Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting

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

Eindhoven University of Technology, The Netherlands
{T.S.Neele,T.A.C.Willemse,J.F.Groote}@tue.nl

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 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. Unlike existing approaches, our technique is not restricted to subfragments of PBESs. 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.

1 Introduction

A *parameterised Boolean equation system* (PBES) [12] 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 [11] and mCRL2 [6], 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 it—if needed—using static analysis techniques, instantiating it to a finite *Boolean equation system* (BES) and subsequently solving this BES. 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 [21,18,10], 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. In this paper, we present a more general approach that is applicable to the *full* class of PBESs, without such limitations. Our contributions are as follows:

- We introduce a new normal form for PBESs which we call *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*.
- We provide an algorithm that computes, using quotienting, a minimal reduced dependency graph from a symbolic representation of the dependency graph of a PBES. Upon termination of the algorithm, the computed artefact can then be used to solve the PBES. The correctness is given by Theorem 3.
- On top of this, we provide an algorithm that extracts finite partial solutions from PBESs that have an infinite minimal reduced dependency graph. The correctness of this approach is given by Theorem 4.

To validate the above, we perform a number of experiments with an implementation of our two algorithms and compare these to the approach of [18]. 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 on systems with infinite data domains and also equivalence checking of systems with infinite data domains.

The rest of the paper is structured as follows: Section 2 introduces the basic theoretical concepts. Section 3 contains an example that shows how PBESs can be applied and what the shortcomings of current solving techniques are. Then, Sections 4 and 5 show how a minimal representation of the semantics of a PBES can be computed. An improved algorithm is presented in Section 6, and the performance of an experimental implementation is evaluated in Section 7. Finally, Section 8 gives an overview of related work and Section 9 presents a conclusion and suggestions for future work. For detailed proofs of our lemmas and theorems we refer to a technical report [23].

2 Preliminaries

In this paper, we work with abstract data types and denote their non-empty data sorts with the letters $D, E, \ldots$ and their corresponding semantic domains by $\mathbb{D}, \mathbb{E}, \ldots$ In addition, we use B to denote the Booleans and N to denote the natural numbers $\{0, 1, 2, \ldots\}$, 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 $\mathcal{V}$. Expressions not containing variables are called *ground terms*. For expressions that do contain

variables, we have a data environment δ 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 δ is denoted $[\![f]\!]\delta$. The set of all data environments is Δ. Updates to an environment δ 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 $d' \neq d$.

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 [12] for a more in-depth treatment and additional examples.

Definition 1. *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 a data term 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 ϕ 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 δ is denoted by $[\![\phi]\!]\eta\delta$ and inductively defined as follows:*

$$[\![b]\!]\eta\delta \;=\; [\![b]\!]\delta \qquad\qquad [\![X(f)]\!]\eta\delta \;=\; \begin{cases} \textit{true if } [\![f]\!]\delta \in \eta(X) \\ \textit{false otherwise} \end{cases}$$

$$[\![\varphi \wedge \psi]\!]\eta\delta \;\Leftrightarrow\; [\![\varphi]\!]\eta\delta \textit{ and } [\![\psi]\!]\eta\delta \textit{ hold} \qquad [\![\varphi \vee \psi]\!]\eta\delta \;\Leftrightarrow\; [\![\varphi]\!]\eta\delta \textit{ or } [\![\psi]\!]\eta\delta \textit{ hold}$$

$$[\![\varphi \Rightarrow \psi]\!]\eta\delta \;\Leftrightarrow\; [\![\varphi]\!]\eta\delta \textit{ holds implies that } [\![\psi]\!]\eta\delta \textit{ holds}$$

$$[\![\forall d{:}\, E.\, \varphi]\!]\eta\delta \;\Leftrightarrow\; \textit{for all } v \in \mathbb{E},\; [\![\varphi]\!]\eta\delta[v/d] \textit{ holds}$$

$$[\![\exists d{:}\, E.\, \varphi]\!]\eta\delta \;\Leftrightarrow\; \textit{for some } v \in \mathbb{E},\; [\![\varphi]\!]\eta\delta[v/d] \textit{ holds}$$

A predicate formula is *syntactically monotone* iff all its subformulae of the form $\varphi \Rightarrow \psi$ are such that φ contains no predicate variables. Without loss of generality, in the theory we develop in this paper we only consider parameterised Boolean equation systems where each equation carries the same single parameter of a given data sort D. In our examples, we use (multi-parameter) equations ranging over the Booleans (B) and the natural numbers (N).

