem to (partially) *solve* PBESs by searching for concise representations of proof graphs. Before we explain this idea in detail, we revisit the bakery example to illustrate how to apply PBESs in model checking.

## *4.2. Bakery example*

Recall from Section 3 that we want to check the formula $\nu X.([-]X \wedge \forall m{:}N.[\mathit{pick}_0(m)]\mu Y.([\overline{\mathit{enter}_0}]Y \wedge \langle - \rangle \mathit{true}))$ on the LPS that represents two customers in a bakery. From the LPS and the formula, the following PBES can be constructed automatically [19]:

$$
\begin{aligned}
&\nu X(s_0{:}S, n_0{:}N, s_1{:}S, n_1{:}N) = && (1)\\
&\quad s_0 = \mathit{idle} \Rightarrow Y(n_1 + 1, s_1, n_1) \wedge && (2)\\
&\quad s_0 = \mathit{idle} \Rightarrow X(\mathit{waiting}, n_1 + 1, s_1, n_1) \wedge && (3)\\
&\quad s_0 = \mathit{waiting} \wedge (n_1 = 0 \vee n_0 < n_1) \Rightarrow X(\mathit{cs}, n_0, s_1, n_1) \wedge && (4)\\
&\quad s_0 = \mathit{cs} \Rightarrow X(\mathit{idle}, 0, s_1, n_1) \wedge && (5)\\
&\quad s_1 = \mathit{idle} \Rightarrow X(s_0, n_0, \mathit{waiting}, n_0 + 1) \wedge && (6)\\
&\quad s_1 = \mathit{waiting} \wedge (n_0 = 0 \vee n_1 < n_0) \Rightarrow X(s_0, n_0, \mathit{cs}, n_1) \wedge && (7)\\
&\quad s_1 = \mathit{cs} \Rightarrow X(s_0, n_0, \mathit{idle}, 0) && (8)\\
&\mu Y(n_0{:}N, s_1{:}S, n_1{:}N) = && (9)\\
&\quad ((n_1 = 0 \vee n_0 < n_1) \vee && (10)\\
&\qquad s_1 = \mathit{idle} \vee (s_1 = \mathit{waiting} \wedge (n_0 = 0 \vee n_1 < n_0)) \vee s_1 = \mathit{cs}) \wedge && (11)\\
&\quad s_1 = \mathit{idle} \Rightarrow Y(n_0, \mathit{waiting}, n_0 + 1) \wedge && (12)\\
&\quad s_1 = \mathit{waiting} \wedge (n_1 = 0 \vee n_1 < n_0) \Rightarrow Y(n_0, \mathit{cs}, n_1) \wedge && (13)\\
&\quad s_1 = \mathit{cs} \Rightarrow Y(n_0, \mathit{idle}, 0) && (14)
\end{aligned}
$$

$$\begin{tikzcd}
& (X,(\mathit{waiting},1,\mathit{idle},0)) \arrow[dr] & & (X,(\mathit{cs},1,\mathit{waiting},2)) \arrow[dr] & & (X,(\mathit{waiting},3,\mathit{waiting},2)) \cdots & \\
(X,(\mathit{idle},0,\mathit{idle},0)) \arrow[ur] \arrow[d] & & (X,(\mathit{waiting},1,\mathit{waiting},2)) \arrow[ur] & & (X,(\mathit{idle},0,\mathit{waiting},2)) \arrow[ur] \arrow[dl] & & \\
(Y,(1,\mathit{idle},0)) \arrow[r] & (Y,(1,\mathit{waiting},2)) & & (Y,(3,\mathit{waiting},2)) \arrow[r] & (Y,(3,\mathit{cs},2)) \arrow[r] & (Y,(3,\mathit{idle},0)) \arrow[r] & (Y,(3,\mathit{waiting},4))
\end{tikzcd}$$

Figure 6: Part of the infinite proof graph of the bakery example.

Predicate variable $X$ represents the fact that the property has to hold at any point in time. Therefore, it is labelled with the greatest fixpoint and it encodes the full behaviour of the system in a way very similar to the *Bakery* LPS (lines 3 to 8). When customer 0 picks a number, we check the second half of the property using $Y$ (line 2). For predicate variable $Y$, we assume that customer 0 is in the state *waiting*. Then, $Y$ is *true* if customer 0 can enter the critical section (line 10) or customer 1 does something else after which $Y$ holds (line 11 and lines 12 to 14). However, customer 1 is only allowed to do something finitely often, so the equation for $Y$ is labelled with the least fixpoint. The property holds, since the solution for the initial state is *true*, *i.e.*, $(\mathit{idle}, 0, \mathit{idle}, 0) \in [\![\mathcal{E}]\!](X)$.

There are a few interesting observations that we can make based on this PBES. Firstly, the quantifier that occurs in the mu-calculus formula does not occur in the PBES above; it has been eliminated with automated techniques [5]. This implies that the number $m$ picked by customer 0 is not relevant, which only becomes apparent after constructing the PBES. It is not obvious how one can draw similar conclusions solely based on the LPS and the formula. Secondly, it is not possible to solve this PBES with traditional instantiation-based techniques, since the dependency graph is infinite. Moreover, there is no finite proof graph that contains $(X, (\mathit{idle}, 0, \mathit{idle}, 0))$, so even the application of smart heuristics to guide the instantiation does not improve the situation. See Figure 6 for a part of the infinite proof graph. Lastly, the actual value of $n_0$ and $n_1$ is not essential to the problem. What matters is which of the two is larger. This inspired us to investigate symbolic techniques for solving PBESs.

## 5. Standard and Clustered Recursive Form

To reason symbolically about the underlying dependency graph of a PBES $\mathcal{E}$, we need to rely on the information contained in $\mathcal{E}$. However, for PBESs with an arbitrary structure, that is not trivial [6]. Therefore, we introduce a normal form that simplifies the reasoning about transitions in the underlying proof graph.

A common normal form for Boolean equation systems is *standard recursive form* (SRF) [20]. This normal form is commonly used to translate a BES into a *parity game*, for which efficient solving techniques exist. We generalise the definition to PBESs.

**Definition 12.** Let $\mathcal{E}$ be a PBES. Then $\mathcal{E}$ is in *standard recursive form* (SRF) iff for all $(\sigma_i X_i(d{:}D) = \phi) \in \mathcal{E}$, where $\sigma_i \in \{\mu, \nu\}$, $\phi$ is either disjunctive or conjunctive, *i.e.*, the equation for $X_i$ has the shape

$$\sigma_i X_i(d{:}D) = \bigvee_{j \in J_i} \exists e_j{:}E_j.\, f_j(d, e_j) \wedge X_j(g_j(d, e_j))$$

or

$$\sigma_i X_i(d{:}D) = \bigwedge_{j \in J_i} \forall e_j{:}E_j.\, f_j(d, e_j) \Rightarrow X_j(g_j(d, e_j))$$

Furthermore, we add the semantic restriction that for every $(X, v) \in \mathsf{sig}(\mathcal{E})$, at least one condition $f_j$ should evaluate to *true*, *i.e.*, there is a $j \in J$, a data environment $\delta$ and a $v_j \in \mathbb{E}_j$ such that $[\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d]$ holds.

Standard recursive form is similar to the *parameterised parity game* form of [21]. We call each of the disjuncts or conjuncts of a right-hand side a *clause*. For a PBES $\mathcal{E}$ in SRF, we define a function $\mathsf{op}_{\mathcal{E}} : \mathsf{bnd}(\mathcal{E}) \to \{\wedge, \vee\}$ that indicates for each predicate variable whether its equation is conjunctive or disjunctive. The next proposition states that SRF is a proper normal form, *i.e.*, every PBES can be transformed into SRF while preserving the solution of bound variables.

**Proposition 2.** *For every PBES $\mathcal{E}$, there is an $\mathcal{E}'$ in SRF such that $[\![\mathcal{E}]\!](X) = [\![\mathcal{E}']\!](X)$ for every $X \in \mathsf{bnd}(\mathcal{E})$.*

Proof. For each equation in $\mathcal{E}$ that is not yet of the required form, we can stepwise transform it into one that is. This is done by eliminating nested conjunctions, disjunctions and quantifiers by introducing new predicate variables and extra equations for these variables, see [1]. For instance, an equation that is of the form $(\sigma X(d{:}D) = \forall e{:}E.\, \phi)$ can be replaced by two equations $(\sigma X(d{:}D) = \forall e{:}E.\, Y(d, e))$ $(\sigma Y(d{:}D, e{:}E) = \phi)$ for some fresh variable $Y$. Note that this results in at most a linear blow-up of the size of $\mathcal{E}$, since the number of new equations introduced is at most equal to the number of disjunctive/conjunctive alternations in all right-hand sides of $\mathcal{E}$. Furthermore, the number of new parameters introduced is bounded by the number of variables that occur in quantifiers.

The semantic restriction that at least one clause should be satisfiable can be met by adding the equations $(\nu X_{true}(d{:}D_\star) = X_{true}(\star))$ and $(\mu X_{false}(d{:}D_\star) = X_{false}(\star))$ to $\mathcal{E}$, and adding a clause $X_{true}(\star)$ to every conjunctive right-hand side and a clause $X_{false}(\star)$ to every disjunctive right-hand side. $\square$

We say a formula is in *clustered recursive form* (CRF) iff the predicate variable in each of the clauses is unique, *i.e.*, $X_j \neq X_k$ for all distinct $j, k \in J$. A PBES is in CRF iff all its right-hand sides are CRF formulae. We observe that every PBES can be transformed to CRF by applying Proposition 2 and subsequently combining clauses that have the same predicate variable, relying on suitable projection operators for the data arguments. This is formalised in the next proposition and corollary.

