Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games

Gijs Kant*  
kant@cs.utwente.nl  
Formal Methods & Tools  
University of Twente  
Enschede, The Netherlands

Jaco van de Pol  
vdpol@cs.utwente.nl

Parameterised Boolean Equation Systems (PBESs) are sequences of Boolean fixed point equations with data variables, used for, e.g., verification of modal $\mu$-calculus formulae for process algebraic specifications with data.

Solving a PBES is usually done by instantiation to a Parity Game and then solving the game. Practical game solvers exist, but the instantiation step is the bottleneck.

We enhance the instantiation in two steps. First, we transform the PBES to a Parameterised Parity Game (PPG), a PBES with each equation either conjunctive or disjunctive. Then we use LTSMIN, that offers transition caching, efficient storage of states and both distributed and symbolic state space generation, for generating the game graph. To that end we define a language module for LTSMIN, consisting of an encoding of variables with parameters into state vectors, a grouped transition relation and a dependency matrix to indicate the dependencies between parts of the state vector and transition groups.

Benchmarks on some large case studies, show that the method speeds up the instantiation significantly and decreases memory usage drastically.

Key words: Parameterised Boolean Equation Systems, Parity Games, Instantiation, LTSMIN.

1 Introduction

Parameterised Boolean Equation Systems (PBESs) are sequences of fixed point equations with data variables. They form a very expressive formalism for encoding a wide range of problems, such as the verification of modal $\mu$-calculus formulae [14,6] for process algebraic specifications with data (see, e.g., [10,11]) and checking for (branching) bisimilarity of process equations [7].

PBESs have been described extensively in [11]. A method for solving PBESs directly has been presented [10], but usually PBESs are solved by first instantiating the system to a plain Boolean Equation System (BES) and then solving the BES. Instantiation of PBESs is described in [8,16], where clever rewriters and enumeration of quantifier expressions play an important role. We focus on instantiation to a Parity Game (PG), which is a restricted BES with equations that are either conjunctive or disjunctive. Although no polynomial time algorithm for solving parity games is known (however, the problem is known to be in NP $\cap$ co-NP), effective parity game solvers exist (see, e.g., [9]), especially when the alternation depth is low, and the instantiation step is currently the bottleneck of the whole procedure for many practical cases.

*Gijs Kant is sponsored by the NWO under grant number 612.000.937 (VOCHS).
There are clear similarities between instantiation of PBESs and state space generation, a well known problem in model checking. In both, an abstract description gives rise to a large graph, which requires efficient storage of the generated graph. Also, in both we often have that the description consists of a combination of reasonably independent components or equations. This ‘locality’ can be used to speed up the generation of successor nodes. Inspired by these similarities, we apply in this paper optimisations from model checking to the PBES instantiation problem, devising a more efficient method. We use LTSMIN, a language independent toolset for state space exploration which enables efficient state space generation and offers both symbolic exploration tools based on Binary Decision Diagrams (BDDs) and distributed exploration tools (see, e.g., [5]). The tools make use of knowledge about the dependencies for better efficiency, which can be specified for every language in a separate language module. Instantiating PBESs to parity games in our enhanced method has two phases:

1) Transforming the PBES into an equivalent system that consists of expressions that are either purely conjunctive or purely disjunctive. We call such a system a Parameterised Parity Game (PPG). The result of this operation is that any instantiation of the PPG will result directly in a parity game.

2) Instantiating the PPG to a PG using LTSMIN. To this end we defined a PBES language module for LTSMIN, in which we specify a state vector representation of instantiated PBES variables (and the corresponding node in the generated game graph) and the dependencies between (parts of) the equations and the parts of the state vector.

**Figure 1:** Overview of the verification approach, consisting of various transformations, an instantiation step, and available reductions and solvers.

An overview of the method is shown in Figure[1] We consider PBESs in Bounded Quantifier Normal Form (BQNF), which is a subset of all PBESs, but any PBES can be rewritten automatically to a system in BQNF with the same solution. PBESs and their normal forms are described in Section[2] The contributions of this article are the transformation from BQNF to PPG and the instantiation from PPG to PG. Both steps are not trivial. We will explain here where the obstacles lie.

In general, each system of PBES equations in BQNF can be transformed automatically into a system consisting of equations in PPG while preserving the solution. An equation can be transformed to PPG by introducing fresh equations for subexpressions and replacing the subexpressions by the corresponding variable. However, it is important not to separate quantifiers from the expressions that restrict the data elements that have to be considered, so called bounds. If a bound for a quantifier over an infinite data sort is replaced by a variable, the instantiator might generate an infinite number of successors for a node in the game graph. See Section[3] for our solution.

For the instantiation step we implemented a PBES language module for LTSMIN using the Partitioned Interface for the Next State function (PINS). This includes partitioning each PBES equation into transition groups and defining a dependency matrix that specifies the dependencies between transition group and parts of the state vector. We then have a high-performance instantiation tool that offers both distributed
and symbolic generation of a parity game. This requires some delicacy, as splitting a formula too much may result in infinite computation (as in the transformation phase) and not splitting enough could result in a dependency matrix that is too dense, which ruins the effect of transition caching and symbolic computation. The implementation is described in Section 4.

In Section 5 we present performance results for a number of case studies, comparing our sequential, distributed and symbolic implementations based on LTS\textsc{min} to the existing PBES instantiation tools in the mCRL2 toolset. In almost all cases memory usage is orders of magnitude better for our tool. In all cases also the execution time is much better.

2 Background

In this section we will treat PBESs, normal forms for PBESs, and Parity Games.

2.1 PBES

Definition 2.1. Predicate formulae \( \varphi \) are defined by the following grammar:

\[
\varphi ::= b \mid X(\vec{e}) \mid \neg \varphi \mid \varphi \oplus \varphi \mid Q d : D . \varphi
\]

where \( \oplus \in \{\land, \lor, \Rightarrow\} \), \( Q \in \{\forall, \exists\} \), \( b \) is a data term of sort \( \text{Bool} \), \( X \in \mathcal{X} \) is a predicate variable, \( d \) is a data variable of sort \( D \), and \( \vec{e} \) is a vector of data terms. We will call any predicate formula without predicate variables a simple formula. We denote the class of predicate formulae \( \mathcal{F} \).

Definition 2.2. A First-Order Boolean Equation is an equation of the form:

\[
\sigma X(\vec{d} : D) = \varphi
\]

where \( \sigma \in \{\mu, \nu\} \) is a minimum (\( \mu \)) or maximum (\( \nu \)) fixed point operator, \( \vec{d} \) is a vector of data variables of sort \( D \), and \( \varphi \) is a predicate formula.

Definition 2.3. A Parameterised Boolean Equation System (PBES) is a sequence of First-Order Boolean Equations:

\[
\mathcal{E} = (\sigma_1 X_1(\vec{d}_1 : D_1) = \varphi_1) \ldots (\sigma_n X_n(\vec{d}_n : D_n) = \varphi_n)
\]

The semantics and solution of PBESs are described in, e.g., [11]. We say that two equation systems \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) are equivalent, written as \( \mathcal{E}_1 \equiv \mathcal{E}_2 \), if they have the same solution for every variable that occurs in both systems.