Definition 2. *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, μ and ν 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 φ 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 by $\mathcal{E}$, *i.e.*, those variables occurring at the left-hand side of an equation. For an equation for X,

d_X denotes its parameter and φ_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 φ_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 3. *The* solution $[\![\mathcal{E}]\!]\eta\delta$ *of a PBES* $\mathcal{E}$ *in the context of a predicate environment* η *and a data environment* δ*, 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 $T_X \colon 2^{\mathbb{D}} \to 2^{\mathbb{D}}$, which follows from syntactic monotonicity of φ_X, guarantees the existence of the least fixpoint μT_X and greatest fixpoint νT_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 η and δ. For this reason, we often omit η and δ 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 $v \in [\![\mathcal{E}]\!](X)$.

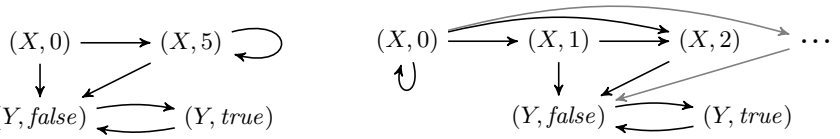
Example 1. Consider the following PBES consisting of an equation for X and an equation for Y, both carrying a single parameter. Furthermore, the equation for X has a least fixpoint, and the equation for Y has a greatest fixpoint.

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

The solution η for this PBES satisfies $\eta(X) = \emptyset$ and $\eta(Y) = \mathbb{B}$. □

The theory of this paper is built on the notion of *dependency graphs* and *proof graphs* explored in [8]. 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



(a) A positive dependency graph

(b) A negative dependency graph

Fig. 1: Dependency graphs for the PBES from Example 1.

$\mathsf{env}(S, \mathit{false})(X) = \mathbb{D} \setminus \mathsf{env}(S, \mathit{true})(X)$. Furthermore, every predicate variable bound in $\mathcal{E} = (\sigma_1 X_1(d{:}D) = \varphi_1) \dots (\sigma_n X_n(d{:}D) = \varphi_n)$ is assigned a *rank*, where $\mathsf{rank}_{\mathcal{E}}(X_i)$ is the number of alternations in the sequence of fixpoint symbols $\nu\sigma_1\sigma_2 \dots \sigma_i$. Observe that $\mathsf{rank}_{\mathcal{E}}(X_i)$ is *even* iff $\sigma_i = \nu$.

Definition 4. *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 δ, $[\![\varphi_X]\!]\eta(\delta[v/d_X]) = r$ with $\eta = \mathsf{env}((X, v)^\bullet, r)$, where $s^\bullet$ denotes the successor set of a node, defined as $s^\bullet = \{t \mid s\, E\, t\}$.*

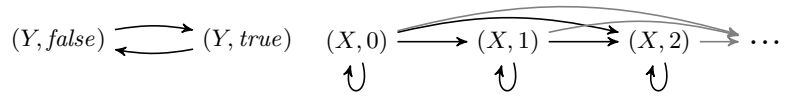
Intuitively, in a *positive dependency graph* (where r is *true*), $\eta = \mathsf{env}((X, v)^\bullet, r)$ is a predicate environment that maps all successors of (X, v) to *true* and all other nodes to *false*. Then, the requirement is that φ_X (and thus $X(v)$) is *true* under η and a data environment 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* (where r is *false*) indicates a node (X, v) is *false*, because its successors are all *false*.

Example 2. Recall the PBES from Example 1. Figure 1 depicts a positive and a negative dependency graph for this PBES. We focus on node $(X, 0)$ in the positive dependency graph of Figure 1(a). Its successors are $(X, 5)$ and (Y, false). The environment η induced by these successors is given by $\mathsf{env}((X, 0)^\bullet, \mathit{true})$, which sets these successors to *true*; *i.e.* η 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 η 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 $(X, 0)$.

Note that nodes (Y, false) and (Y, true) are dependent on each other in both dependency graphs. Furthermore, in the negative case, $(X, 0)$ needs no dependency on (Y, false) as long as it depends on all (X, i) with $i \in \mathbb{N}$. □

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 5. *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 π in G, the number* $\min\{\mathsf{rank}_{\mathcal{E}}(X) \mid X \in V^\infty(\pi)\}$



(a) A positive proof graph (b) A negative proof graph

Fig. 2: Proof graphs for the PBES from Example 1.

is even (respectively odd), where $V^\infty(\pi)$ *is the set of predicate variables that occur infinitely often along* π*.*

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 3).

Example 3. Recall again the PBES from Example 1. In this PBES, the rank of X is 1, and the rank of Y is 2. Figure 2 depicts a positive and a negative proof graph for this PBES. Note that Figure 2(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, false) or (Y, true). Similarly, the proof graph in Figure 2(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 $(X, 0)$. □

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

Theorem 1 ([8]). *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 [8], proof graphs were introduced mainly to formalise the concept of witnesses and counterexamples. 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 illustrate how to apply PBESs in model checking with an example.