**Proposition 3.** *For every PBES $\mathcal{E}$ in SRF, there is a PBES $\mathcal{E}'$ in CRF such that $[\![\mathcal{E}]\!](X) = [\![\mathcal{E}']\!](X)$ for every $X \in \mathsf{bnd}(\mathcal{E})$.*

Proof. Let $\mathcal{E}$ be a PBES in SRF. Furthermore, let $(\sigma_i X_i(d{:}D) = \varphi_{X_i}) \in \mathcal{E}$ be some equation and $Y \in \mathsf{bnd}(\mathcal{E})$ a predicate variable that occurs multiple times in $\varphi_{X_i}$. We consider the case that $\varphi_{X_i}$ is disjunctive, *i.e.*, it is of the shape $\bigvee_{j \in J_i} \exists e_j{:}E_j.\, f_j(d, e_j) \wedge X_j(g_j(d, e_j))$. The proof for the conjunctive case is analogous. We consider the set of clauses that contain $Y$; their indices are $\{j_1, \ldots, j_n\} \subseteq J_i$. More formally, $Y = X_{j_k}$ for all $1 \leq k \leq n$ and $Y \neq X_j$ for all $j \in J_i \setminus \{j_1, \ldots, j_n\}$. Let $E_n = \{1, \ldots, n\}$ be a sort with $n$ elements and $pr_n$ a polymorphic projection function that has signature $E_n \times T^n \to T$ for any type $T$; it is defined as $pr_n(k, t_1, \ldots, t_k, \ldots, t_n) = t_k$ for all $k \in E_n$. Then, the $n$ clauses for $Y$ can be grouped in one clause as follows:

$$\exists e_n{:}E_n, e_{j_1}{:}E_{j_1}, \ldots, e_{j_n}{:}E_{j_n}.\, pr_n(e_n, f_{j_1}(d, e_{j_1}), \ldots, f_{j_n}(d, e_{j_n})) \wedge Y(pr_n(e_n, g_{j_1}(d, e_{j_1}), \ldots, g_{j_n}(d, e_{j_n})))$$

Unfolding the existential quantifier over $E_n$, applying the definition of $pr_n$ and eliminating unused quantified variables, results in exactly the original set of clauses, so this transformation preserves the solution of $\mathcal{E}$. By applying the same construction to all predicate variables that occur in multiple clauses of the same equation, $\mathcal{E}$ can be rewritten to CRF. $\square$

**Corollary 4.** *For every PBES $\mathcal{E}$, there is an $\mathcal{E}'$ in CRF such that $[\![\mathcal{E}]\!](X) = [\![\mathcal{E}']\!](X)$ for every $X \in \mathsf{bnd}(\mathcal{E})$.*

Proof. Follows from Propositions 2 and 3.

Henceforward we only consider PBESs in CRF[1]. The structure offered by CRF enables us to reason about the edges that exist in proof graphs. Intuitively, an outgoing edge from a node $(X_i, v)$ must be based on some clause $j \in J_i$ whose guard $f_j(v, e_j)$ is *true* for some $e_j$ of sort $E_j$. The target node of that edge is associated to predicate variable instance $X_j(g_j(v, e_j))$. The following definition formalises this.

**Definition 13.** Let $\mathcal{E}$ be a PBES in CRF, where each equation has the same structure as in Definition 12. Then, the *dependency space* of $\mathcal{E}$ is a graph $G = (\mathsf{sig}(\mathcal{E}), E)$, where $E$ is the set satisfying $(X_i, v)E(X_j, w)$ for given $X_i$, $X_j$ for $j \in J_i$, $v$ and $w$ iff for some $\delta$ and $v_j \in \mathbb{E}_j$, both $[\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d]$ and $w = [\![g_j(d, e_j)]\!]\delta[v_j/e_j, v/d]$ hold.

[1]We remark that the theory in this paper can also be applied to PBESs in SRF, but the use of CRF simplifies the presentation.

Definition 13 generalises the definition of a dependency space from [9], since it is applicable to all PBESs (after translation to CRF), not only to disjunctive or conjunctive PBESs. Note that every node in a dependency space has an outgoing edge, since CRF imposes this semantic requirement. This is necessary for the validity of the next lemma.

**Lemma 5.** *The dependency space $G = (\mathsf{sig}(\mathcal{E}), E)$ of $\mathcal{E}$ is both a positive and a negative dependency graph.*

Proof. Let $(X_i, v)$ be a node of $G$. There are four cases that we must consider. Case 1: suppose the equation for $X_i$ is conjunctive and we want to prove that $G$ is a positive dependency graph, *i.e.*, $r$ (from Definition 10) is *true*. From the definition of $\mathsf{env}(S, \mathit{true})$ and Definition 13 we know the following:

$$\begin{aligned} & \mathsf{env}((X_i, v)E, \mathit{true})(X_j) \\ = \;& \{[\![g_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \mid \delta \in \Delta, v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d]\} \end{aligned} \tag{†}$$

Using the definition of the semantics and (†), we can follow the reasoning below to deduce that $[\![\varphi_{X_i}]\!]\eta\delta[v/d] = r$, where $\eta = \mathsf{env}((X_i, v)E, \mathit{true})$.

$$\begin{aligned}
[\![\varphi_{X_i}]\!]\eta\delta[v/d] &= [\![ \bigwedge_{j \in J_i} \forall e_j{:}E_j.\, f_j(d, e_j) \Rightarrow X_j(g_j(d, e_j))]\!]\eta\delta[v/d] \\
&= \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\eta\delta[v_j/e_j, v/d] \Rightarrow [\![X_j(g_j(d, e_j))]\!]\eta\delta[v_j/e_j, v/d] \\
&= \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \Rightarrow [\![g_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \in \eta(X_j) \\
&\overset{(†)}{=} \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \Rightarrow [\![g_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \in \\
&\qquad \{[\![g_j(d, e_j)]\!]\delta'[v_j'/e_j, v/d] \mid \delta' \in \Delta, v_j' \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta'[v_j'/e_j, v/d]\} \\
&= \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \Rightarrow \exists \delta' \in \Delta, v_j' \in \mathbb{E}_j.\, v_j' = v_j \wedge \delta' = \delta \wedge [\![f_j(d, e_j)]\!]\delta'[v_j'/e_j, v/d] \\
&= \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \Rightarrow [\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d] \\
&= \bigwedge_{j \in J_i} \forall v_j \in \mathbb{E}_j.\, \mathit{true} \\
&= r
\end{aligned}$$

With this, the condition on transitions in a positive dependency graph is satisfied. The proofs for the other three combinations are analogous. □

**Theorem 6.** *The dependency space of a PBES $\mathcal{E}$ is the unique smallest dependency graph with $V = \mathsf{sig}(\mathcal{E})$ that is both positive and negative.*

Proof. By contradiction. Let $G = (\mathsf{sig}(\mathcal{E}), E)$ be the dependency space for some PBES $\mathcal{E}$ and let $G' = (\mathsf{sig}(\mathcal{E}), E')$ be a dependency graph that is both positive and negative such that $E \nsubseteq E'$, *i.e.*, $G$ is not strictly smaller than or equal to $G'$. That means that there is at least one edge in $E$ that is missing from $E'$. Let $(X, v)E(Y, w)$ be such an edge. From the definition of a dependency space, we can deduce that there is some $j$ such that $Y = X_j$. Furthermore, for some value of $e_j$, if $d$ has value $v$, the condition $f_j(d, e_j)$ holds and $g_j(d, e_j)$ has value $w$. Therefore, $(X, v)$ depends on $(Y, w)$ in one of two ways:

- In case the equation for $X$ is conjunctive, $Y(w)$ necessarily has to hold in order for $X(v)$ to hold. This is not reflected by $G'$. Therefore $G'$ is not a positive dependency graph, contrary to our assumption.
- In case the equation for $X$ is disjunctive, $Y(w)$ necessarily has to be false in order for $X(v)$ to be false. This is not reflected by $G'$. Therefore $G'$ is not a negative dependency graph, again contrary to our assumption.

We conclude that $G'$ is either not a positive or not a negative dependency graph, which contradicts our initial assumption. $\square$

## 6. Reduced Dependency Space

In the literature, different approaches to solving PBESs have been proposed. Many of those rely on instantiation of the PBES to a finite Boolean equation system. The BES can then be solved with Gaussian elimination [22] or with a parity game solver [7]. However, for PBESs with an underlying infinite BES, instantiation is not possible. Several symbolic approaches have been proposed to reason about the solution of such a PBES. Most notably, Koolen *et al.* [9] use SMT solvers to find proof graphs and Nagae *et al.* [23, 8] compute reduced proof graphs that finitely represent an infinite proof graph. We extend that latter work to arbitrary PBESs and show how a reduced proof graph can be computed, if it is finite, with *PBES quotienting*.

**Definition 14.** Let $G = (V, E)$ be a dependency graph for a PBES $\mathcal{E}$. Then $G' = (V', E')$ is a *reduced dependency graph*, iff:

- $V' \subseteq 2^V$ is a partition of $V$,
- $E' = \{(b, b') \in V' \times V' \mid \exists s \in b, t \in b'.\, s\, E\, t\}$.

We say $G$ is the *base graph* of $G'$.

The intuition behind reduced dependency graphs is that nodes that are in some way equivalent, are grouped. In this way, some infinite dependency graphs can be represented finitely. We again use bisimulation as equivalence relation on nodes. We remark that bisimulation for dependency graphs relies on the labels that are associated with nodes, and not on action labels (cf. Definition 5).

**Definition 15.** Let $G = (V, E)$ be a dependency graph for $\mathcal{E}$. A relation $\mathcal{R} \subseteq V \times V$ is a *bisimulation relation* iff for all $(X, v)\mathcal{R}(Y, w)$:

- $\mathsf{rank}_{\mathcal{E}}(X) = \mathsf{rank}_{\mathcal{E}}(Y)$ and $\mathsf{op}_{\mathcal{E}}(X) = \mathsf{op}_{\mathcal{E}}(Y)$.
- If $(X, v)E(X', v')$, then there is a $(Y', w')$ such that $(Y, w)E(Y', w')$ and $(X', v')\mathcal{R}(Y', w')$.
- If $(Y, w)E(Y', w')$, then there is a $(X', v')$ such that $(X, v)E(X', v')$ and $(X', v')\mathcal{R}(Y', w')$.