We adopt the standard limitations: expressions are in positive form (negation occurs only in data expressions) and every predicate variable occurs exactly once as the left hand side of an equation. A PBES that contains no quantifiers and parameters is called a Boolean Equation System (BES). A finitary PBES can be instantiated to a BES by expanding the quantifiers to finite conjunctions or disjunctions and substituting concrete values for the data parameters. Every instantiated PBES variable \( X(\vec{e}) \) should then be read as a BES variable “\( X(\vec{e}) \)”.

A one-to-one mapping can be made from a BES to an equivalent parity game if the BES has only expressions that are either conjunctive or disjunctive. The parity game is then represented by a game graph with nodes that represent variables with concrete parameters and edges that represent dependencies. Parity games will be further explained in Section 2.2. To make instantiation of a PBES to a parity game more directly we will preprocess the PBES to a format that only allows expressions to be either conjunctive or disjunctive. This format is a normal form for PBESs that we call the Parameterised Parity Game, defined as follows:
Definition 2.4. A PBES is a Parameterised Parity Game (PPG) if every right hand side of an equation is a formula of the form:

\[ \bigwedge_{i \in I} f_i \land \bigwedge_{j \in J} \forall \vec{\theta} \in D_j \cdot (g_j \Rightarrow X_j(\vec{e}_j)) \lor \bigvee_{i \in I} f_i \lor \bigvee_{j \in J} \exists \vec{\theta} \in D_j \cdot (g_j \land X_j(\vec{e}_j)) . \]

where \( f_i \) and \( g_j \) are simple boolean formulae, and \( \vec{e}_j \) is a data expression. \( I \) and \( J \) are finite (possibly empty) index sets.

The expressions range over two index sets \( I \) and \( J \). The left part is a conjunction (or disjunction) of simple expressions \( f_i \) that can be seen as conditions that should hold in the current state. The right part is a conjunction (or disjunction) of a quantified vector of variables for next states \( X_j \) with parameters \( \vec{e}_j \), guarded by simple expression \( g_j \).

Before transforming arbitrary PBESs to PPGs we first define another normal form on PBESs to make the transformation easier. This normal form can have an arbitrary sequence of bounded quantifiers as part of the equation with the remainder of the expression as right hand side), which would result in an infinite disjunction when instantiating the PPG. BQNF allows the original expression to be rewritten to a conjunctive part and existential quantification is allowed within the disjunctive parts. These bounds are needed to avoid problems when transforming to PPG. Consider the expression

\[ \forall i : \mathbb{N} \cdot (i < 5) \Rightarrow Y(i) \lor (\exists j : \mathbb{N} \cdot (j < 3) \land Z(j)) . \]

Rewriting that expression to PPG would split the expression such that the initial expression is \( \exists j : \mathbb{N} \cdot X_1(j) \) (\( X_1 \) is a newly introduced variable for the equation with the remainder of the expression as right hand side), which would result in an infinite disjunction when instantiating the PPG. BQNF allows the original expression to be rewritten to \( \exists j : \mathbb{N} \cdot (j < 3) \land \forall i : \mathbb{N} \cdot (i < 5) \Rightarrow (Y(i) \lor Z(j)) \) with the bounds close to the quantifiers, which allows to split the expression after the bound, preventing the instantiation to result in an infinite expression. Requiring that a system is specified in BQNF does not limit the expressiveness, as each PBES can be transformed into an equivalent system in PFNF that has the same solution and PFNF is a subset of BQNF.

The translation from process algebraic specifications in mCRL2 and \( \mu \)-calculus formulae to PBESs is given in [10] and is illustrated by the following example. Throughout the paper we expect the reader to know process algebras and to be able to read mCRL2 specifications.

1 A transformation to PFNF is implemented in the pbesrewr tool and documented at http://www.win.tue.nl/mcrl2/wiki/index.php/Parameterised_Boolean_Equation_Systems

2 See http://mcrl2.org for documentation on the mCRL2 language.
Example 2.6 (Buffer). Consider the specification of a simple buffer with a capacity of 2.

\[
\text{sort } D = \text{struct } d_1 \mid d_2; \quad \text{act } r_1, s_4 : D; \\
\text{proc } \text{Buffer}(q : \text{List}(D)) = \sum_{d \in D} (\#q < 2) \rightarrow r_1(d) \cdot \text{Buffer}(q \triangleleft d) \\
\quad + (q \neq []) \rightarrow s_4(\text{head}(q)) \cdot \text{Buffer}((\text{tail}(q)));
\]

\text{init } \text{Buffer}([[]]);

The specification consists of sort and action definitions, process specifications where alternatives are in summands and an initial state. On the first line an enumerated data sort \( D \) is introduced with data values \( d_1 \) and \( d_2 \), and the actions \( r_1 \) and \( s_4 \) are specified, both having a data parameter of type \( D \). A process \( \text{Buffer} \) is specified that has a data parameter \( q \), which is a list of elements of type \( D \). The process consists of summands, separated by the + operator. Each summand may start with a summation over a data set, followed by a guard that is closed with a \( \rightarrow \), then an action, followed by a call to the process that describes the behaviour after the action, typically a recursive call to the process itself with different parameters.

The first summand specifies that any element \( d \) can be added to \( q \) by the action \( r_1(d) \) if the size of the internal buffer \( q \) is smaller than 2. The second summand specifies that if \( q \) is not empty, elements can be popped by the action \( s_4(\text{head}(q)) \). The initial state of the system is the \( \text{Buffer} \) process with an empty list in this case, which models that initially the buffer is empty.

We can check the specification for absence of deadlock, which is expressed in \( \mu \)-calculus as follows:

\[
[\top^*] (\top) \top \quad \text{(which is syntactic sugar for: } \nu X . (\top) \top \land [\top] X) 
\]

which reads: after any sequence of actions (\([\top^*]\)), always some action is enabled (\((\top) \top\)). Satisfaction of the formula by the specification, translated to a PBES, looks as follows:

\[
\text{sort } D = \text{struct } d_1 \mid d_2; \\
\text{pbes } \nu X(q : \text{List}(D)) = (q \neq []) \lor (\#q < 2) \\
\quad \land (q \neq []) \Rightarrow X(\text{tail}(q)) \\
\quad \land \forall d \in D . (\#q < 2) \Rightarrow X(q \triangleleft d); \\
\text{init } X([[]]);
\]

This PBES is true if from the initial state \( X([[]]) \) an element can be added to \( q \) if \#\( q \) is smaller than 2, an element can be popped from \( q \) if it is not empty and any of these actions is enabled (\( q \neq [] \) or \#\( q < 2 \), which is obviously true for any \( q \)). The same has to hold for the successor states (\( X \) with an element added to, respectively popped from \( q \) as parameter). The solution of the PBES is \text{true}.

Remark. The equation system in the example above is already a PPG, which is no coincidence as any system when combined with the absence of deadlock property will result in a PBES in PPG form because of the form of the formula: a conjunction of “we can do an action now” (a disjunctive expression without recursion) and “for all possible actions the property holds in all next states” (universal quantification with recursion). Note that checking the absence of deadlock property is almost the same as standard reachability analysis.