3 Motivating Example

To show how PBESs can be used for model checking and to motivate our approach, we introduce a slightly larger example in this section. The model we consider is a simplified version of Lamport's bakery protocol [19]. 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.

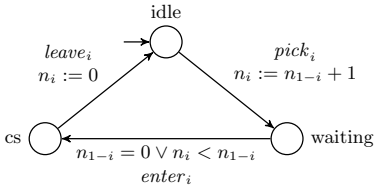


Fig. 3: Process i from the simplified bakery protocol.

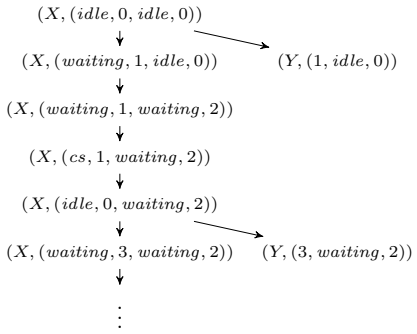


Fig. 4: Part of the infinite proof graph of the bakery example.

On this model, we would like to check the property "whenever customer 0 picks a number, it will unavoidably enter the critical section within a finite amount of time". This can be formalised with the modal mu-calculus formula $\nu X.([true]X \wedge [pick_0]\mu Y.([\overline{enter_0}]Y \wedge \langle true\rangle true))$. From the model and the formula, the following PBES can be constructed automatically:

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

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 S. Predicate variable X represents the fact that the property has to hold at any point in time. Therefore, it encodes the full behaviour of the system (lines 3 to 8) and is labelled with a greatest fixpoint. 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 a least fixpoint. The property holds, since the solution for the initial state is *true*, *i.e.*, $(X, (idle, 0, idle, 0)) \in [\![\mathcal{E}]\!](X)$.

There are a few interesting observations that we can make based on this PBES. Firstly, it is not possible to solve it with traditional instantiation-based techniques, since the dependency graph is infinite. Moreover, there is no finite proof graph that contains $(X, (idle, 0, idle, 0))$, so even the application of smart heuristics to guide the instantiation does not improve the situation. See Figure 4 for a part of the infinite proof graph. Secondly, 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.

4 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 [15]. 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) [16]. 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 6. *Let $\mathcal{E}$ be a PBES. Then $\mathcal{E}$ is in* standard recursive form *(SRF) iff for all $(\sigma_i X_i(d{:}D) = \phi) \in \mathcal{E}$, ϕ 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 δ 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 [14]. 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 1. *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 [12]. 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 $\mathcal{E}$.

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. □

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 1 and subsequently combining clauses that have the same predicate variable, relying on suitable pairing and projection operators for the data arguments.

Henceforward we only consider PBESs in CRF. 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 7. *Let $\mathcal{E}$ be a PBES in CRF, where each equation has the same structure as in Definition 6. 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 δ and $v_j{\in}\mathbb{E}_j$, both $w = [\![g_j(d, e_j)]\!]\delta[v_j/e_j, v/d]$ and $[\![f_j(d, e_j)]\!]\delta[v_j/e_j, v/d]$ hold.*

Definition 7 generalises the definition of a dependency space from [18]. 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 1. *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 4) is *true*. From the definition of $\mathsf{env}(S, true)$ and Definition 7 we know the following:

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

Using the definition of the semantics and (†), we can deduce that $[\![\varphi_{X_i}]\!]\eta\delta[v/d] = r$, where $\eta = \mathsf{env}((X_i, v)^\bullet, true)$. With this, the condition on transitions in a positive

dependency graph is satisfied. The proofs for the other three combinations are analogous. □

Theorem 2. *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 $E \not\subseteq E'$, *i.e.*, G is not strictly smaller than 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. □

5 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 or with a parity game solver. 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.* [18] use SMT solvers to find proof graphs and Nagae *et al.* [22,21] 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 efficiently.

Definition 8. *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 finite partition of V,* i.e. *$\bigcup V' = V$ and for all distinct $b, b' \in V'$ we have $b \cap b' \neq \emptyset$,*
- *$E' = \{(b, b') \in V' \times V' \mid \exists s \in b, t \in b' . s \, E \, t\}$.*

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. As equivalence relation on nodes we use *bisimulation* [24].

Definition 9. *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 $(X, v)\mathcal{R}(Y, w)$:*

- $\mathit{rank}_{\mathcal{E}}(X) = \mathit{rank}_{\mathcal{E}}(Y)$ *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 $(X', v')\mathcal{R}(Y', w')$.*

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

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 8), the *bisimulation quotient* of G, notation ${}^{G}\!/_{\leftrightarroweq}$.