Nodes $(X, v)$ and $(Y, w)$ are bisimilar, denoted $(X, v) \leftrightarroweq (Y, w)$, iff they are related by some bisimulation relation. Two graphs $G$ and $H$ are bisimilar iff for every node in $G$ there is a bisimilar node in $H$ and vice versa.

Remark that two nodes $(X, v)$ and $(Y, w)$ may be bisimilar even though they originate from different equations in the PBES, as long as they have the same rank and operand. Since bisimilarity is an equivalence relation it induces a partition of the node set $V$ into equivalence classes. We call the reduced dependency graph $G_r = (V/{\leftrightarroweq}, E_r)$, that has $G$ as its base graph (cf. Definition 14), the *bisimulation quotient* of $G$, notation $G/{\leftrightarroweq}$.

Using the CRF normal form and the notions of a (reduced) dependency space and bisimulation, we now have a setting similar to Section 2.1. Procedure 1 can thus be applied with only minor changes, see Procedure 2. First, in case we are interested in the solution of a particular node $(\hat{X}, \hat{v})$, called the *node of interest*, we should provide it as input. The node of interest plays the same role as the initial state $\hat{d}$ does for an LPS. Alternatively, if we want to solve the complete PBES, we should skip the reachability check (line 6) in every iteration. Second, we need an adapted definition of the initial partition and the splitting operations. In the PBES setting, the initial partition needs to distinguish nodes that have a different rank or operand (first bullet of Definition 15), so $\pi_0$ is set to $\{\{(X, v) \in V \mid v \in \mathbb{D} \wedge \mathsf{rank}_{\mathcal{E}}(X) = \mathsf{rank}_{\mathcal{E}}(Y) \wedge \mathsf{op}_{\mathcal{E}}(X) = \mathsf{op}_{\mathcal{E}}(Y)\} \mid Y \in \mathsf{bnd}(\mathcal{E})\}$. The *split* and *co-split* functions no longer have an argument that defines the action on which the split is

**Procedure 2:** PBES Quotienting

```
Input: PBES $\mathcal{E}$, initial partition $\pi_0$, node of interest $(\hat{X}, \hat{v})$
1 $i := 0$;
2 while $\pi_i$ is not stable do
3     $i := i + 1$;
4     $\pi_i := (\pi_{i-1} \setminus \{k\}) \cup \{split(k, k'), co\text{-}split(k, k')\}$ for some $k, k' \in \pi_{i-1}$ such that $split(k, k')$ and $co\text{-}split(k, k')$
          are non-empty;
5     $\rightarrow_i := \{(k, k') \mid \exists (X_i, v) \in \mathcal{X} \times \mathbb{D}. [\![k(X_i, d) \wedge \bigvee_{j \in J_i} (\exists e_j{:}E_j.\, f_j(d, e_j) \wedge k'(X_j, g_j(d, e_j)))]\!]\delta[v/d]\}$;
6     $\pi_i := \{k \mid \hat{k} \rightarrow_i^* k\}$ where $(\hat{X}, \hat{v}) \in \hat{k}$;
7 return $(\pi_i, \rightarrow_i, \hat{k})$ where $(\hat{X}, \hat{v}) \in \hat{k}$;
```

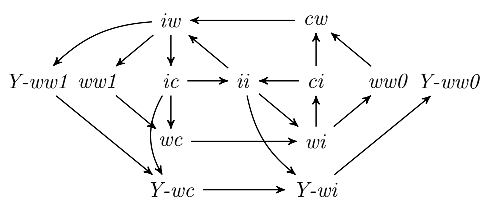

Figure 7: Equivalence classes and transitions in the reduced dependency space of the bakery protocol example.

based. Below, we use the index $i$ of the predicate variable $X_i$ bound in the lambda to construct an expression based on the right-hand side of $X_i$.

$$split(k, k') = \lambda X_i \in \mathcal{X}, v \in \mathbb{D}. [\![k(X_i, d) \wedge \bigvee_{j \in J_i} \exists e_j{:}E_j.\, f_j(d, e_j) \wedge k'(X_j, g_j(d, e_j))]\!]\delta[v/d]$$
$$co\text{-}split(k, k') = \lambda X_i \in \mathcal{X}, v \in \mathbb{D}. [\![k(X_i, d) \wedge \neg \bigvee_{j \in J_i} \exists e_j{:}E_j.\, f_j(d, e_j) \wedge k'(X_j, g_j(d, e_j))]\!]\delta[v/d]$$

**Example 6.** We revisit the bakery protocol example from Section 3. Running Procedure 2 on that PBES yields a finite reduced dependency space (depicted in Figure 7) which contains 14 reachable equivalence classes. Here, we abbreviated state names. For example, in state *wi*, process 0 is waiting and process 1 is idle. Furthermore, in state *ww0*, both processes are waiting, but process 0 has preference to enter the critical section first. States belonging to predicate variable $Y$ are prefixed with *Y-*. We omitted the state containing $X_{true}$ for simplicity (cf. proof of Proposition 2). Note the symmetry between process 0 and process 1 in those states belonging to variable $X$ and also the parallels between $X$ and $Y$. □

Procedure 2 can be used to solve a PBES as follows. Upon its termination, a Boolean equation system or a parity game can be generated from the stable partition; either of them then finitely represents the dependency graph of the original PBES. For both possible types of outputs, there are existing solvers that can compute their solution. From this solution we can then derive the solution to the original PBES.

**Example 7.** Consider again the reduced dependency space of Example 6. This reduced dependency space contains only one positive reduced proof graph, *viz.* the graph containing all nodes, and no negative reduced proof graph. This implies that the original dependency space also contains only a positive proof graph. We conclude that the solution for the node $(X, (idle, 0, idle, 0))$ is *true*, and that the formula holds, *i.e.*, the bakery protocol does not cause starvation of customers. □

We next formalise these steps. Since we are reasoning in the context of partition refinement, we know that all partitions that are finer than $\pi_0$ are (by definition of $\pi_0$ given above) such that all nodes in a block have the same rank and operand. We call a partition with this property *consistent*. We say that a reduced dependency graph is consistent iff its set of vertices is a consistent partition. For a consistent reduced dependency graph $G$ with associated PBES $\mathcal{E}$, we overload the $\mathsf{op}$ and $\mathsf{rank}$ functions by defining them on blocks, such that $\mathsf{op}(b) = \mathsf{op}_{\mathcal{E}}(X)$ and $\mathsf{rank}(b) = \mathsf{rank}_{\mathcal{E}}(X)$ for some $(X, v) \in b$. The following definition shows how to construct a BES (in CRF) for a consistent reduced dependency graph.

**Definition 16.** Let $G = (V, E)$ be a consistent reduced dependency graph of a PBES $\mathcal{E}$. The *induced* Boolean equation system, denoted $\mathcal{E}_G$, is the BES containing, per block $b \in V$, exactly one equation $(\sigma_b X_b(d{:}D_\star) = \phi_b)$ such that:

- $\mathsf{rank}_{\mathcal{E}_G}(X_b) = \mathsf{rank}(b)$,
- If $\mathsf{op}(b) = \wedge$ then $\phi_b = \bigwedge_{(b,b') \in E}(\mathit{true} \Rightarrow X_{b'}(\star))$,
- If $\mathsf{op}(b) = \vee$ then $\phi_b = \bigvee_{(b,b') \in E}(\mathit{true} \wedge X_{b'}(\star))$.

Before we state several interesting properties of an induced BES, we introduce one additional notion. Given two (reduced) dependency graphs $G$ and $G'$, we say $G$ and $G'$ are *rank-operand-isomorph* when there is an isomorphism between them that preserves the $\mathsf{rank}$ and $\mathsf{op}$ functions.

The following lemma formalises that the BES induced by a consistent reduced dependency graph is a correct representation of the reduced dependency graph.

**Lemma 7.** *Let $G$ be a consistent reduced dependency graph of a PBES $\mathcal{E}$ and $\mathcal{E}_G$ be the induced BES. Then, the dependency space of $\mathcal{E}_G$ is rank-operand-isomorph to $G$.*

Proof. Let $G = (V, E)$ be a consistent reduced dependency graph of a PBES $\mathcal{E}$ and $\mathcal{E}_G$ be the induced BES. Furthermore, let $G' = (V', E')$ be the dependency space of $\mathcal{E}_G$. Let $R : V \to V'$ be defined as $R(b) = (X_b, \star)$ for all $b \in V$. We will show that $R$ is an isomorphism. Clearly, $R$ is injective, since $R(b) = R(b') \Rightarrow (X_b, \star) = (X_{b'}, \star) \Rightarrow b = b'$. Surjectivity of $R$ follows from the definition of $V'$, which is $V' = \mathsf{sig}(\mathcal{E}_G) = \{X_b \mid b \in V\} \times D_\star$ (Definition 16). So for every element $(X_b, \star) \in V'$ we have $R(b) = (X_b, \star)$. We conclude that $R$ is bijective.

Let $b \in V$ be some block. The equation for $X_b$ has the same rank and operand as $b$ (Definition 16) and thus $(X_b, \star)$ also has the same rank and operand. Therefore, $R$ preserves the rank and operand.

What remains is to show that $R$ preserves the edge relation, *i.e.*, for all nodes $b, b' \in V$, $b\,E\,b'$ if and only if $R(b)E'R(b')$.