Definition 2.7 (Block). A PBES is divided into blocks, which are subsequences of equations with the same fixed point operator such that subsequent equations with the same fixed point operator belong to the same block.

2.2 Parity Games

A parity game is a game between two players, player 0 (also called Eloise or player even) and player 1 (also called Abelard or player odd), where each player owns a set of places. On one place a token is
placed that can be moved by the owner of the place to an adjacent place. The parity game is represented as a graph. We borrow notation from \cite{6} and \cite{15}.

**Definition 2.8 (Parity Game).** A parity game is a graph $G = \langle V, E, V_0, V_1, v_1, \Omega \rangle$, with

- $V$ the set of vertices (or places or states);
- $E: V \times V$ the set of transitions;
- $V_0 \subseteq V$ the set of places owned by player 0;
- $V_1 \subseteq V$ the set of places owned by player 1;
- $v_1 \in V$ the initial state of the game;
- $\Omega: V \to \mathbb{N}$ assigns a priority $\Omega(v)$ to each vertex $v \in V$;

where $V_0 \cup V_1 = V$ and $V_0 \cap V_1 = \emptyset$.

The nodes in the graph represent the places and correspond to the instantiated variables from the equation system. The edges represent possible moves of the token (initially placed on $v_1$) and encode dependencies between variables. A node does not necessarily have outgoing transitions, i.e., deadlock nodes are allowed. In the parity game, player 0 owns the nodes that represent disjunctions, player 1 the nodes that represent conjunctions.

The node priorities correspond to the number of the block to which the corresponding variable belongs (see Def. 2.7), such that variables in earlier blocks have lower priorities, $\nu$-blocks have even priorities, $\mu$-blocks have odd priorities and the earliest $\mu$-block has priority 1. The following table shows an intuitive overview of the relations between BESs and parity games.

| $\nu$ blocks | Even priorities (0, 2, 4, …) |
| $\mu$ blocks | Odd priorities (1, 3, 5, …) |
| $\vee, \exists, \langle$ | Player 0, \textit{Else}, Even, Prover |
| $\wedge, \forall, \rangle$ | Player 1, \textit{Bel}, Odd, Refuter |

The values true ($\top$) and false ($\bot$) are represented as a node with priority 0, player 1 and a transition to itself, and a node with priority 1, player 0 and a transition to itself, respectively.

A play in the game is a finite path $\pi = v_0v_1\cdots v_r \in V^+$ ending in a deadlock state $v_r$ or an infinite path $\pi = v_0v_1\cdots \in V^\omega$ such that $(v_i, v_{i+1}) \in E$ for every $v_i \in \pi$. Priority function $\Omega$ extends to plays in the following way: $\Omega(\pi) = \Omega(v_0)\Omega(v_1)\cdots$. $\text{Inf}(\rho)$ returns the set of values that occur infinitely often in a sequence $\rho$.

**Definition 2.9 (Winner of a play).** Player 0 is the winner of a play $\pi$ if

- $\pi$ is a finite play $v_0v_1\cdots v_r \in V^+$ and $v_r \notin V_0$ and no move is possible from $v_r$; or
- $\pi$ is an infinite play and $\min(\text{Inf}(\Omega(\pi)))$, the minimum of the priorities that occur infinitely often in $\pi$, is even. This is called the min-parity condition.

**Definition 2.10 (Strategy).** A (memoryless) strategy for player $a$ is a function $f_a: V_a \to V$. A play $\pi = v_0v_1\cdots$ is conform to $f_a$ if for every $v_i \in \pi$, $v_i \in V_a \Rightarrow v_{i+1} = f_a(v_i)$.

**Definition 2.11 (Winner of the game).** Player 0 is the winner of the game if and only if there exists a winning strategy for player 0, i.e., from the initial state every play conforming to the strategy will be won by player 0.

The model checking problem is encoded as a PBES (see \cite{10}) which is instantiated to a parity game (see \cite{16}) such that player 0 is the winner of the game iff the property holds for the system.
Solving Parity Games  Solving a parity game means finding a winning strategy for one of the players. Various algorithms exist, such as the recursive algorithm by Zielonka [20] and Small Progress Measures by Jurdziński [13], with a multi-core implementation in [13]. An overview and performance comparison of the algorithms are given in [9].

3 Transformation from BQNF to Parameterised Parity Games

In order to automatically transform a PBES to a PPG, we define a transformation function \( s \) from BQNF to PPG. The transformation rewrites expressions that contain both conjunctions and disjunctions to equivalent expressions that are either conjunctive or disjunctive, by introducing new equations for certain subformulae and substituting calls to the new equations for these subformulae in the original expression.

The function \( t \) below replaces an expression by a call to a new equation if the expression is not already a variable instantiation. The function \( t' \) introduces a new equation for an expression if needed.

\[
\begin{align*}
t(X, \vec{d}, \varphi) & \overset{\text{def}}{=} \\
& \begin{cases} \\
\varphi & \text{if } \varphi \text{ is of the form } X'(\vec{c}), \\
\text{X}(\vec{d}) & \text{otherwise}; \\
\end{cases} \\
t'(\sigma, X, \vec{d}, \varphi) & \overset{\text{def}}{=} \\
& \begin{cases} \\
\emptyset & \text{if } \varphi \text{ is of the form } X'(\vec{c}), \\
\sigma(\text{X}(\vec{d}) = \varphi) & \text{otherwise}. \\
\end{cases}
\end{align*}
\]

For brevity, we leave out the types of the parameters. A tilde is used to introduce a fresh variable: \( \vec{X} \). For equation system \( \mathcal{E} = (\sigma X_1(\vec{d}_1) = \xi_1) \ldots (\sigma X_n(\vec{d}_n) = \xi_n) \), with each \( \xi_i \) in BQNF, the translation to PPG is defined as follows:

\[
\begin{align*}
s(\mathcal{E}) & \overset{\text{def}}{=} s(\sigma X_1(\vec{d}_1) = \xi_1) \ldots s(\sigma X_n(\vec{d}_n) = \xi_n) \\
s(\sigma X(\vec{d}) = f) & \overset{\text{def}}{=} \sigma X(\vec{d}) = f \\
s(\sigma X(\vec{d}) = \forall \vec{v} \cdot b \Rightarrow \varphi) & \overset{\text{def}}{=} \left( \sigma X(\vec{d}) = \forall \vec{v} \cdot b \Rightarrow t(\vec{X}, \vec{d} + \vec{v}, \varphi) \right) \\
& \quad t'(\sigma, \vec{X}, \vec{d} + \vec{v}, \varphi) \\
s(\sigma X(\vec{d}) = \exists \vec{v} \cdot b \land \varphi) & \overset{\text{def}}{=} \left( \sigma X(\vec{d}) = \exists \vec{v} \cdot b \land t(\vec{X}, \vec{d} + \vec{v}, \varphi) \right) \\
& \quad t'(\sigma, \vec{X}, \vec{d} + \vec{v}, \varphi) \\
s(\sigma X(\vec{d}) = \bigwedge_{k \in K} f_k \\
\quad \land \bigwedge_{i \in I} (\forall \vec{v}_i \cdot g_i \Rightarrow \varphi_i)) & \overset{\text{def}}{=} \left( \sigma X(\vec{d}) = \bigwedge_{k \in K} f_k \\
& \quad \land \bigwedge_{i \in I} (\forall \vec{v}_i \cdot g_i \Rightarrow t(\vec{X}_i, \vec{d} + \vec{v}_i, \varphi_i)) \right) \\
& \quad t'(\sigma, \vec{X}_i, \vec{d} + \vec{v}_i, \varphi_i) \ldots t'(\sigma, \vec{X}_m, \vec{d} + \vec{v}_m, \varphi_m) \\
s(\sigma X(\vec{d}) = \bigvee_{k \in K} f_k \\
\quad \lor \bigvee_{i \in I} (\exists \vec{v}_i \cdot g_i \land \varphi_i)) & \overset{\text{def}}{=} \left( \sigma X(\vec{d}) = \bigvee_{k \in K} f_k \\
& \quad \lor \bigvee_{i \in I} (\exists \vec{v}_i \cdot g_i \land t(\vec{X}_i, \vec{d} + \vec{v}_i, \varphi_i)) \right) \\
& \quad t'(\sigma, \vec{X}_i, \vec{d} + \vec{v}_i, \varphi_i) \ldots t'(\sigma, \vec{X}_m, \vec{d} + \vec{v}_m, \varphi_m) \\
\end{align*}
\]

with \( I = 1 \ldots m, \vec{v} \cap \vec{d} = \emptyset \) (variables in \( \vec{v} \) do not occur in \( \vec{d} \)), \( b, f, f_k, g_i \) are simple formulae, \( \varphi, \varphi_i \) are formulae that may contain predicate variables.
Proposition 3.1. The transformation $s$ is solution preserving, i.e., for any $E$ in BQNF, $s(E) \equiv E$: bound variables $X(d)$ have the same solution in $s(E)$ as in $E$.

Proof. Every change made by $s$ to an equation $\sigma X = \xi$ is a substitution of a subexpression $\varphi$ by a fresh variable $\tilde{X}$, while adding at the same time a new equation $\sigma \tilde{X} = \varphi$ in the same block as $X$. We can apply backward substitution (using [11, Lemma 18]) $s(\sigma X = \xi) [\tilde{X} := \varphi]$ for every substitution caused by the transformation to get the original equation system (plus an unused equation $s(\sigma \tilde{X} = \varphi)$ for every fresh variable $\tilde{X}$). From that we can conclude that $s(E) \equiv E$. \qed

Example 3.2 (Example of the transformation). We combine the buffer from Example 2.6 with the property that in every state both $r_1$ and $s_4$ actions are enabled:

$$\nu X : (\exists_d : D . (r_1(d)) X) \land (\exists_d : D . (s_4(d)) X)$$

The resulting PBES has an equation which does not conform to the PPG form, but is in BQNF:

```plaintext
sort D = struct d1 | d2;

pbes $\nu X(q : List(D)) = (\exists_d : D . (#q < 2) \land X(q < d))$
\land ($\exists_d : D . (head(q) = d) \land (q \neq []) \land X(tail(q))$);

init X([]);
```

The transformation $s$ replaces both conjuncts by a fresh variable and adds equations for these variables with the substituted expression as right hand side, resulting in equations:

```plaintext
pbes $\nu X(q : List(D)) = X_1(q) \land X_2(q)$;
$\nu X_1(q : List(D)) = \exists_d : D . (#q < 2) \land X(q < d)$;
$\nu X_2(q : List(D)) = \exists_d : D . (head(q) = d) \land (q \neq []) \land X(tail(q))$;
```

The first equation is purely conjunctive, while that latter two equations are (guarded) disjunctive.

4 Instantiation of Parameterised Parity Games

We view the instantiation of PPGs to Parity Games as generating a transition system, where states are predicate variables with concrete parameters and transitions are dependencies, specified by the right hand side of the corresponding equation in the PPG.

Example 4.1. Consider the equation:

$$\nu X(d : D) = (d > 0 \land d < 10) \Rightarrow X(d - 1) \land X(d + 1)$$

If $X(5)$ is the initial value, its successors are $X(4)$ and $X(6)$, so the graph starts with a node owned by player 1 representing $X(5)$ with transitions to nodes $X(4)$ and $X(6)$.

4.1 LTSMIN

We use the tool LTSMIN to generate a parity game given a PPG. LTSMIN is a language independent tool for state-space generation [5]. Different language-modules are available, which are connected to different exploration algorithms through the so-called PINS-interface. This interface allows for certain language-independent optimisations, such as transition caching and distributed generation (see [4]), and an efficient compressed storage of states in a tree database (see [2]). Also symbolic reachability analysis is possible, where the state space is stored as a Binary Decision Diagram (BDD) [3].
4.1.1 Partitioned Interface for the Next State function

LTS\textsubscript{MIN} uses a Partitioned Interface for the Next State function (\textsc{PINS}), where states are represented as a vector \(\langle x_1, x_2, \ldots, x_M \rangle\) with size \(M\) that is fixed for the whole system (to be determined statically). These values are stored in a globally accessible table, so that the states can also be represented as a vector of integer indices \(\langle i_1, i_2, \ldots, i_M \rangle\). The \textsc{PINS} interface functions on this level of integer vectors, so that each tool can really be language-independent. Throughout the text we will often use value vectors instead of index vectors for better readability.

For a system with a state vector of \(M\) parts, the universe of states is \(S = \mathbb{N}^M\). For each language module a transition function \(\text{NEXT} : S \to \mathcal{P}(S)\) has to be defined that computes the set of successor states for a given state. This transition relation is preferably split into transition groups in order to reflect the compositional structure of the system, by defining a function \(\text{GROUP-NEXT} : S \times \mathbb{N} \to \mathcal{P}(S)\) that computes successors for state \(s\) as defined in group \(k\). Suppose we have \(K\) transition groups. \(\text{NEXT}\) can then be defined as

\[
\text{NEXT}(s) = \bigcup_{k=1}^{K} \text{GROUP-NEXT}(s, k)
\]

4.1.2 Dependence

An important optimisation comes from the observation that not all parts of the state vectors are relevant in every transition group. To indicate the relevant parts of the vector for each of the transition groups, LTS\textsubscript{MIN} uses a dependency matrix, which has to be computed statically.

**Definition 4.2** (\textsc{PINS} Matrix: [4], Def. 4). A dependency matrix \(D_{K \times N} = DM(P)\) for system \(P\) is a matrix with \(K\) rows and \(N\) columns containing \(\{0, 1\}\) such that if \(D_{k,i} = 0\) then group \(k\) is independent of element \(i\).

For any transition group \(1 \leq k \leq K\), we define \(\pi_k\) as the projection \(\pi_k : S \to \Pi_{\{1 \leq i \leq N \mid D_{k,i} = 1\}} S_i\).