Partition refinement

To compute the bisimulation quotient, we rely on partition refinement. In this algorithm, a partition of the state space is iteratively refined until it becomes *stable* (a formal definition follows). The coarsest stable partition coincides with the equivalence classes under bisimulation.

In the context of partition refinement, a *block* is a set of nodes. A *partition* P of a set of nodes V is a set of blocks that are pairwise disjoint. Furthermore, the union over all blocks in P is equal to V. We say a partition P is *finer* than a partition P' iff all blocks of P are contained in some block of P'.

Algorithm 1 shows how to perform partition refinement on a dependency graph $G = (V, E)$ that underlies the PBES $\mathcal{E}$. The initial partition P_0 is set to $\{\{(X, v) \in V \mid v \in \mathbb{D} \wedge \mathsf{rank}_{\mathcal{E}}(X) = \mathsf{rank}_{\mathcal{E}}(Y) \wedge \mathsf{op}_{\mathcal{E}}(X) = \mathsf{op}_{\mathcal{E}}(Y)\} \mid Y \in \mathsf{bnd}(\mathcal{E})\}$. In every iteration, we find two blocks $b, b' \in P$ and split b with respect to b' in the following way:

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

Algorithm 1: Partition refinement for PBESs

Input: PBES $\mathcal{E}$, initial partition P_0

1 $i \leftarrow 0$;
2 **while** P_i *is not stable* **do**
3 $\quad P_{i+1} \leftarrow (P_i \setminus \{b\}) \cup \{\mathit{split}(b, b'), \mathit{co\text{-}split}(b, b')\}$ **for some** $b, b' \in P_i$ such that $\mathit{split}(b,' b)$ and $\mathit{co\text{-}split}(b, b')$ are non-empty;
4 $\quad i \leftarrow i + 1$;
5 **return** P_i;

Then we update the partition to reflect this split (line 3). Note that each partition P_{i+1} is finer than partition P_i.

If a block b cannot be split with respect to a block b', we say b is *stable* with respect to b'. Block b is stable with respect to a set of blocks K if it is stable with respect to all the blocks in K. A partition P is stable (with respect to itself) iff all of the blocks in P are stable with respect to P. The partition refinement algorithm terminates when P is stable (line 2).

Since our goal is to enable reasoning about infinite dependency graphs, we cannot store blocks explicitly. Instead, we represent each block with a *characteristic function*.

Definition 10. *Let $\mathcal{E}$ be a PBES and b be a set of nodes in a dependency graph of $\mathcal{E}$. The corresponding* characteristic function $k_b : \mathsf{sig}(\mathcal{E}) \to \mathbb{B}$ *is defined as:*

$$k_b(X, v) = \begin{cases} \mathit{true} & \mathit{if}\ (X, v) \in b \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

With this representation, we can also provide a symbolic implementation of the split and co-split functions. In the following definitions, k and k' are Boolean expressions representing characteristic functions.

$$\mathit{split}(k, k') = \lambda X_i{\in}\mathcal{X}, d{:}D.\, k(X_i, d) \wedge \bigvee_{j \in J_i} \exists e_j{:}E_j.\, f_j(d, e_j) \wedge k'(X_j, g_j(d, e_j))$$

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

Example 4. We revisit the bakery protocol example from Section 3. Running Algorithm 1 on that PBES yields a finite reduced dependency space (depicted in Figure 5) 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 1). Note the symmetry between process 0 and process 1 in those states belonging to variable X and also the parallels between X and Y. □

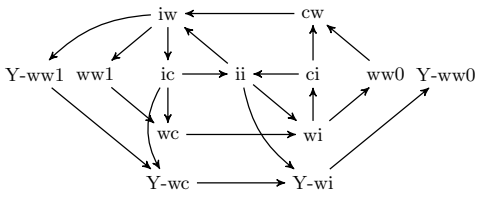


Fig. 5: Equivalence classes and transitions in the reduced dependency space of the bakery protocol example.

Algorithm 1 can be used to solve a PBES as follows. Upon its termination[1] 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 these solutions we can then derive the solution to the original PBES. We next formalise these steps.

Since we are reasoning in the context of partition refinement, we know that all partitions that are finer than P_0 are (by definition of P_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. The following definition shows how to construct a BES (in CRF) for a consistent reduced dependency graph.