$\Leftarrow$ Let $b, b' \in V$ be two blocks satisfying $R(b)E'R(b')$, *i.e.*, $(X_b, \star)E'(X_{b'}, \star)$. According to Definition 13, this implies that there is a $j \in J_b$ such that $X_j = X_{b'}$ (the other conditions of Definitions 13 are trivially true in the context of $\mathcal{E}_G$). From the definition of the equations of $\mathcal{E}_G$ (Definition 16), we deduce that this can only be the case if $b\,E\,b'$.

$\Rightarrow$ Let $b, b' \in V$ be two blocks satisfying $b\,E\,b'$. Then, the equation for $X_b$ in $\mathcal{E}_G$ contains a clause that has $X_{b'}(\star)$ as predicate variable (Definition 16). From Definition 13, it follows that $(X_b, \star)E'(X_{b'}, \star)$ and thus we conclude that $R(b)E'R(b')$. □

The following theorem states that the solution to the BES that is induced by the bisimulation quotient of the dependency space of a PBES $\mathcal{E}$, preserves and reflects the solution to that PBES.

**Theorem 8.** *Let $\mathcal{E}$ be a PBES, $G = (V, E)$ be the dependency space of $\mathcal{E}$ and $\mathcal{E}'$ the BES induced by ${}^{G}/_{\leftrightarrow}$. Then, $v \in [\![\mathcal{E}]\!](X)$ iff $[\![\mathcal{E}']\!](X_b) = \{\star\}$, where $(X, v) \in b$.*

Proof. This result follows directly from Theorem 1, Lemma 7, the reasoning that bisimulation reduction preserves bisimilarity and that bisimilarity is a *consistent correlation* [24], *i.e.*, bisimilarity preserves and reflects the solution of a PBES.

We also sketch an alternative proof that does not rely on the notion of consistent correlation. We restrict ourselves to a proof of the positive case, the negative case is analogous. Let $\mathcal{E}$, $G$ and $\mathcal{E}'$ be as above. Without loss of generality, let $G'$ be a positive proof graph of $\mathcal{E}'$ such that $G'$ is a subgraph of the dependency space of $\mathcal{E}'$. From Lemma 7, it follows that there is a subgraph $G_r^+ \subseteq {}^{G}/_{\leftrightarrow}$ that is rank-operand-isomorph to $G'$. Such a subgraph also exists in $G$, since $G$ and ${}^{G}/_{\leftrightarrow}$ are bisimilar. Let us call this subgraph $G^+$. Furthermore, these subgraphs can represent exactly the same nodes: some $(X_b, \star)$ is in $G'$ implies $b$ is in $G_r^+$ implies $(X, v)$ is in $G^+$, where $(X, v) \in b$. Since $G'$ and $G^+$ have exactly the same paths, $G^+$ is a proof graph of $\mathcal{E}$, and thus $v \in [\![\mathcal{E}]\!](X)$. □

We remark that the procedure presented in this section generalises the procedures presented by Nagae *et al.* in [23] and [8], which only apply to PBESs consisting of predicate formulae that contain no predicate variables within the scope of universal quantifiers.

## 7. Stable Kernel

The approach presented in the previous section terminates when the reachable part of the bisimulation quotient is finite and all the operations on data are decidable. However, we are also interested in solving PBESs for which the bisimulation quotient is not finite. Therefore, we propose an improvement that allows for reasoning about the solution of a single node $(X, v)$, even when some part of the dependency space is not finitely representable. This is illustrated by the following example.

**Example 8.** Consider the following PBES:

$$\begin{aligned}
&\nu X(n{:}N) = X(n+1) \lor (n = 0 \land Y(0)) \\
&\mu Y(n{:}N) = Y(n+1) \land (n = 0 \Rightarrow X(0)) \land (n > 1 \Rightarrow Y(n-1))
\end{aligned}$$

The (stable) bisimulation quotient of the dependency space of this PBES is infinite and looks as follows:

$$\begin{tikzcd}
\{(X,0)\} \arrow[r] \arrow[d, shift left] & \{(X,n) \mid n \geq 1\} \arrow[loop right] & & \\
\{(Y,0)\} \arrow[u, shift left] \arrow[r, bend left] & \{(Y,1)\} \arrow[l, bend left] \arrow[r, bend left] & \{(Y,2)\} \arrow[l, bend left] \arrow[r, bend left] & \cdots \arrow[l, bend left]
\end{tikzcd}$$

While this reduced dependency graph is infinite, there is a finite reduced proof graph for $X(0)$, namely the subgraph that only contains the blocks $\{(X, 0)\}$ and $\{(X, n) \mid n \geq 1\}$. Therefore, to draw conclusions about the solution for $X(0)$, it is not necessary to refine the part of the partition that concerns $Y$. $\square$

The example suggests that we may in general search for a proof graph in a—not yet stable—reduced dependency graph and use that to partially solve a PBES. However, not every proof graph obtained that way necessarily induces a proper proof graph for the original PBES: stability of the subgraph representing the proof graph is required. This is formalised by the concept of a *stable kernel*.

**Definition 17.** Let $G = (V, E)$ be a dependency graph of a PBES $\mathcal{E}$ and $G_r = (V_r, E_r)$ a consistent reduced dependency graph of $G$. Furthermore, let $G'_r = (V'_r, E'_r)$ be a subgraph of $G_r$. Then, $G'_r$ is a *stable kernel* of $G_r$ if and only if $V'_r$ is stable with respect to itself.

The following lemma and theorem show how stable kernels can help to identify a proof graph in a partially stable reduced dependency graph.

**Lemma 9.** *Let $G = (V, E)$ be a dependency graph of a PBES $\mathcal{E}$ and $G_r = (V_r, E_r)$ a consistent reduced dependency graph of $G$. Furthermore, let $G'_r = (V'_r, E'_r)$ be a stable kernel of $G_r$. Then $G'_r$ is bisimilar to its base graph $G' = (V', E')$.*

Proof. The situation from Lemma 9 is depicted in the figure below.

$$\begin{tikzcd}
G = (V, E) \arrow[r, "\text{base graph of}"] \arrow[d, phantom, "\cup\!\mid"] & G_r = (V_r, E_r) \arrow[d, phantom, "\cup\!\mid"] \\
G' = (V', E') \arrow[r, "\text{base graph of}"] & G'_r = (V'_r, E'_r)
\end{tikzcd}$$

For bisimilarity of $G'_r$ and $G'$ we reason as follows. First, note that $V' = \bigcup V'_r$ (see Definition 14). Let $\mathcal{R} \subseteq V' \times V'_r$ be a relation defined as $\mathcal{R} = \{((X, v), b) \mid (X, v) \in b\}$. We will show that $\mathcal{R}$ is a bisimulation relation.

Pick an arbitrary $(X, v)\mathcal{R}b$. By definition, we have $(X, v) \in b$. Note that $G'_r$ is consistent due to consistency of $G_r$; and we find that both the rank and operand of the equation for $X$ match the rank and operand of the equation for $X_b$ in the BES induced by $G'_r$. For the transfer conditions we observe the following:

- From the definition of a reduced graph it follows directly that if $(X, v)E'(Y, w)$, then $b\, E'_r\, b'$, where $(Y, w) \in b'$, *i.e.*, $(Y, w)\mathcal{R}b'$.
- Suppose we have $b\, E'_r\, b'$. Since $V'_r$ is stable, in particular $b \in V'_r$ is stable with respect to $b' \in V'_r$. Since all nodes in a stable block have the same transitions, $b\, E'_r\, b'$ implies that there must be a node $(Y, w) \in b'$ such that $(X, v)E'(Y, w)$. Moreover, $(Y, w) \in b'$ implies the required $(Y, w)\mathcal{R}b'$.

It follows that $G'$ is bisimilar to $G'_r$. □

Although Lemma 9 gives a strong indication that a stable kernel accurately represents its base graph $G'$, we cannot conclude immediately that $G'$ is a valid dependency graph or even proof graph. Sufficient conditions for $G'$ to be a proof graph are stated in the theorem below. This theorem is the basis for the correctness of Procedure 3, which we will present below.

**Theorem 10.** *Let $G = (V, E)$ be a dependency graph for a PBES $\mathcal{E}$ and $G_r = (V_r, E_r)$ a consistent reduced dependency graph of $G$. Furthermore, let $G'_r = (V'_r, E'_r)$ be a stable kernel of $G_r$ and $G' = (V', E')$ the base graph of $G'_r$. If $G'_r$ is rank-operand-isomorph with a proof graph of the BES induced by $G_r$, then $G'$ is a proof graph for $\mathcal{E}$.*

Proof. The situation is depicted in the figure below. Here, $H$ is the dependency space of $\mathcal{E}'$, the BES induced by $G_r$, and $H'$ is a proof graph for $\mathcal{E}'$.

$$\begin{tikzcd}
G = (V, E) \arrow[r, "\text{base graph of}"] & G_r = (V_r, E_r) \arrow[rr, dash, "\text{ro-isomorph}"] \arrow[dr, "\text{ind. BES}"] & & H = (V_H, E_H) \\
\rotatebox{90}{$\subseteq$} & \rotatebox{90}{$\subseteq$} & \mathcal{E}' \arrow[ur, "\text{dep. space}"] \arrow[dr, "\text{pr. graph}"] & \rotatebox{90}{$\subseteq$} \\
G' = (V', E') \arrow[r, "\text{base graph of}"] & G'_r = (V'_r, E'_r) \arrow[rr, dash, "\text{ro-isomorph}"] & & H' = (V'_H, E'_H)
\end{tikzcd}$$

We observe that $G'$ trivially satisfies the path conditions of a proof graph, since it has exactly the same infinite paths as $H'$. In the following, we reason that $G'$ also satisfies the conditions of a dependency graph.