Independence here means that for given transition group \(k\) the transitions do not depend on part \(i\) of the state vector (read independence) and the transitions do not change part \(i\) of the successor state vector (write independence) or that part \(i\) is irrelevant in both the current state and all successor states. Irrelevant here means that changing the value of that part would still result in a bisimilar state space. For a more precise definition, see [17, Def. 9]. This definition of independence is slightly more liberal than the one in [4] in that we added this notion of relevance.

4.1.3 Transition caching

One way of exploiting the dependency information in the matrix is by using transition caching. Only the dependent parts of the transition are stored in a cache \(C_k\) for every group \(k\) by using the projection function \(\pi_k\), as described in [4] and shown in Alg. 4.1. This way time is saved, because caching of transitions avoids calling \(\text{GROUP-NEXT}\) at every step. The density of the matrix has great influence on the performance of caching and of the symbolic tools.

4.2 PBES Language Module

In this section we describe states, transition groups and the dependency matrix for PPGs. We assume to have a rewriter \texttt{simplify} that is powerful enough to evaluate any closed data expression to \texttt{true} or \texttt{false} or to a disjunction or conjunction of predicate variables with closed data expressions as parameters. We use the same rewriter by Van Weerdenburg [19] as used in [16].
Algorithm 4.1 NEXT-CACHE(s, k) computes successors of s for group k using a cache.

<table>
<thead>
<tr>
<th>NEXT-CACHE(s, k)</th>
<th>UPDATE-CACHE(s, k)</th>
<th>NEXT-APPLY(s, t, k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Update-CACHE(s, k)</td>
<td>1: if π_k(s) ∈ dom(C_k) then</td>
<td>1: j := 1</td>
</tr>
<tr>
<td>2: S := ∅</td>
<td>2: S := ∅</td>
<td>2: for 1 ≤ i ≤ N do</td>
</tr>
<tr>
<td>3: for all t ∈ C_k[π_k(s)] do</td>
<td>3: S′ := GROUP-NEXT(s, k)</td>
<td>3: if D_{k,i} = 0 then</td>
</tr>
<tr>
<td>4: t′ := NEXT-APPLY(s, t, k)</td>
<td>4: for all s′ ∈ S′ do</td>
<td>4: s′[i] := s[i]</td>
</tr>
<tr>
<td>5: Add t′ to S</td>
<td>5: Add π_k(s′) to S</td>
<td>5: else</td>
</tr>
<tr>
<td>6: return S</td>
<td>6: C_k[π_k(s)] := S</td>
<td>6: s′[i] := t[j]</td>
</tr>
<tr>
<td></td>
<td>7: j := j + 1</td>
<td>7: j := j + 1</td>
</tr>
<tr>
<td></td>
<td>8: return s′</td>
<td>8: return s′</td>
</tr>
</tbody>
</table>

4.2.1 States and transition groups

For PPGs, the state vector is partitioned as follows: \( \langle x_1, x_2, \ldots, x_M \rangle \), where X is a propositional variable, and for \( i \in \{1 \ldots M\} \) each \( x_i \) is the value of parameter \( i \). \( M \) is the total number of parameter signatures in the system (consisting of name and type).

We assume the existence of a function \( \text{priority} : S \rightarrow \text{Int} \) that assigns a priority to each state (based on the block of the corresponding equation) and a function \( \text{player} : S \rightarrow \{0, 1\} \) that assigns a player to each state (0 if the corresponding expression is a disjunction, 1 if it is a conjunction). In particular, the true state has priority 0 and is owned by player 1 and the false state has priority 1 and belongs to player 0.

The equations in the PPG specify the transitions between states. The right hand side of the equation is split into conjuncts or disjuncts if possible, which form the transition groups, which are numbered subsequently. We use a mapping \( \text{var} : \text{Int} \rightarrow X' \) from group number to variable and a mapping \( \text{expr} : \text{Int} \rightarrow \mathcal{F} \) from group number to corresponding conjunct or disjunct. In the following we assume the index sets \( I \) and \( J \) to be disjoint.

For a sequence of equations of the form

\[
\sigma X(\vec{d} : D) = \bigwedge_{i \in I} f_i \land \bigwedge_{j \in J} \forall \vec{v} \in D_j . (g_j(\vec{d}, \vec{v}) \Rightarrow X_j(e_j(\vec{d}, \vec{v}))),
\]

for each \( i \in I \) there is a group \( k \) with \( \text{expr}(k) = f_i \) and for each \( j \in J \) there is a group \( k \) with

\[
\text{expr}(k) = \forall \vec{v} \in D_j . (g_j(\vec{d}, \vec{v}) \Rightarrow X_j(e_j(\vec{d}, \vec{v}))),
\]

and \( \text{var}(k) = X \). Symmetrically for disjunctive equations.

Example 4.3. We will explain these concepts using a specification of two sequential buffers (buffer.2):

\[\begin{align*}
\text{proc} \ln(i : \text{Pos}, q : \text{List(D)}) &= \sum_{d:D} \#q < 2 \rightarrow r_1(d) \cdot \ln(i, q < d) \\
&\quad + (q \neq []) \rightarrow w(i+1, \text{head}(q)) \cdot \ln(i, \text{tail}(q)); \\
\text{proc} \text{Out}(i : \text{Pos}, q : \text{List(D)}) &= \sum_{d:D} \#q < 2 \rightarrow r(i, d) \cdot \text{Out}(i, q < d) \\
&\quad + (q \neq []) \rightarrow s_4(\text{head}(q)) \cdot \text{Out}(i, \text{tail}(q)); \\
\text{init hide} &\{c\}, \allow(\{r_1, c, s_4\}, \comm(\{w \mid r \rightarrow c\}, \ln(1, []) || \text{Out}(2, [])))\;.
\end{align*}\]

The specification of the initial state the system is specified as composed of an ln and an Out component, composed with the parallel composition (||) operator. Synchronisation of r and w actions of the two processes proceeds in two steps. The simultaneous occurrence of actions r and w (the multi-action w \( \rightarrow r \) is renamed to c (comm) and separate occurances of r and w are ruled out by the restriction operator (allow). The internal action c is hidden (hide). This specification is translated to a single process by linearising it to Linear Process Specification (LPS) format. The result is the following specification:
\[
\text{proc } P(q_{in}, q_{out} : \text{List}(D)) = \sum_{d \in D} (\#q_{in} < 2) \rightarrow r_1(d) \cdot P(q_{in} \triangleleft d, q_{out}) \\
+ (q_{out} \neq \emptyset) \rightarrow s_4(\text{head}(q_{out})) \cdot P(q_{in}, \text{tail}(q_{out})); \\
+ (q_{in} \neq \emptyset \land \#q_{out} < 2) \rightarrow \text{tau} \cdot P(\text{tail}(q_{in}), q_{out} \triangleleft \text{head}(q_{in})) \\
+ \text{delta};
\]

\textbf{init } P([], []);

The result of hiding the \( c \) action is the internal \( \text{tau} \) transition in the third summand. Actions that are not in the set \{\( r_1, c, s_4 \)\} are replaced by a \text{delta} as a result of the restriction operator.