Definition 11. *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:*

- $\mathsf{rank}_{\mathcal{E}_G}(X_b) = \mathsf{rank}_{\mathcal{E}}(X)$ *for all* $(X, v) \in b$,
- *If* $\mathsf{op}_{\mathcal{E}}(X) = \wedge$ *for all* $(X, v) \in b$ *then* $\phi_b = \bigwedge_{(b,b')\in E}(\mathit{true} \Rightarrow X_{b'}(\star))$,
- *If* $\mathsf{op}_{\mathcal{E}}(X) = \vee$ *for all* $(X, v) \in b$ *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' and their associated PBESs $\mathcal{E}$ and $\mathcal{E}'$, we say G and G' are *rank-operand-isomorph* when there is an isomorphism between them that preserves the rank and operand, which follow from $\mathcal{E}$ and $\mathcal{E}'$ respectively.

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

Lemma 2. *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.*

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

Proof. The proof is based on Theorem 1, Lemma 2, the reasoning that bisimulation reduction preserves bisimilarity and that bisimilarity is a *consistent correlation* [26], *i.e.*, bisimilarity preserves and reflects the solution of a PBES. □

We remark that the algorithm presented in this section generalises the algorithms presented by Nagae *et al.* in [22] and [21], which only apply to PBESs consisting of predicate formulae that contain no predicate variables within the scope of universal quantifiers.

[1] We remark that termination is not guaranteed as not every infinite dependency graph has a finite bisimulation quotient.

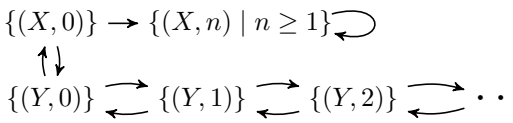
6 Computing Local Solutions

The approach presented in the previous section terminates when the bisimulation quotient is finite and all the operations on data are decidable. However, we are also interested in cases where 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 5. Consider the following PBES:

$$\nu X(n{:}N) = X(n+1) \vee (n = 0 \wedge Y(0))$$
$$\mu Y(n{:}N) = Y(n+1) \wedge (n = 0 \Rightarrow X(0)) \wedge (n > 1 \Rightarrow Y(n-1))$$

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



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. □

The example suggests 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. The following lemma and theorem formalise this.

Lemma 3. *Let $G = (V, E)$ be a dependency graph and $G_r = (V_r, E_r)$ a consistent reduced dependency graph of G. Furthermore, let $G'_r = (V'_r, E'_r)$ be a reduced dependency graph that is a subgraph of G_r and $G' = (V', E')$ its base graph. If $V'_r \subseteq V_r$ is stable with respect to itself then G'_r is bisimilar to G'.*

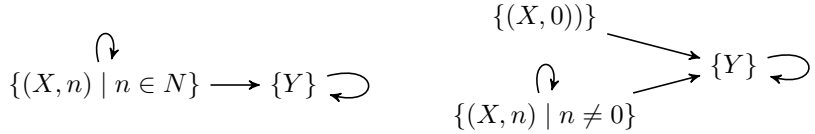
Proof. By proving that $\mathcal{R} = \{((X, v), b) \mid (X, v) \in b\}$ is a bisimulation relation, we show that G'_r is bisimilar to G'. □

Theorem 4. *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 reduced dependency graph that is a subgraph of G_r and $G' = (V', E')$ its base graph. If $V'_r \subseteq V_r$ is stable with respect to itself and 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 proof proceeds by showing that G' satisfies the conditions of a dependency graph and of a proof graph, by applying Lemma 3. □

The following example illustrates that the assumptions from Theorem 4 are necessary conditions.

Example 6. 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).



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. □

Based on this theorem, 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 are finished (by Theorem 4). See Algorithm 2. We maintain two sets of blocks: P contains the blocks in the proof graph that we are currently considering and U contains the other blocks. At line 4, we split a block in P and temporarily store the resulting partition in Q. Then, the set of blocks of the whole partition, reachable from the block containing the initial node (X, v) is computed (lines 5 and 6). Thereby, blocks that are not reachable from the initial node are effectively "thrown away", *i.e.*, they are not considered during next iterations. Since unreachable blocks cannot be part of a minimal proof graph for the initial node, this does not affect the correctness of the algorithm. From the reachable blocks, we extract a proof graph for the initial node (X, v) (line 7). Searching for a proof graph can be done with existing techniques, such as a solver for Boolean equations systems or for parity games. The blocks contained in the proof graph

Algorithm 2: Computing local partitions

Input: PBES $\mathcal{E}$, **initial partition** P_0, **initial node** (X, v)

1 $U_0 \leftarrow \emptyset$;
2 $i \leftarrow 0$;
3 **while** P_i *is not stable* **do**
4 $\quad Q \leftarrow (P_i \setminus \{k\}) \cup \{\mathit{split}(k, k'), \mathit{co\text{-}split}(k, k')\}$ **for some** $k, k' \in P_i$ such that $\mathit{split}(k, k')$ and $\mathit{co\text{-}split}(k, k')$ are non-empty;
5 $\quad Q \leftarrow Q \cup U_i$;
6 $\quad Q \leftarrow$ `computeReachableBlocks(`Q, (X, v)`)`;
7 $\quad (P_{i+1}, U_{i+1}) \leftarrow$ `findProofGraph(`Q, (X, v)`)`;
8 $\quad i \leftarrow i + 1$;
9 **return** P_i;

are again stored in P, the remaining blocks are stored in U. After every iteration, we check whether P is stable (line 3). If so, the algorithm terminates.