We assume that $G'$ is not a dependency graph. Then there must be a 'missing' edge that violates the conditions of a dependency graph. Let $((X, v), (Y, w)) \notin E'$ be such an edge, *i.e.*, $(X, v) \in V'$ and $[\![\phi_X]\!]\eta'(\delta[v/d_X]) \neq r = [\![\phi_X]\!]\eta(\delta[v/d_X])$ for some $r \in \mathbb{B}$, where $\eta' = \mathsf{env}((X, v)E', r)$ and $\eta = \mathsf{env}((X, v)E' \cup \{(Y, w)\}, r)$. From bisimilarity with $G'_r$ (see Lemma 9), it follows that $(b, b') \notin E'_r$, where $(X, v) \in b$ and $(Y, w) \in b'$. Furthermore, the corresponding edge is also missing from $H'$.

Since the presence of the edge $((X, v), (Y, w))$ is necessary to satisfy the condition on dependency graphs, it must be present in $G$. As per the definition of a reduced graph, it is also present in $G_r$, *i.e.*, $(b, b') \in E_r$, where $(X, v) \in b$ and $(Y, w) \in b'$. Thus, the corresponding edge is also present in $H$: $((X_b, \star), (X_{b'}, \star)) \in E_H$.

We now analyse whether $H'$ is indeed a valid proof graph for $\mathcal{E}'$. There are two possible cases:

- $\mathsf{op}_{\mathcal{E}}(X) = \wedge$ and $r = \mathit{true}$ or $\mathsf{op}_{\mathcal{E}}(X) = \vee$ and $r = \mathit{false}$. In this case, any proof graph that contains the node $(X_b, \star)$ must also contain the edge $((X_b, \star), (X_{b'}, \star))$. This contradicts the earlier claim that this edge is missing from $H'$.

- $\mathsf{op}_{\mathcal{E}}(X) = \wedge$ and $r = \mathit{false}$ or $\mathsf{op}_{\mathcal{E}}(X) = \vee$ and $r = \mathit{true}$. If $X_{b'}$ is the only predicate variable in the right-hand side of $X_b$, then it is indeed necessary to include $((X_b, \star), (X_{b'}, \star))$ in $E'_H$ whenever $V'_H$ contains $(X_b, \star)$. The earlier claim that this edge is missing is again contradicted.

  If there are more predicate variables in the right-hand side of $X_b$, then there are also multiple successors of $(X, v)$ in $G$. This can be derived from the stability of $b$ with respect to $b'$. Therefore, the edge $((X, v), (Y, w))$ was not required to be in $E$, which contradicts our initial assumption.

We derive that $G'$ satisfies the conditions of a dependency graph. □

The following example illustrates that the assumption "$G'_r$ is a stable kernel" from Theorem 10 is a necessary condition.

**Example 9.** Consider the PBES $(\nu X(n{:}N) = ((n \neq 0) \wedge X(n)) \vee Y)(\mu Y = Y)$. The figures below depict the initial partition of the dependency space of this PBES (on the left-hand side) and the stable partition (on the right-hand side).

$$\begin{tikzcd}
\{(X,n) \mid n \in \mathbb{N}\} \arrow[loop above] \arrow[r] & \{Y\} \arrow[loop right]
\end{tikzcd}
\qquad
\begin{tikzcd}
\{(X,0))\} \arrow[dr] & \\
 & \{Y\} \arrow[loop right] \\
\{(X,n) \mid n \neq 0\} \arrow[loop above] \arrow[ur] &
\end{tikzcd}$$

In the initial partition, there is a positive reduced proof graph that contains $(X, 0)$, *viz.* the graph containing only $\{(X, n) \mid n \in \mathbb{N}\}$. Note that this block is not stable with respect to itself. In contrast, in the stable partition, there is only a negative proof graph for $(X, 0)$. This shows that a reduced proof graph that is not stable with respect to itself can in general not be used to draw conclusions about the solution of the PBES under consideration. □

**Procedure 3:** PBES quotienting with stable kernels

```
   Input: PBES $\mathcal{E}$, initial partition $\pi_0$, node of interest $(\hat{X}, \hat{v})$
1  $\rho_0 := \emptyset$;
2  $i := 0$;
3  while $\pi_i$ is not a stable kernel do
4      $q := (\pi_i \setminus \{k\}) \cup \{\mathit{split}(k, k'), \mathit{co\text{-}split}(k, k')\}$ for some $k, k' \in \pi_i$ such that $\mathit{split}(k, k')$ and $\mathit{co\text{-}split}(k, k')$ are
           non-empty;
5      $q := q \cup \rho_i$;
6      $E_{i+1} := \{(k, k') \mid \exists (X_i, v) \in \mathcal{X} \times \mathbb{D}. [\![k(X_i, d) \wedge \bigvee_{j \in J_i} (\exists e_j{:}E_j \,.\, f_j(d, e_j) \wedge k'(X_j, g_j(d, e_j)))]\!]\delta[v/d]\}$;
7      $q := \{k \mid \hat{k} E^*_{i+1} k\}$ where $(\hat{X}, \hat{v}) \in \hat{k}$;
8      $(\pi_{i+1}, \rho_{i+1}) :=$ findProofGraph($q$, $E_{i+1}$, $\hat{k}$) where $(\hat{X}, \hat{v}) \in \hat{k}$;
9      $i := i + 1$;
10 return $(\pi_i, E_i)$;
```

Based on the theory of stable kernels, we propose the following changes to our approach: after every iteration, we search for a proof graph in the current partition. In the next iteration, only the blocks that are contained in the proof graph will be refined. When the blocks in the proof graph are stable with respect to each other, we have found a stable kernel and the procedure can terminate (by Theorem 10). See Procedure 3. We maintain two sets of blocks: $\pi_i$ contains the blocks in the proof graph that we are currently considering and $\rho_i$ contains the other blocks. At line 4, we split a block in $\pi_i$ and temporarily store the resulting partition in $q$. Then, the set of blocks of the whole partition, reachable under the new transition relation from the block containing the node of interest $(X, v)$ is computed (lines 5 to 7). Thereby, blocks that are not reachable from the node of interest are effectively "thrown away", *i.e.*, they are not considered during the next iterations. Since unreachable blocks cannot be part of a minimal proof graph for the node of interest, this does not affect the correctness of the procedure. From the reachable blocks, we extract a proof graph for the node of interest

$(X, v)$ (line 8). Searching for a proof graph (function `findProofGraph`) can be done with existing algorithms, such as a solver for Boolean equations systems or for parity games, *e.g.*, Zielonka's recursive algorithm [25]. The blocks contained in the proof graph are again stored in $\pi_i$, the remaining blocks are stored in $\rho_i$. After every iteration, we check whether $\pi_i$ is a stable kernel (line 3). If so, the procedure terminates.

## 8. Stability Under Solution

The procedures presented so far rely purely on bisimulation as the notion of equivalence. However, we are only interested in the solution of the PBES. Bisimulation is much stronger than solution equivalence, resulting in a reduced dependence space that is larger than necessary to determine the solution. Besides bisimulation, several other equivalences have been defined in the literature, such as *consistent correlation* [24] and the corresponding preorder *consistent consequence* [26]. This inspired us to investigate how our techniques can benefit from a weaker equivalence relation.

**Example 10.** Consider the following PBES $\mathcal{E}$ with node of interest $(X, \mathit{true})$:

$$\begin{aligned} \mu X(b{:}B) &= (b \wedge Y) \vee (\neg b \wedge Z) \\ \nu Y &= X(\mathit{true}) \\ \mu Z &= Z \end{aligned}$$

The initial and stable partition of the dependency space for $\mathcal{E}$ are respectively:

$$\begin{tikzcd} & \{Y\} \arrow[dl, bend right] & \{(X,\mathit{true})\} \arrow[r, bend left] & \{Y\} \arrow[l, bend left] \\ \{(X,\mathit{true}),(X,\mathit{false})\} \arrow[ur, bend right] \arrow[dr] & & & \\ & \{Z\} \arrow[loop right] & \{(X,\mathit{false})\} \arrow[r] & \{Z\} \arrow[loop right] \end{tikzcd}$$

In the initial partition, there is one (negative) proof graph which contains the initial block, *viz.*, the graph that contains all blocks. Since the transition relation of a reduced dependency graph is an over-approximation (cf. Definition 14, existential quantifier in the second bullet), after splitting the block $\{(X, \mathit{true}), (X, \mathit{false})\}$, each of the resulting blocks will either have the same or fewer outgoing transitions. More formally, $[\![(b \wedge Y) \vee (\neg b \wedge Z)]\!]\eta\delta \Rightarrow [\![(b \wedge Y) \vee (\neg b \wedge Z)]\!]\eta\delta'$ for all $\delta$ and $\delta'$, where $\eta = [\![\mathcal{E}]\!]$. In other words, since all possible successors of a node $(X, b)$, *viz.* $Y$ and $Z$, have the solution *false*, the actual value of $b$ is of no influence on the value of the right-hand side of $X$. In general, in the setting of a negative reduced proof graph, removing outgoing edges of disjunctive blocks preserves the validity of that proof graph (as long as each node has at least one outgoing edge).

We formalise this observation in the following definition.

**Definition 18.** Let $\mathcal{E}$ be a PBES and $G_r = (V_r, E_r)$ be a consistent reduced dependency space and $\mathcal{E}_{G_r}$ its associated BES. A block $b \in V_r$ is *stable under solution* iff at least one of the following conditions holds:

- $b$ is stable under bisimulation.
- If $\mathsf{op}(b) = \wedge$, then $[\![\mathcal{E}_{G_r}]\!](X_b) = \{\star\}$ and if $\mathsf{op}(b) = \vee$, then $[\![\mathcal{E}_{G_r}]\!](X_b) = \emptyset$.

A reduced dependency space is stable under solution iff all its blocks are stable under solution.