For this process specification, we want to verify the property that if a message is read through \( r_1 \), it will eventually be sent through \( r_1 \).

\[ \nu Y \cdot (\forall d : D \cdot ([r_1(d)](\mu X \cdot ((\text{true}) \land \neg s_4(d) X))) \land \text{true} Y) \]

Satisfaction of this formula by the LPS translates to the following PBES:

\textbf{pbes } \nu Y(q_{in}, q_{out} : \text{List}(D)) = 
\begin{align*}
& (\forall d : D \cdot (\#q_{in} < 2) \Rightarrow X(q_{in} \triangleleft d, q_{out}, d)) \quad (1) \\
& \land (\forall d : D \cdot (\#q_{in} < 2) \Rightarrow Y(q_{in} \triangleleft d, q_{out})) \quad (2) \\
& \land ((q_{out} \neq \emptyset) \Rightarrow Y(q_{in}, \text{tail}(q_{out}))) \quad (3) \\
& \land ((q_{in} \neq \emptyset \land \#q_{out} < 2) \Rightarrow Y(\text{tail}(q_{in}), q_{out} \triangleleft \text{head}(q_{in}))); \quad (4)
\end{align*}

\[ \mu X(q_{in}, q_{out} : \text{List}(D), d : D) = 
\begin{align*}
& (\#q_{in} < 2) \lor (q_{out} \neq \emptyset) \lor (q_{in} \neq \emptyset \land \#q_{out} < 2) \quad (5) \\
& \land (\forall d : D \cdot (\#q_{in} < 2) \Rightarrow X(q_{in} \triangleleft d, q_{out}, d)) \quad (6) \\
& \land ((\text{head}(q_{out}) \neq d) \land (q_{out} \neq \emptyset) \Rightarrow X(q_{in}, \text{tail}(q_{out}), d)) \quad (7) \\
& \land ((q_{in} \neq \emptyset \land \#q_{out} < 2) \Rightarrow X(\text{tail}(q_{in}), q_{out} \triangleleft \text{head}(q_{in}), d)); \quad (8)
\end{align*}

\textbf{init } Y([], []);

For this equation system, the structure of the state vector is \( (X, q_{in}, q_{out}, d) \). The initial state would be encoded as \( \langle Y, [], [], 0 \rangle \). Since the initial state has no parameter \( d \), a default value is chosen. The numbers \( 1, 2 \) behind the equation parts denote the different transition groups, i.e., each conjunct of a conjunctive expression forms a group. For instance, for group \( 3 \) the associated expression is \( \text{expr}(3) = ((q_{out} \neq \emptyset) \Rightarrow Y(q_{in}, \text{tail}(q_{out}))) \) and it is associated with variable \( \text{var}(3) = Y \). Group \( 1 \) encodes the \([r_1(d)] \varphi \) part of the formula (where \( \varphi \) is the \( \mu X \) part of the formula), groups \( 2 \) encode the \( \text{true} Y \) part, group \( 5 \) encodes that a transition is enabled \( ((\text{true}) \text{true}) \), and groups \( 6, 8 \) encode the cases that not an \( r_1(d) \) transition is taken.

For an equation \( \sigma X(d : D) = \varphi \), let \( \text{params}(X) \) be the list of parameters \( d \) and \( \text{params}(X)_i \) the \( i \)-th element of that list. The next state function \( \text{GROUP-NEXT} \) is defined as follows. For every \( k \) with \( \text{var}(k) = X \),

\[
\text{GROUP-NEXT}(X(\bar{v}), k) \defeq \begin{cases} 
\{\text{simplify}(f[\text{params}(X) := \bar{\epsilon}] ) \} \\
\text{if } f = \text{expr}(k) \text{ is a simple formula;}
\{X'(h(\bar{e}, \bar{v})) \mid \bar{v} \in D \land g(\bar{e}, \bar{v}) \} \\
\text{if } \text{expr}(k) \text{ is of the form } Q\bar{v} \in D . (g(\bar{e}, \bar{v}) \oplus X'(h(\bar{e}, \bar{v}))
\end{cases}
\]
Note that if \( f \) is a simple expression, \( \text{simplify}(f[\text{params}(X):=\vec{c}]) \) will result in either \textbf{true} or \textbf{false}. In the case that \( f \) is not simple, all concrete variable instantiations are enumerated for every quantifier variable \( \vec{v} \) for which the guard \( g \) is satisfied.

**Example 4.4.** For the example above, GROUP-NEXT(\( Y([],[]) \)) yields the empty set because \( q_{\text{out}} = [] \). GROUP-NEXT(\( Y([],[]) \)) results in \{\( Y([d_1],[]), Y([d_2],[]) \} \).

### 4.2.2 Dependency matrix

Let \( \text{occ}(\varphi) \) be the set of propositional variable occurring in a term \( \varphi \), let \( \text{free}(d) \) be the set of free data variables occurring in a data term \( d \), and \( \text{used}(\varphi) \) the set of free data variables occurring in an expression \( \varphi \) such that the variables are not merely passed on to the next state. E.g., with \( \text{X}(a,b) = \xi \), for the expression \( \varphi = a \land \text{X}(c,b) \), \( \text{used}(\varphi) = \{a,c\} \). Parameter \( b \) is not in the set because it does not influence the computation, but is only passed on to the next state. For a formula \( \varphi \), the function \( \text{changed}(\varphi) \) computes the variable parameters changed in the formula:

\[
\text{changed}(\text{X}(e_1, \ldots, e_m)) \overset{\text{def}}{=} \{d_i \mid i \in \{1, \ldots, m\} \land d_i = \text{params}(X)_i \land e_i \neq d_i\}
\]

The function \( \text{tf}(\varphi) \) determines if \( \varphi \) contains a branch that directly results in a \textbf{true} or \textbf{false} (not a variable). This is needed because the boolean constants are encoded as a vector with variable names “true” and “false”, hence a transition to one of them changes the first part of the state vector. For group \( k \) and part \( i \), we define read dependence \( d_R \) and write dependence \( d_W \):

\[
d_R(k,i) \overset{\text{def}}{=} \begin{cases} \text{true} & \text{if } i = 1; \\ p_i \in (\text{params}(\text{var}(k)) \cap \text{used}(\text{expr}(k))) & \text{otherwise.} \end{cases}
\]

\[
d_W(k,i) \overset{\text{def}}{=} \begin{cases} (\text{occ}(\text{expr}(k)) \setminus \{\text{var}(k)\} \neq \emptyset) \lor \text{tf}(\text{expr}(k)) & \text{if } i = 1; \\ p_i \in \text{changed}(\text{expr}(k)) & \text{otherwise.} \end{cases}
\]

\( d_R(k,1) \) is true for every group \( k \), since the variable has to be read to determine if a transition group is applicable.