7 Implementation and Experiments

We implemented Algorithms 1 and 2 in a tool called `pbessymbolicbisim` that is part of the mCRL2 toolset [6]. The implementation calls the Z3 SMT-solver to determine whether one of the sets $\mathit{split}(k, k')$ or $\mathit{co\text{-}split}(k, k')$ is empty, *i.e.*, whether its characteristic function is unsatisfiable. Further simplification of the characteristic functions is handled by the mCRL2 term rewrite system. Choosing which block to split each iteration (line 3 of Algorithm 1 and line 4 of Algorithm 2) is done in such a way that an unreachable block is never split, in similar vein to [20].

We compare the performance of three approaches: our implementation of Algorithms 1 and 2 and the `pbes-cvc4` tool from [18]. We originally also aimed to compare with the tool `PBESSolver` from [21]. 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 8GB of memory running Ubuntu 16.04 and mCRL2 commit 9068139379.

Our set of benchmarks 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 [13]: cache consistency, and all writes finish).

The scheduling problem we consider is due to [22]; it encodes a fair trading problem encoded as a PBES. Furthermore, two recursive functions we consider are based on classical benchmarks for verification tools [17]. A modified version

Table 1: Runtime comparison between `pbessymbolicbisim` and `pbes-cvc4`. All runtimes are in seconds. 't.o.' indicates a time-out, 'o.o.m' indicates an out-of-memory error and a cross indicates that a PBES cannot be handled.

			Alg. 1		Alg. 2		`pbes-cvc4`
PBES	initial node/property	solution	$\|V\|$	time	$\|V\|$	time	time
ball game	winning impossible	*false*	12	0.55	12/12	0.65	0.27
	infinitely often *put_ball*	*true*	3	0.006	1/3	0.006	t.o.
train gate	*go(1)* at time 20	*true*	29	11.51	6/28	5.15	0.39
	fairness	*false*	19	21.95	5/32	4.79	✗
Fischer (N=3)	no deadlock	*true*	65	74.77	64/65	61.41	✗
Fischer (N=4)	request must serve	*false*		o.o.m.	5/38	20.87	✗
bakery	no deadlock	*true*	23	3.08	23/23	2.26	t.o.
	request must serve	*false*	123	85.02	14/111	14.52	0.44
Hesselink	cache consistency	*false*		o.o.m.	21/1807	387.54	✗
	all writes finish	*false*		t.o.	13/724	117.47	✗
CABP	receive infinitely often	*true*	260	267.95	25/702	61.02	✗
trading	$X_a(1,1)$	*true*	7	0.02	5/7	0.02	t.o.
McCarthy	$M(0,10)$	*true*	1633	1299.17	14/405	59.46	✗
	$M(0,9)$	*false*	1633	1364.33	128/178	8.57	✗
Takeuchi	$T(3,2,1,3)$	*true*		t.o.	9/187	35.64	✗
	$T(3,2,1,2)$	*false*		t.o.	77/198	39.42	✗
ABP+buffer	branching bisimilar	*true*	132	4.89	131/132	4.86	✗

of the McCarthy 91 function, as per [21], is represented with the following PBES:

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

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 [17]:

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

Finally, we consider the decision problem whether Alternating Bit Protocol (ABP) is branching bisimilar to a one-place buffer, both with infinite data. This PBES is encoded using the techniques in [4], as implemented in the mCRL2 tool `lpsbisim2pbes`.

The results are listed in Table 1. For each PBES, we report the solution for the initial node and the runtime in seconds for each approach. In addition, for Algorithm 1, we report the number of blocks in the reachable part of the bisimulation quotient as $|V|$. For Algorithm 2, we list the size of the reduced proof graph and the total number of blocks in memory at the moment the algorithm terminates. A timeout is represented with 't.o.', and an out-of-memory error with 'o.o.m.'. Furthermore, we write a cross for the PBESs that cannot be handled.