The intuition behind this is that conjunctive blocks for which the solution is *true* do not have to be split according to bisimulation, since that will not change their solution. Dually, disjunctive blocks do not have to be split in negative reduced proof graphs. We now proceed by proving that stability under solution is a sufficient condition to preserve the solution. In the proof below, recall that $vE$ denotes the successor set of $v$ under the transition relation $E$ (cf. Definition 10).

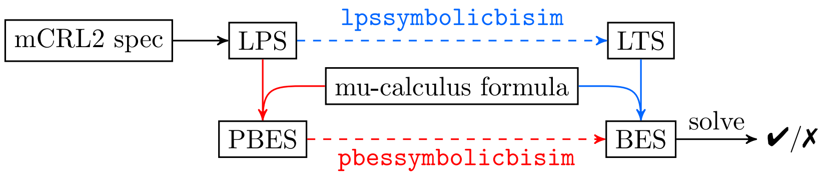

Figure 8: Workflow for model checking based on minimal model generation (in blue) and PBES quotienting (in red).

**Theorem 11.** *Let $\mathcal{E}$ be a PBES, $G_r = (V_r, E_r)$ a reduced dependency space that is stable under solution and $\mathcal{E}_{G_r}$ the associated BES of $G_r$. Then, $v \in [\![\mathcal{E}]\!](X)$ iff $[\![\mathcal{E}_{G_r}]\!](X_b) = \{\star\}$, where $(X, v) \in b$.*

Proof. Let $\mathcal{E}$ be a PBES, $G_r = (V_r, E_r)$ a reduced dependency space that is stable under solution and $\mathcal{E}_{G_r}$ the associated BES of $G_r$. Here we provide the proof for the positive case, *i.e.*, we show that $[\![\mathcal{E}_{G_r}]\!](X_b) = \{\star\}$ implies $v \in [\![\mathcal{E}]\!](X)$. The proof for the negative case, *viz.* $[\![\mathcal{E}_{G_r}]\!](X_b) = \emptyset$ implies $v \notin [\![\mathcal{E}]\!](X)$, is completely analogous.

Let $R$ be a rank-operand-isomorphism between $G_r$ and the dependency space of $\mathcal{E}_{G_r}$ (its existence is stated in Lemma 7). Furthermore, let $G'$ be a positive proof graph of $\mathcal{E}_{G_r}$ such that $G'$ is a subgraph of the dependency space of $\mathcal{E}_{G_r}$. The subgraph of $G_r$ that is rank-operand-isomorphic to $G'$ under $R$ is called $G_r^+$. Since $[\![\mathcal{E}_{G_r}]\!](X_b) = \{\star\}$ for all blocks $b$ in $G_r^+$, it has to hold that $b$ is either conjunctive, *i.e.*, $\mathsf{op}(b) = \wedge$, or $b$ is stable under bisimulation. Let $G^+$ be the base graph of $G_r^+$ (cf. Definition 14). We deduce that $G^+$ is a positive proof graph of $\mathcal{E}$. Let $(X, v)$ be an arbitrary node in $G^+$ and $b$ a block in $G_r^+$ such that $(X, v) \in b$.

- $\mathsf{op}(b) = \vee$. Since $b$ must be stable under bisimulation, $(X, v)$ has similar outgoing edges as $b$, *i.e.*, for all $b'$ such that $bE_r^+b'$, there exists a node $(X', v') \in b'$ such that $(X, v)E^+(X', v')$. There must be at least one block $b' \in bE_r^+$, because $\mathcal{E}_{G_r}$ is in SRF. Therefore, there is also at least one node $(X', v') \in (X, v)E^+$. Using disjunctivity of $\varphi_X$, we conclude that $[\![\varphi_X]\!](\mathsf{env}((X, v)E^+, \mathit{true}))\delta[v/d_X] = \mathit{true}$.
- $\mathsf{op}(b) = \wedge$. Since $X_b$ is conjunctive, $G'$ must contain all predicate variables occurring on the right-hand side of $X_b$. Therefore, $bE_r = bE_r^+$, and thus $(X, v)E_r = (X, v)E_r^+$. Based on the definition of a dependency space and the fact that $\varphi_X$ is conjunctive, we can conclude that $[\![\varphi_X]\!](\mathsf{env}((X, v)E^+, \mathit{true}))\delta[v/d_X] = \mathit{true}$.

We conclude that $G^+$ is a positive dependency graph. Since the set of infinite paths in $G^+$ is a subset of the infinite paths in $G'$, we know that $G^+$ also satisfies the path condition on proof graphs. Therefore, $G^+$ is a valid proof graph containing $(X, v)$ and thus it holds that $v \in [\![\mathcal{E}]\!](X)$ (Theorem 1). □

The idea of stability under solution can be applied as an early termination heuristic. In Procedure 3, the stability check on line 3 can be implemented with solution stability.

## 9. Implementation and Experiments

We implemented the ideas presented in the previous sections in two tools that are part of the mCRL2 toolset [3]. The first tool, `lpssymbolicbisim`, performs minimal model generation on LPSs according to Procedure 1. Upon termination, it produces an LTS. The second tool, called `pbessymbolicbisim`, implements PBES quotienting (cf. Section 6) and also the optimisations identified in Sections 7 and 8.

Figure 8 shows an overview of the workflow for both tools in the setting of the mCRL2 toolset. In case we want to perform model checking of mu-calculus formulae, the mCRL2 specification is first transformed to an LPS. Then, one can choose to use `lpssymbolicbisim` to obtain an LTS, and subsequently use the property to construct a BES which can be solved. Alternatively, one first constructs a PBES and then applies `pbessymbolicbisim` to obtain the BES.

Both `lpssymbolicbisim` and `pbessymbolicbisim` call the Z3 SMT-solver to determine whether one of the sets computed with the functions *split* and *co-split* is empty, *i.e.*, whether its characteristic function is unsatisfiable. When choosing which block to split in each iteration (line 4 of Procedure 1 and line 4 of

Procedure 3), preference is given to blocks that are the least far away from the block containing the node of interest.

A critical component of the implementation is the handling of characteristic functions. To manipulate these expressions, we first of all rely on the mCRL2 term rewrite system. Furthermore, the tools also contain several specialised algorithms that simplify the characteristic functions after splitting a block. Our experience is that it is worthwhile spending some runtime on these simplifications. If the characteristic functions are never minimised, they contain a lot of redundancy and quickly grow prohibitively large. This slows down any subsequent computation by the procedure.

We compare the performance of several approaches: our implementations of minimal model generation and PBES quotienting and the `pbes-cvc4` tool from [9]. We originally also aimed to compare with the tool `PBESSolver` from [8]. However, their implementation has several practical limitations, making a fair comparison impossible. We therefore decided to exclude `PBESSolver` from our experiments. The experiments were performed on a machine with an Intel Core i5 3350P processor and 8 GB of memory running Ubuntu 18.04 and mCRL2 commit hash 066ba9f36b[2] compiled with GCC7.3.

Our set of benchmarks[3] consists of various PBESs that encode different types of decision problems, covering typical linear-time, branching-time and real-time model checking problems, a scheduling problem, recursive functions and behavioural equivalence checking problems. The PBESs encoding model checking problems mostly originate from the set of examples included in mCRL2, which in some cases have been modified to generate infinite state spaces. Classical approaches that generate the state space explicitly fail for all of these models. We remark that most of the models contain multiple concurrent processes. Each model is combined with one or more formal properties in the form of a modal mu-calculus formula to obtain a PBES. More specifically, we verified the following properties:

- two *reachability* properties (the real-time ball game: winning impossible; and the real-time train gate system: action *go(1)* can be executed at time 20);
- two *invariants* (Fischer's real-time mutual exclusion protocol and Lamport's bakery protocol: no deadlock);
- six linear and branching-time properties (the ball game: infinitely often put ball; the train gate: fairness; Fischer's protocol and Lamport's bakery protocol: request must be served; the Concurrent Alternating Bit Protocol (CABP): a message can be received infinitely often; Hesselink's handshake register [27]: cache consistency, and all writes finish).

The scheduling problem we consider is due to [23]; it encodes a fair trading problem encoded as a PBES. Furthermore, two recursive functions we consider are based on classical benchmarks for verification tools [28]. A modified version of the McCarthy 91 function, as per [8], is represented with the following PBES:

$$\mu M(x, y{:}N) = (x > 10 \land x = y + 1) \lor \exists e{:}N.\, x \leq 10 \land M(x+2, e) \land M(e, y)$$

Here, $M(x, y)$ is *true* if and only if $(x, y)$ is a solution for the function we represent. In a similar fashion, we have a PBES for Takeuchi's function [28]:

$$\begin{aligned}\mu T(x, y, z, w{:}N) = (x \leq y \land y = w) \lor (\exists t_1, t_2, t_3{:}N.\, x > y \land \\ T(x-1, y, z, t_1) \land T(y-1, z, x, t_2) \land T(z-1, x, y, t_3) \land T(t_1, t_2, t_3, w))\end{aligned}$$

Finally, we consider the decision problem whether the Alternating Bit Protocol (ABP) is branching bisimilar to a one-place buffer, both with infinite data. We have two variants of this problem: one where these models are indeed equal, and one where we intentionally introduced an error in the implementation of the buffer. These PBESs are encoded using the techniques in [4], as implemented in the mCRL2 tool `lpsbisim2pbes`.

[2]The sources of mCRL2 are available via `https://github.com/mCRL2org/mCRL2`

[3]The experiments are available online via `https://doi.org/10.5281/zenodo.3528141`.