**Definition 4.5 (PPG Dependency matrix).** For a PPG \( P \) the dependency matrix \( DM(P) \) is a \( K \times M \) matrix defined for \( 1 \leq k \leq K \) and \( 1 \leq i \leq M \) as:

\[
DM(P)_{k,i} = \begin{cases} + & \text{if } d_R(k,i) \land d_W(k,i); \\ r & \text{if } d_R(k,i) \land \neg d_W(k,i); \\ w & \text{if } \neg d_R(k,i) \land d_W(k,i); \\ - & \text{otherwise.} \end{cases}
\]

**Example 4.6.** For the PBES in Example 4.3, the dependency matrix looks like this:

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( q_{\text{in}} )</th>
<th>( q_{\text{out}} )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>( w )</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>( r )</td>
<td>( r )</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>( r )</td>
</tr>
<tr>
<td>8</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

The first row lists the state vector parts. The left column lists the group numbers. A ‘+’ denotes both read and write dependency, ‘\( w \)’ denotes write dependency, ‘\( r \)’ read dependency, and ‘-’ no dependency between the group and the state vector part. For group \( 1 \) we can see that the variable is changed from \( Y \) to \( X \), which results in a ‘+’ in the \( X \) column. The \( q_{\text{in}} \) parameter is both read and changed (\( d \) is added to it). The \( q_{\text{out}} \) parameter is not touched, which results in a ‘-’. The parameter \( d \) is not in \text{params}(\( Y \)) and therefore there is no read dependence. However, the value of \( d \) is set for the next state, resulting in a ‘\( w \)’ in the last column.
5 Performance Evaluation

In this section we report the performance of our tools compared to existing tools in the mCRL2 toolset.

5.1 Experiment setup

As input we used PBESs that are derived from the following mCRL2 models: $n$ sequential buffers (buffer-*), the Sliding Window Protocol (SWP), the IEEE 1394 protocol, a Sokoban puzzle, and state machines that are part of the control system for an experiment at CERN (wheel_sector), described in [12]. The models are combined with $\mu$-calculus properties that check absence of deadlock (nodeadlock, see Example 2.6), if $x$ is read, then eventually $x$ will be written (evt_send, see Example 4.3), or that from the initial state there is a path on which a push action is possible (always_push: $\langle$true$\rangle$)$\langle$push$\rangle$true – only applicable to the Sokoban puzzle).

As preprocessing steps, we applied pbesparelm and pbesrewr -psimplify to every equation system, which are rewriters that apply some obvious simplifications to the equation systems. In the reported cases no transformation to PPG was needed, as the systems were already in the required form.

The tools that we compared are:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Toolset</th>
<th>Groups</th>
<th>Caching</th>
<th>Distributed</th>
<th>Symbolic</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbes2bes</td>
<td>mCRL2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>pbes2bes -rjittyyc</td>
</tr>
<tr>
<td>pbespgsolve</td>
<td>mCRL2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>pbespgsolve -rjittyyc -g</td>
</tr>
<tr>
<td>pbes2lts -black</td>
<td>LTSMIN</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>pbes2lts-grey --black --always-split</td>
</tr>
<tr>
<td>pbes2lts -grey</td>
<td>LTSMIN</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>pbes2lts-grey --grey --always-split</td>
</tr>
<tr>
<td>pbes2lts -cache</td>
<td>LTSMIN</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>pbes2lts-grey -rgs -c --always-split</td>
</tr>
<tr>
<td>pbes2lts-mpi-*</td>
<td>LTSMIN</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>pbes2lts-mpi -rgs -c --always-split</td>
</tr>
<tr>
<td>pbes-reach</td>
<td>LTSMIN</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>pbes-reach --order=chain-prev\n                        --saturation=sat-like\n                        --save-levels -rgs\n                        --always-split</td>
</tr>
</tbody>
</table>

It is indicated whether transition groups, caching, distributed generation or symbolic generation are available. pbes2bes and pbespgsolve from the mCRL2 toolset are similar in functionality, but different in implementation. For pbespgsolve the -g option means only generating the parity game without solving. For the LTSMIN tools pbes2lts-* and pbes-reach the option -rgs enables regrouping, -c enables caching, and --black disables the use of transition groups. pbes-reach uses the sat-like saturation strategy.

The experiments were performed on a cluster of 10 machines with each two quad-core Intel Xeon E5520 CPUs @ 2.27 GHz (with 2 hyperthreads per core) and 24GB memory. Every tool was given a 20 GB memory limit and a 10 ks time limit. Elapsed time and memory usage have been measured by the tool memtime. The experiments were executed using Linux 2.6.34, mCRL2 svn rev. 10785 and for LTSMIN the git rev. after commit 4d11bc20 in the experimental ‘next’ branch. The tools were built using GCC 4.4.1. Open MPI 1.4.3 was used for the distributed tool.
### Table 1: Time performance in seconds. ‘T’ indicates a timeout, ‘M’ out of memory.

<table>
<thead>
<tr>
<th>Equation system</th>
<th># States</th>
<th>pbes2bes</th>
<th>pbespgsolve</th>
<th>pbes2lts -black</th>
<th>pbes2lts -grey</th>
<th>pbes2lts -cache</th>
<th>pbes2lts-mpi-1</th>
<th>pbes2lts-mpi-4</th>
<th>pbes2lts-mpi-8</th>
<th>pbes-reach</th>
</tr>
</thead>
<tbody>
<tr>
<td>swp.nodeadlock</td>
<td>1,862</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>swp.evt_send</td>
<td>33,554</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>1394.nodeadlock</td>
<td>173,101</td>
<td>199</td>
<td>202</td>
<td>231</td>
<td>1,387</td>
<td>120</td>
<td>125</td>
<td>56</td>
<td>73</td>
<td>114</td>
</tr>
<tr>
<td>sokoban.372.always_push</td>
<td>834,397</td>
<td>69</td>
<td>78</td>
<td>258</td>
<td>T</td>
<td>403</td>
<td>419</td>
<td>182</td>
<td>62</td>
<td>31</td>
</tr>
<tr>
<td>buffer.7.nodeadlock</td>
<td>823,545</td>
<td>32</td>
<td>33</td>
<td>48</td>
<td>76</td>
<td>13</td>
<td>16</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>buffer.7.evt_send</td>
<td>2,466,257</td>
<td>111</td>
<td>107</td>
<td>157</td>
<td>266</td>
<td>22</td>
<td>27</td>
<td>13</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>buffer.8.nodeadlock</td>
<td>5,764,803</td>
<td>235</td>
<td>237</td>
<td>357</td>
<td>594</td>
<td>82</td>
<td>93</td>
<td>31</td>
<td>20</td>
<td>37</td>
</tr>
<tr>
<td>buffer.8.evt_send</td>
<td>17,281,283</td>
<td>820</td>
<td>859</td>
<td>1,256</td>
<td>2,171</td>
<td>158</td>
<td>191</td>
<td>71</td>
<td>67</td>
<td>42</td>
</tr>
<tr>
<td>buffer.9.nodeadlock</td>
<td>40,353,607</td>
<td>1,059</td>
<td>965</td>
<td>354</td>
<td>91</td>
<td>89</td>
<td>89</td>
<td>36</td>
<td>27</td>
<td>49</td>
</tr>
<tr>
<td>buffer.9.evt_send</td>
<td>121,021,455</td>
<td>M</td>
<td>M</td>
<td>T</td>
<td>T</td>
<td>1,172</td>
<td>1,448</td>
<td>520</td>
<td>306</td>
<td>282</td>
</tr>
<tr>
<td>wheel_sector.nodeadlock</td>
<td>4,897,760</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>2,337</td>
<td>2,368</td>
<td>828</td>
<td>939</td>
<td>1,904</td>
</tr>
</tbody>
</table>