We observe that Algorithm 2 performs better than Algorithm 1 for nearly every PBES in our set of benchmarks. Algorithm 1 also runs into several out-of-memory errors, while Algorithm 2 manages to find a proof graph for every PBES.

The runtime of `pbes-cvc4` is very small for the four 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 1) is not present in the proof graph. In those cases, Algorithm 2 does not perform better than Algorithm 1.

8 Related Work

The first works on generating minimal representations from behavioural specifications were written by Bouajjani *et al.* [3]. Later, these ideas were applied to timed automata [1,25]. Similar to our approach, they rely on bisimulation to compute the minimal quotient directly from a specification. Fisler and Vardi [9] 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 [22,21,18,16]. Nagae *et al.* [22] 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 [21]. 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.* [18] 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. Even if 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 [16], 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 [16] applies explicit-state techniques by working directly on a BES, our work is based on a symbolic representation. The disadvantages of the explicit approach of [16] is that it requires one to instantiate a PBES to BES first. Therefore, it is not suitable 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 [2], which only supports a subset of TCTL. Contrary to our approach, the proof they produce is not necessarily minimal with respect to bisimulation.

Although our work was not inspired by counterexample-guided abstraction refinement (CEGAR) [5], 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 second algorithm 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.

9 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 possible direction for future work is to 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 [7]. We also want to investigate heuristics for the choice of blocks that are used for splitting in every iteration. The heuristics can for example be based on information obtained from static analysis of the PBES. We believe the choice of blocks during splitting can have a significant influence on the runtime.

Acknowledgements We would like to thank the anonymous reviewers for their constructive feedback. Their suggestions helped us to improve the paper before publication.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of Timed Transition Systems. In *CONCUR 1992*, volume 630 of *LNCS*, pages 340–354, 1992.
2. G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *SFM-RT 2004*, volume 3185 of *LNCS*, pages 200–236, 2004.
3. A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, 1992.
4. T. Chen, B. Ploeger, J. van de Pol, and T. A. C. Willemse. Equivalence Checking for Infinite Systems using Parameterized Boolean Equation Systems. In *CONCUR 2007*, volume 4703 of *LNCS*, pages 120–135, 2007.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV 2000*, volume 1855 of *LNCS*, pages 154–169, 2000.

6. S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In *TACAS 2013*, volume 7795 of *LNCS*, pages 199–213, 2013.
7. S. Cranen, J. J. A. Keiren, and T. A. C. Willemse. A Cure for Stuttering Parity Games. In *ICTAC 2012*, volume 7521 of *LNCS*, pages 198–212, 2012.
8. S. Cranen, B. Luttik, and T. A. C. Willemse. Proof graphs for parameterised Boolean equation systems. In *CONCUR 2013*, volume 8052 of *LNCS*, pages 470–484, 2013.
9. K. Fisler and M. Y. Vardi. Bisimulation and model checking. In *CHARME 1999*, volume 1703 of *LNCS*, pages 338–342, 1999.
10. P. Fontana and R. Cleaveland. The Power of Proofs: New Algorithms for Timed Automata Model Checking. In *FORMATS 2014*, volume 8711 of *LNCS*, pages 115–129, 2014.
11. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
12. J. F. Groote and T. A. C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.
13. W. H. Hesselink. Invariants for the construction of a handshake register. *Inf. Process. Lett.*, 68(4):173–177, 1998.
14. G. Kant and J. van de Pol. Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games. In *GRAPHITE 2012*, volume 99 of *EPTCS*, pages 50–65, 2012.
15. J. J. A. Keiren, J. W. Wesselink, and T. A. C. Willemse. Liveness Analysis for Parameterised Boolean Equation Systems. In *ATVA 2014*, volume 8837 of *LNCS*, pages 219–234, 2014.
16. J. J. A. Keiren and T. A. C. Willemse. Bisimulation minimisations for Boolean equation systems. In *HVC 2009*, volume 6405 of *LNCS*, pages 102–116, 2011.
17. D. E. Knuth. Textbook Examples of Recursion. *Artificial and Mathematical Theory of Computation*, 91:207–229, 1991.
18. R. P. J. Koolen, T. A. C. Willemse, and H. Zantema. Using SMT for Solving Fragments of Parameterised Boolean Equation Systems. In *ATVA 2015*, volume 9364 of *LNCS*, pages 14–30, 2015.
19. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
20. D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC '92*, pages 264–274, 1992.
21. Y. Nagae and M. Sakai. Reduced Dependency Spaces for Existential Parameterised Boolean Equation Systems. In *WPTE 2017*, volume 265 of *EPTCS*, pages 67–81, 2018.
22. Y. Nagae, M. Sakai, and H. Seki. An Extension of Proof Graphs for Disjunctive Parameterised Boolean Equation Systems. In *WPTE 2016*, volume 235 of *EPTCS*, pages 46–61, 2017.
23. T. Neele, T. A. C. Willemse, and J. F. Groote. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting (Technical Report). Technical report, Eindhoven University of Technology, 2018.
24. D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183, 1981.
25. S. Tripakis and S. Yovine. Analysis of Timed Systems Using Time-Abstracting Bisimulations. *FMSD*, 18(1):25–68, 2001.
26. 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*, pages 584–598, 2010.