Table 1: Runtime comparison between several variants of PBES quotienting and `pbes-cvc4`. All runtimes are in seconds. 't.o.' indicates a time-out and a cross indicates that a PBES cannot be handled.

| | node of interest/ property | | PQ | | | PQ+sk | | | PQ+sk+ss | | | `pbes-cvc4` |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| model | | result | $\|V\|$ | iter. | time | $\|P\|$ | iter. | time | $\|P\|$ | iter. | time | time |
| ball game | winning impossible | *false* | 12 | 65 | 1.59 | 11 | 65 | 1.73 | 11 | 11 | 0.19 | 0.27 |
| | infinitely often *put_ball* | *true* | 2 | 4 | 0.01 | 1 | 2 | <0.01 | 1 | 2 | <0.01 | t.o. |
| train gate | *go(1)* at time 20 | *true* | 7 | 10 | 1.37 | 6 | 8 | 0.58 | 6 | 6 | 0.46 | 0.39 |
| | fairness | *false* | 15 | 57 | 19.58 | 4 | 31 | 9.28 | 4 | 31 | 11.59 | ✗ |
| Fischer (N=3) | no deadlock | *true* | 5 | 5 | 0.24 | 3 | 4 | 0.22 | 3 | 4 | 0.24 | ✗ |
| Fischer (N=4) | request must serve | *false* | 18 | 50 | 431.43 | 3 | 16 | 9.04 | 3 | 3 | 0.91 | ✗ |
| bakery | no deadlock | *true* | 1 | 1 | <0.01 | 1 | 1 | <0.01 | 1 | 1 | <0.01 | t.o. |
| | request must serve | *false* | 64 | 153 | 6.79 | 5 | 17 | 0.36 | 5 | 17 | 0.37 | 0.44 |
| Hesselink | cache consistency | *false* | | t.o. | | 20 | 754 | 345.89 | 20 | 753 | 348.43 | ✗ |
| | all writes finish | *false* | | t.o. | | 24 | 219 | 11.07 | 24 | 259 | 11.41 | ✗ |
| CABP | receive infinitely often | *true* | 79 | 313 | 14.12 | 16 | 114 | 2.69 | 17 | 111 | 3.13 | ✗ |
| trading | infinite run possible | *true* | 10 | 16 | 0.1 | 6 | 9 | 0.03 | 6 | 9 | 0.04 | t.o. |
| McCarthy | $M(0,10)$ | *true* | | t.o. | | 14 | 726 | 45.38 | 14 | 725 | 55.67 | ✗ |
| | $M(0,9)$ | *false* | | t.o. | | 299 | 1025 | 85.67 | 299 | 1025 | 90.04 | ✗ |
| Takeuchi | $T(3,2,1,3)$ | *true* | | t.o. | | 12 | 54 | 6.48 | 12 | 54 | 6.62 | ✗ |
| | $T(3,2,1,2)$ | *false* | | t.o. | | 186 | 79 | 16.32 | 185 | 79 | 17.92 | ✗ |
| ABP + buffer | branching bisimilar | *true* | 30 | 55 | 0.21 | 29 | 49 | 0.24 | 23 | 42 | 0.24 | ✗ |

## *9.1. Comparison of PBES solvers*

We ran `pbessymbolicbisim` and `pbes-cvc4` for each of the PBESs. The results are listed in Table 1. Here, 'PQ' denotes the standard PBES quotienting procedure and '+sk' and '+ss' denote the additional use of the stable kernel and stability under solution, respectively. For each PBES, we report the solution for the node of interest and the runtime in seconds for each approach. The runtimes reported here do not include the time required to compile the rewriter, which is roughly constant for each PBES. For each PBES quotienting experiment, we also report the size of the resulting reduced dependency space or proof graph, denoted with $|V|$ and $|P|$ respectively, and the number of iterations required to compute it, denoted with 'iter.'. A timeout, set to half an hour, is represented with 't.o.' and we write a cross for the PBESs that cannot be handled.

We observe that the use of stable kernels improves the performance over the basic PBES quotienting procedure for nearly every PBES in our set of benchmarks. Furthermore, several timeouts occur for 'PQ', while 'PQ+sk' and 'PQ+sk+ss' manage to solve these PBESs. The added value of stability under solution over the stable kernels procedure is not clear. The only instances where it performs significantly better are the ball game with the 'winning impossible' property and Fischer's protocol with the 'request must serve' property. For most other models, stability under solution causes some overhead, leading to a longer runtime.

The runtime of `pbes-cvc4` is very small for the three cases it can solve. However, it fails to provide a solution in most cases. The three cases where a timeout occurs for `pbes-cvc4` (trading, ball game and bakery) are similar: the models contain one or more variables that strictly increase. Since `pbes-cvc4` can only find lasso-shaped proof graphs, it does not terminate for PBESs with infinite proof graphs that are not lasso-shaped.

For Fischer and bakery with the no deadlock property and the equivalence problem on ABP and buffer, the reduced proof graph covers almost the entire reduced dependency space. Only the block containing $X_{false}$ (cf. proof of Proposition 2) is not present in the proof graph. In those cases, PBES quotienting does not benefit from the optimisation of using stable kernels (Procedure 3).

In order to obtain a result for Takeuchi with the node of interest $T(3,2,1,2)$, we modified our implementation slightly: after every iteration, we randomly shuffle the order in which blocks are stored. This can affect the *splitting strategy*: the choice of blocks $b$ and $b'$ to be used for splitting. This is discussed in more detail in Section 9.3.

Table 2: Runtime comparison between minimal model generation and PBES quotienting. All runtimes are in seconds. 't.o.' indicates a time-out and a cross indicates that a property cannot be handled.

| | | MMG | | | PQ+sk+ss | | |
|---|---|---|---|---|---|---|---|
| model | property | $\|P\|$ | iter. | time | $\|P\|$ | iter. | time |
| ball game | winning impossible | 11 | 64 | 0.50 | 11 | 11 | 0.19 |
| | infinitely often *put_ball* | | | | 1 | 2 | <0.01 |
| train gate | *go(1)* at time 20 | | ✗ | | 6 | 6 | 0.46 |
| | fairness | 15 | 72 | 15.64 | 4 | 31 | 11.59 |
| Fischer (N=3) | no deadlock | 62 | 180 | 22.36 | 3 | 4 | 0.24 |
| Fischer (N=4) | request must serve | | t.o. | | 3 | 3 | 0.91 |
| bakery | no deadlock | 22 | 52 | 1.31 | 1 | 1 | <0.01 |
| | request must serve | | | | 5 | 17 | 0.37 |
| ABP | branching bisimilar | 28 | 111 | 5.12 | 23 | 42 | 0.42 |
| buffer | | 3 | 2 | <0.01 | | | |

## *9.2. Comparison of PBES quotienting and MMG*

We also conducted several experiments with our implementation of minimal model generation in the tool `lpssymbolicbisim`. To obtain meaningful results, we made the following changes to our models:

- For the timed models (ball game, train gate and Fischer), we removed the time tags from actions and encoded the timed semantics of mCRL2 using a new process parameter. As a consequence, we cannot verify properties that refer to absolute time. This transformation is implemented in the tool `lpsuntime`.
- We removed the infinite data domain from the Hesselink, CABP, ABP and buffer models.

Without these modifications, MMG cannot compute a bisimulation quotient.

The results are listed in Table 2. For MMG, $|P|$ indicates the size of the resulting reduced LTS. For ball game and bakery, we only have to run `lpssymbolicbisim` once (cf. Figure 8). On the other hand, to check branching bisimilarity of ABP and the buffer, MMG has to generate the bisimulation quotient for both models before we can compare them with the tool `ltscompare`. Minimal model generation cannot be used to model check two of our instances. First, for the train gate model, MMG can compute a bisimulation quotient, but it is not possible to subsequently construct a PBES that accurately encodes the property "*go(1)* at time 20", since the quotient does not contain references to absolute time. Second, MMG times out for the Fischer model with four processes.

For the train gate model with the fairness property, our PBES quotienting performs similarly to MMG. This is partially due to the fact that the state space is partly encoded twice in the PBES, once for each fixpoint in the formula, similar to the bakery PBES of Section 4. For most of the other benchmarks, PBES quotienting outperforms MMG.

## *9.3. Splitting Strategy*

While performing these experiments, we noticed quite some variability in the results for certain models, especially McCarthy and Takeuchi. Slight alterations in the formulation of the PBES can have a significant effect on the runtime. Closer inspection revealed that the cause is the choice of blocks used to split. In our current implementation, we apply the simple heuristic of giving preference to blocks close to the block containing the node of interest. The following example shows the importance of the splitting strategy.

**Example 11.** We consider the node of interest $(X, (\mathit{true}, 0))$ in the PBES below.

$$\begin{aligned}\nu X(b{:}B, n{:}Z) &= ((b \vee n > 0) \wedge X(b, n-1)) \vee (b \wedge Y)\\ \mu Y &= Y\end{aligned}$$

Here, $Z$ represents the integers. Since splits happen closest to the block containing the node of interest, the block containing $(X, (\mathit{true}, 0))$ is continuously split with respect to itself, and we obtain the following partition after $i$ iterations (while not performing reachability analysis):

$$\{\{(X, (b, n)) \mid b \vee n \geq i\}, \{(X, (\mathit{false}, n)) \mid n \leq 0\}, \{Y\}\} \cup \bigcup_{0<n<i} \{\{(X, (\mathit{false}, n))\}\}$$

Splitting $\{(X, (b, n)) \mid b \vee n \geq i\}$ with respect to $\{Y\}$ results in $\{(X, (\mathit{true}, n)\}$ and $\{(X, (\mathit{false}, n)) \mid n \geq i\}$. This leads to immediate termination, since the former block – which contains the node of interest – is a stable kernel. In this example, the choice of splitting strategy determines whether PBES quotienting terminates. □

To find a good algorithm or heuristic for this block selection, one can draw inspiration from works on minimal model generation, *e.g.*, [29]. Performing static analysis to obtain invariants (*e.g.* $b$ is invariably *true* for the node of interest) can be another way to identify which blocks to split. Furthermore, the splitting strategy can be made more robust by introducing some randomness: this is to prevent certain blocks from being ignored indefinitely.