### Table 2: Memory usage in MB. ‘T’ indicates a timeout, ‘M’ out of memory.

<table>
<thead>
<tr>
<th>Equation system</th>
<th># States</th>
<th>pbes2bes</th>
<th>pbespgsolve</th>
<th>pbes2lts -black</th>
<th>pbes2lts -grey</th>
<th>pbes2lts -cache</th>
<th>pbes2lts-mpi-1</th>
<th>pbes2lts-mpi-4</th>
<th>pbes2lts-mpi-8</th>
<th>pbes-reach</th>
</tr>
</thead>
<tbody>
<tr>
<td>swp.nodeadlock</td>
<td>1,862</td>
<td>12</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>16</td>
<td>13</td>
<td>15</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>swp.evt_send</td>
<td>33,554</td>
<td>58</td>
<td>29</td>
<td>20</td>
<td>20</td>
<td>18</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>1394.nodeadlock</td>
<td>173,101</td>
<td>227</td>
<td>168</td>
<td>31</td>
<td>30</td>
<td>89</td>
<td>86</td>
<td>60</td>
<td>50</td>
<td>57</td>
</tr>
<tr>
<td>sokoban.372.always_push</td>
<td>834,397</td>
<td>1,187</td>
<td>768</td>
<td>34</td>
<td>T</td>
<td>220</td>
<td>217</td>
<td>69</td>
<td>45</td>
<td>47</td>
</tr>
<tr>
<td>buffer.7.nodeadlock</td>
<td>823,545</td>
<td>965</td>
<td>354</td>
<td>32</td>
<td>32</td>
<td>91</td>
<td>89</td>
<td>36</td>
<td>27</td>
<td>49</td>
</tr>
<tr>
<td>buffer.7.evt_send</td>
<td>2,466,257</td>
<td>3,340</td>
<td>1,215</td>
<td>63</td>
<td>64</td>
<td>181</td>
<td>179</td>
<td>67</td>
<td>43</td>
<td>49</td>
</tr>
<tr>
<td>buffer.8.nodeadlock</td>
<td>5,764,803</td>
<td>7,179</td>
<td>2,579</td>
<td>117</td>
<td>117</td>
<td>528</td>
<td>525</td>
<td>145</td>
<td>81</td>
<td>49</td>
</tr>
<tr>
<td>buffer.8.evt_send</td>
<td>17,281,283</td>
<td>18,136</td>
<td>9,056</td>
<td>345</td>
<td>345</td>
<td>1,155</td>
<td>1,152</td>
<td>377</td>
<td>204</td>
<td>49</td>
</tr>
<tr>
<td>buffer.9.nodeadlock</td>
<td>40,353,607</td>
<td>18,451</td>
<td>M</td>
<td>737</td>
<td>737</td>
<td>4,129</td>
<td>4,127</td>
<td>1,048</td>
<td>538</td>
<td>49</td>
</tr>
<tr>
<td>buffer.9.evt_send</td>
<td>121,021,455</td>
<td>M</td>
<td>M</td>
<td>T</td>
<td>T</td>
<td>9,209</td>
<td>9,206</td>
<td>3,003</td>
<td>1,487</td>
<td>49</td>
</tr>
<tr>
<td>wheel_sector.nodeadlock</td>
<td>4,897,760</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1,288</td>
<td>1,285</td>
<td>389</td>
<td>238</td>
<td>90</td>
</tr>
</tbody>
</table>
5.2 Results

Results are in Tables 1 (time performance in seconds) and 2 (memory usage in MB). For the MPI tool, the values are the maximum for the workers. The ‘T’ indicates a timeout, the ‘M’ indicates an Out of Memory error. We can make the following observations.

From the results we see that pbes2bes and pbespgsolve from the mCRL2 toolset perform better than pbes2lts -black, the LTSMIN based tool without any optimisation. The memory performance of the LTSMIN tool however is much better, even over 25 times better in the case of buffer.8.evt_send.

Looking at pbes2lts -grey we observe that only splitting into transition groups without any optimisations has a negative impact on the performance, especially in the case of 1394.nodeadlock.

The LTSMIN tools have a relatively bad performance for the Sokoban puzzle, because of the structure of always_push: either “we can do a push now” or “we move and take a recursive step”. If this formula is evaluated as a whole on a state where we can do a push, the first part will immediately evaluate to true and the formula as well, without taking the recursive step. When the formula is split into transition groups, then both parts may be evaluated independently. Although the second part is not needed, such on-the-fly solving optimisations are not available in the PBES language module yet when transition groups are enabled. This causes LTSMIN to generate a state space of 10,992,856 states (instead of 834,397), but still the symbolic tool of LTSMIN, pbes-reach, is the fastest.

Transition caching pays off for many systems. Compared to the mCRL2 tools, the speedup is between 1.8 and 5.1 for the sequential buffers and for wheel_sector the instantiation is completed within the timebound. The distributed tool does not scale well. The speedup with 8 workers compared to 1 worker is 6.8 for the Sokoban puzzle, but does not exceed 4.7 for the sequential buffers, and is only 2.5 for the wheel_sector case. In the wheel_sector and 1394 cases the execution time for 8 workers is even worse than with 4 workers, indicating that there is a limit to the number of workers that result in a further speedup.

The symbolic tool performs best of all sequential tools in all cases. The tool is up to 19.5 times faster than the fastest tool from the mCRL2 toolset (in the buffer.8.evt_send case). And for some cases LTSMIN could finish within memory and time bounds, whereas the mCRL2 tools could not. Memory usage of pbes-reach is slightly worse in the smallest cases, but up to more than 180 times better than the mCRL2 tools for the other cases.

6 Conclusions

We have defined PPG as normal form for PBESs and a transformation to PPG, making the instantiation to parity games more straightforward. We implemented a PBES language module for LTSMIN. As a result, the high-performance capabilities for state space generation become available for parity game generation. We demonstrated this for distributed state space generation and for symbolic state space generation.

Experimental comparison to existing tools shows good results. The LTSMIN tools reduce memory usage enormously. Transition caching, distributed computation and the symbolic tool speed up the instantiation in all reported cases. However, the distributed tool does not scale well. For all reported cases, the symbolic LTSMIN tool performed the best, with up to 19 times speedup and up to more than 180 times lower memory usage compared to the mCRL2 tools.

We intend to extend the tool with optimisations, such as on-the-fly minimisation and solving, i.e., while generating the parity game (possibly also distributed). Furthermore, the symbolic tool generates a BDD representation of the parity game, which asks for solvers that can deal with such symbolic parity games similar to the tool by [1].
Acknowledgments. We are grateful to Tim Willemse, Jeroen Keiren and Wieger Wesselink for their support on the mCRL2 toolset.

References