Appendix

Derivation of transition condition in the proof of Lemma 1

Lemma 1. *The dependency space $G = (\mathsf{sig}(\mathcal{E}), E)$ of $\mathcal{E}$ is both a positive and a negative dependency graph.*

Recall that we are considering a node (X_i, v) in a graph G, and we want to prove the G is a positive dependency graph, *i.e.*, r from Definition 4 is *true*. Furthermore, we assumed the equation for X_i is conjunctive. We derive that $[\![\varphi_{X_i}]\!]\eta\delta[v/d] = r$, where $\eta = \mathsf{env}((X_i, v)^\bullet, \mathit{true})$.

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

□

Proof for Lemma 2

Lemma 2. *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 11). 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 11) 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 $b, b' \in V$, $b \mathrel{E} b'$ 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 7, this implies that there is a $j \in J_b$ such that $X_j = X_{b'}$ (the other conditions of Definitions 7 are trivially true in the context of $\mathcal{E}_G$). From the definition of the equations of $\mathcal{E}_G$ (Definition 11), we deduce that this can only be the case if $b \mathrel{E} b'$.

$\Rightarrow$ Let $b, b' \in V$ be two blocks satisfying $b \mathrel{E} b'$. Then, the equation for X_b in $\mathcal{E}_G$ contains a clause that has $X_{b'}(\star)$ as predicate variable (Definition 11). From Definition 7, it follows that $(X_b, \star)E'(X_{b'}, \star)$ and thus we conclude that $R(b)E'R(b')$.

□

Proofs for Section 6

Lemma 3. *Let $G = (V, E)$ be a dependency graph and $G_r = (V_r, E_r)$ a consistent reduced dependency graph of G. Furthermore, let $G'_r = (V'_r, E'_r)$ be a reduced dependency graph that is a subgraph of G_r and $G' = (V', E')$ its base graph. If $V'_r \subseteq V_r$ is stable with respect to itself then G'_r is bisimilar to G'.*

Proof. The situation from Lemma 3 is depicted in the figure below.

$$\begin{array}{ccc} G = (V, E) & \xrightarrow{\text{base graph of}} & G_r = (V_r, E_r) \\ \cup\!| & & \cup\!| \\ G' = (V', E') & \xrightarrow{\text{base graph of}} & G'_r = (V'_r, E'_r) \end{array}$$

For bisimilarity of G'_r and G' we reason as follows. First, note that $V' = \bigcup V'_r$ (see Definition 8). 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.

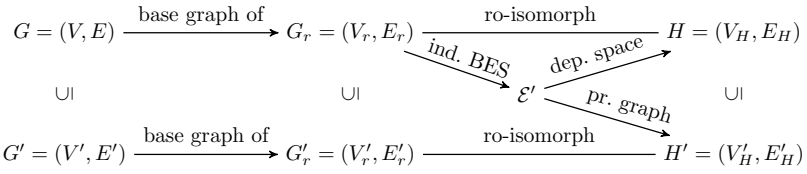
Pick $(X, v)\mathcal{R}b$. By definition, we have $(X, v) \in b$. Note that G'_r is consistent due to consistency of G_r; and we find that both the rank and operand of the equation for X match the rank and operand of the equation for X_b in the BES induced by G'_r. For the transfer conditions we observe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

- From the definition of a reduced graph it follows directly that if $(X, v)E'(Y, w)$, then $b \mathrel{E'_r} b'$, where $(Y, w) \in b'$, *i.e.*, $(Y, w)\mathcal{R}b'$.
- Suppose we have $b \mathrel{E'_r} b'$. 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, $b \mathrel{E'_r} b'$ 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. □

Theorem 4. *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 reduced dependency graph that is a subgraph of G_r and $G' = (V', E')$ its base graph. If $V'_r \subseteq V_r$ is stable with respect to itself and 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 $\mathcal{E}'$.



We observe that G' trivially satisfies the 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 $\eta' = \mathsf{env}((X, v)^\bullet, r)$ and $\eta = \mathsf{env}((X, v)^\bullet \cup \{(Y, w)\}, r)$. From bisimilarity with G'_r (see Lemma 3), 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. □