## 10. Related Work

The first works on generating minimal representations from behavioural specifications were written by Bouajjani *et al.* [12]. Later, these ideas were applied to timed automata [30, 31]. Similar to our approach, they rely on bisimulation to compute the minimal quotient directly from a specification. Fisler and Vardi [14] extended this work to include early termination when performing reachability analysis. Our work is similar in spirit to these methods, but it generalises these by allowing to verify properties expressed in the full modal mu-calculus and by supporting infinite-state systems, not limited to real-time systems.

The techniques and theory we present also generalise several other closely related works, such as [23, 8, 9, 20]. Nagae *et al.* [23] transfer the ideas of Bouajjani *et al.* to disjunctive, quantifier-free PBESs and generate finite parity games that can be solved. They later expanded the work to existential PBESs [8]. These fragments of the PBES logic limit the type of properties one can verify. A small set of experimental results shows that their approach is feasible in practice for small academic examples.

Koolen *et al.* [9] use an SMT solver to search for linear proof graphs in disjunctive or conjunctive PBESs. Their technique manages to find solutions for model checking problems where traditional tools time out. They conclude that even for problems where enumeration of the state space is possible, an instantiation-based approach is not always faster. We remark that the number of unrollings performed by their tool gives a rough indication of the optimal size of the proof graph constructed with our techniques when applied to disjunctive or conjunctive PBESs.

In [20], Keiren *et al.* define two equivalence relations based on bisimulation for BESs. These relations are then used to minimise BESs that represent model checking problems. Experiments show that applying minimisation speeds up the solving procedure, *i.e.*, the time required for minimising and then solving the minimal BES is lower than the time required to solve the original BES. Whereas [20] applies explicit-state techniques by working directly on a BES, our work is based on a symbolic representation. The disadvantage of the explicit approach of [20] is that it requires one to instantiate a PBES to BES first, which can be time consuming. Furthermore, the instantiation does not terminate for infinite-state systems.

Fontana *et al.* [10] construct symbolic proof trees to check alternation-free mu-calculus formulae on timed automata. To recursively prove (sub)formulas, they unfold the transition relation according to a set of proof rules they propose. This approach allows a larger class of properties than UPPAAL [32], which only supports a subset of TCTL. Contrary to our approach, the proof they produce is not necessarily minimal with respect to bisimulation.

The authors of [31] also identified the problem that the characteristic functions should be as compact as possible in order to improve the scalability (cf. Section 9). They develop a specialised partition-refinement technique for the setting of timed automata such that the characteristic functions are always conjunctive, *i.e.*, they represent a convex set of nodes. Preserving convexity comes at a cost, however: the resulting stable partition can be finer that the bisimulation quotient.

Although our work was not inspired by counterexample-guided abstraction refinement (CEGAR) [33], we see many similarities. In this approach, an abstraction of the model under consideration is continuously refined based on spurious traces that are found by a model checker. Our procedure that finds stable kernels essentially refines with respect to 'spurious proof graphs'. Compared to our approach, CEGAR typically supports a less expressive class of properties, such as ACTL or LTL.

## 11. Conclusion

We presented an approach to solving arbitrarily-structured PBESs with infinite data, which enables solving of a larger set of PBESs than possible with existing tools. This improves the state-of-the-art for model checking and equivalence checking on (concurrent) systems with infinite data.

A drawback of performing quotienting on the level of PBESs is that this process has to be repeated for each property that needs to be checked (cf. Figure 8). On the other hand, for minimal model generation, the LTS needs to be generated only once, after which multiple properties can be checked by constructing multiple BESs, which is a relatively cheap operation. However, as we have shown, PBES quotienting also has several fundamental advantages, which improve its applicability in a practical setting.

Further study is required to fully understand the effects of splitting strategy (cf. Section 9.3). We believe that a good strategy, perhaps based on heuristics, can significantly improve the scalability of our approach.

When checking fairness properties, the state space is typically encoded twice in the corresponding PBES. The PBES from Section 4.2 is a perfect example. In the current implementation, the same work is sometimes done twice for different predicate variables. The procedures can be further optimised by exploiting this symmetry.

Another possible direction is to further weaken the equivalence relation on dependency graph nodes. Here, one can draw inspiration from equivalence relations defined on parity games, for instance as defined in [34].

## References

[1] J. F. Groote, T. A. C. Willemse, Parameterised boolean equation systems, Theoretical Computer Science 343 (2005) 332–369.

[2] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes, STTT 15 (2013) 89–107.

[3] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, J. W. Wesselink, A. W. Wijs, T. A. C. Willemse, The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability, in: TACAS 2019, volume 11428 of *LNCS*, pp. 21–39.

[4] T. Chen, B. Ploeger, J. van de Pol, T. A. C. Willemse, Equivalence Checking for Infinite Systems using Parameterized Boolean Equation Systems, in: CONCUR 2007, volume 4703 of *LNCS*, pp. 120–135.

[5] S. Orzan, W. Wesselink, T. A. C. Willemse, Static analysis techniques for parameterised boolean equation systems, in: TACAS, volume 5505 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 230–245.

[6] J. J. A. Keiren, J. W. Wesselink, T. A. C. Willemse, Liveness Analysis for Parameterised Boolean Equation Systems, in: ATVA 2014, volume 8837 of *LNCS*, pp. 219–234.

[7] B. Ploeger, J. W. Wesselink, T. A. C. Willemse, Verification of reactive systems via instantiation of Parameterised Boolean Equation Systems, Information and Computation 209 (2011) 637–663.

[8] Y. Nagae, M. Sakai, Reduced Dependency Spaces for Existential Parameterised Boolean Equation Systems, in: WPTE 2017, volume 265 of *EPTCS*, pp. 67–81.

[9] R. P. J. Koolen, T. A. C. Willemse, H. Zantema, Using SMT for Solving Fragments of Parameterised Boolean Equation Systems, in: ATVA 2015, volume 9364 of *LNCS*, pp. 14–30.

[10] P. Fontana, R. Cleaveland, The Power of Proofs: New Algorithms for Timed Automata Model Checking, in: FORMATS 2014, volume 8711 of *LNCS*, pp. 115–129.

[11] T. Neele, T. A. C. Willemse, J. F. Groote, Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting, in: FACS 2018, volume 11222 of *LNCS*, pp. 216–236.

[12] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, C. Ratel, Minimal state graph generation, Science of Computer Programming 18 (1992) 247–269.

[13] D. Kozen, Results on the propositional $\mu$-calculus, Theoretical Computer Science 27 (1982) 333–354.
[14] K. Fisler, M. Y. Vardi, Bisimulation and model checking, in: CHARME 1999, volume 1703 of *LNCS*, pp. 338–342.
[15] D. Park, Concurrency and automata on infinite sequences, in: Theoretical Computer Science, volume 104 of *LNCS*, pp. 167–183.
[16] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Communications of the ACM 17 (1974) 453–455.
[17] S. Cranen, B. Luttik, T. A. C. Willemse, Proof graphs for parameterised Boolean equation systems, in: CONCUR 2013, volume 8052 of *LNCS*, pp. 470–484.
[18] J. W. Wesselink, T. A. C. Willemse, Evidence Extraction from Parameterised Boolean Equation Systems, in: ARQNL 2018, volume 2095 of *CEUR Workshop Proceedings*, pp. 86–100.
[19] J. F. Groote, T. A. C. Willemse, Model-checking processes with data, Science of Computer Programming 56 (2005) 251–273.
[20] J. J. A. Keiren, T. A. C. Willemse, Bisimulation minimisations for Boolean equation systems, in: HVC 2009, volume 6405 of *LNCS*, pp. 102–116.
[21] G. Kant, J. van de Pol, Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games, in: GRAPHITE 2012, volume 99 of *EPTCS*, pp. 50–65.
[22] A. Mader, Modal $\mu$-calculus, model checking and Gauß elimination, in: TACAS 1995, volume 1019 of *LNCS*, pp. 72–88.
[23] Y. Nagae, M. Sakai, H. Seki, An Extension of Proof Graphs for Disjunctive Parameterised Boolean Equation Systems, in: WPTE 2016, volume 235 of *EPTCS*, pp. 46–61.
[24] T. A. C. Willemse, Consistent Correlations for Parameterised Boolean Equation Systems with Applications in Correctness Proofs for Manipulations, in: CONCUR 2010, volume 6269 of *LNCS*, pp. 584–598.
[25] W. Zielonka, Infinite games on finitely coloured graphs with applications to automata on infinite trees, Theoretical Computer Science 200 (1998) 135–183.
[26] S. Cranen, M. Gazda, J. W. Wesselink, T. A. C. Willemse, Abstraction in Fixpoint Logic, TOCL 16 (2015) Article 29.
[27] W. H. Hesselink, Invariants for the construction of a handshake register, Inf. Process. Lett. 68 (1998) 173–177.
[28] D. E. Knuth, Textbook Examples of Recursion, Artificial and Mathematical Theory of Computation 91 (1991) 207–229.
[29] D. Lee, M. Yannakakis, Online minimization of transition systems (extended abstract), in: STOC '92, pp. 264–274.
[30] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, H. Wong-Toi, Minimization of Timed Transition Systems, in: CONCUR 1992, volume 630 of *LNCS*, pp. 340–354.
[31] S. Tripakis, S. Yovine, Analysis of Timed Systems Using Time-Abstracting Bisimulations, FMSD 18 (2001) 25–68.
[32] G. Behrmann, A. David, K. G. Larsen, A Tutorial on UPPAAL, in: SFM-RT 2004, volume 3185 of *LNCS*, pp. 200–236.
[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement, in: CAV 2000, volume 1855 of *LNCS*, pp. 154–169.
[34] S. Cranen, J. J. A. Keiren, T. A. C. Willemse, A Cure for Stuttering Parity Games, in: ICTAC 2012, volume 7521 of *LNCS*, pp. 198–212.