Generating and Solving Symbolic Parity Games

Gijs Kant*
kant@cs.utwente.nl

Jaco van de Pol
vdpol@cs.utwente.nl

Formal Methods & Tools
University of Twente, Enschede, The Netherlands

We present a new tool for verification of modal μ-calculus formulae for process specifications, based on symbolic parity games. It enhances an existing method, that first encodes the problem to a Parameterised Boolean Equation System (PBES) and then instantiates the PBES to a parity game. We improved the translation from specification to PBES to preserve the structure of the specification in the PBES, we extended LTSmin to instantiate PBESs to symbolic parity games, and implemented the recursive parity game solving algorithm by Zielonka for symbolic parity games. We use Multi-valued Decision Diagrams (MDDs) to represent sets and relations, thus enabling the tools to deal with very large systems. The transition relation is partitioned based on the structure of the specification, which allows for efficient manipulation of the MDDs. We performed two case studies on modular specifications, that demonstrate that the new method has better time and memory performance than existing PBES based tools and can be faster (but slightly less memory efficient) than the symbolic model checker NuSMV.

1 Introduction

When verifying large systems or modelling large games with, say, billions or even trillions of states, datastructures are needed that can represent such large numbers of states, e.g., Multi-valued Decision Diagrams (MDDs). We have developed a tool that enables verification of modal μ-calculus formulae for process algebraic specifications using MDDs. The tool, called `pbes2lts-sym`, is now part of LTSmin, a toolset for high performance verification that is language-independent [3][1]. The tool can deal with very large state spaces, provided that the transition relation of the modelled system can be partitioned into relatively independent groups. LTSmin is used in several application domains, including verification of railway safety systems. In this paper a case study is included where the presented method is applied to analysis of the control software used in the Large Hadron Collider at CERN.

An established method for verification of μ-calculus is translation of the problem to a Parity Game (PG) and then solving the game. Our starting point is a Linear Process Specification (LPS), specified in the process algebraic language mCRL2. A possible translation of the verification problem to a parity game is shown as the dotted route through Fig. 1: first instantiating the LPS to a Labelled Transition System (LTS) and then translating satisfaction of a formula by the LTS to a parity game. We follow the more symbolic route, taken in the mCRL2 toolset.[2] This verification approach corresponds to the solid line route in Fig. 1. The problem is first translated to a Parameterised Boolean Equation System (PBES) [12], a sequence of Boolean fixpoint equations with data variables, of which the solution is **true** if and only if the specification satisfies the formula. The PBES is instantiated to a parity game with the same

---

[*] Gijs Kant is sponsored by the NWO under grant number 612.000.937 (VOCHS).

[1] Available from `http://fmt.cs.utwente.nl/tools/ltsmin` (Open Source).

[2] See `http://mcrl2.org`.

Submitted to:
GRAPHITE 2014

© G. Kant & J.C. van de Pol
This work is licensed under the
Creative Commons Attribution License.

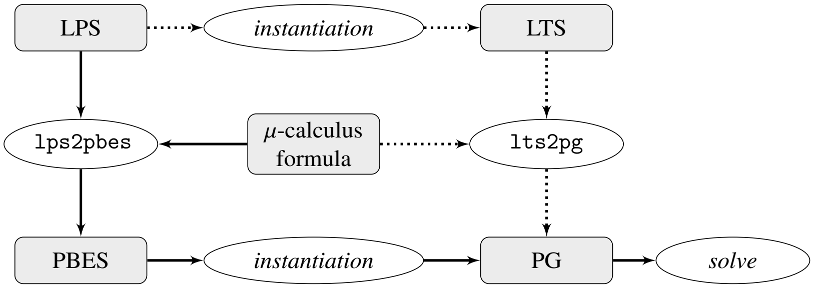

Figure 1: Overview.

solution. An advantage of the second approach is that the intermediate step of generating the LTS, which can be rather large, is not needed. Furthermore, property-specific reduction techniques can be applied to the PBES, which would result in a smaller parity game.

Parity games are two player games, represented by a game graph where the nodes represent the states of the game and the edges the possible moves, and each node belongs to one of the players (representing 'and' and 'or') and has a priority. Solving a parity game (locally) means determining if a winning strategy from the initial state exists for one of the players. The concepts that are used will be briefly explained in Section 2.

When the system is very large, also the parity game that encodes satisfaction of a formula for that system can become very large. Therefore we need efficient data structures and algorithms to generate and solve the game.

In earlier work [14], we presented an early version of the `pbes2lts-sym` tool for generating parity games from PBESs. We use symbolic parity games, in which MDDs are used to represent sets of states and the relations encoding the moves, partitioned in transition groups. Other existing tools for solving PBESs, available in the mCRL2 toolset, use an explicit state representation, which severely limits the size of system that can be verified. The `pbes2lts-sym` tool is based on technology for generating symbolic state spaces, as is the tool `lps2lts-sym` for generating the state space for an LPS, which is described extensively in [2]. In `lps2lts-sym`, states are represented as vectors of values and are stored in an MDD. In order to be efficient, it is required that the specification is modular, i.e., that the transition relation can be partitioned into transition groups and that each of these groups depend on and influence only a small part of the state vector. This locality of transitions is expressed in a dependency matrix. When this matrix is sparse, the relations can be represented very compactly and applied to sets of states efficiently. Because of that, the structure of a system is more important than its size (we consider locality of transitions to be a structural property op the system). For `pbes2lts-sym`, the structure of the PBES is equally important. The generation of symbolic parity games is described in Section 4.

In the existing translation from specification to PBES, as described in [11] and available in the mCRL2 toolset, the structure of the process specification is not explicitly visible in the generated PBES. One equation is generated per propositional variable in the formula, disregarding the structure of the system. This makes it impossible to choose a good partition of the transition relation for generating a parity game. In the previous version of `pbes2lts-sym`, we tried to guess a partition by splitting conjunctive equations in conjuncts and disjunctive equations in disjuncts. That way symbolic data structures could be used, but far from optimal since it disregards the original process structure, hindering efficient MDD manipulation. The importance of the partitioned transition relation in symbolic verification is well known in the literature and the basis of the success of tools like NuSMV [7] and techniques like saturation [6].

In this article we propose a modified translation from LPS to PBES to preserve the structure such that the `pbes2lts-sym` tool can base the partitioning on the structure of the specification and can benefit in the same way from a sparse dependency matrix as the symbolic tools for generating LTSs. The new translation to PBESs is presented in Section 3.

The previous version of `pbes2lts-sym` did only generate parity games, not solve them. We implemented the recursive algorithm of Zielonka [17] (similar to [1]) for symbolic parity games. The solver is decribed in Section 5. The combination of generation and solving is now available as part of the LTSMIN toolset.

We performed two large case studies, presented in Section 6, solving the Connect Four game and verification of control software in use at the CMS particle detector at CERN.[3] Both are challenging problems for model checking tools. The new tools show significant improvement of time and memory performance over existing tools in the mCRL2 toolset and `pbes2lts-sym` with the previous translation from LPS to PBES. For the Connect Four game we also compared to NuSMV, where our tool is faster, but NuSMV has better memory performance.

2 Background

In this section we will briefly explain Linear Process Specifications, Modal μ-calculus, Parameterised Boolean Equation Systems and Parity Games. More extensive descriptions can be found in, e.g., [10] (linear processses), [12] (PBESs), [4] and [9] (parity games).

2.1 Process algebra

The mCRL2 language is a process algebraic modelling language with algebraic data types. Several analysis techniques are available in the mCRL2 toolset, such as simulation, visualisation and model checking. For analysis purposes a specification is *linearised* to Linear Process Specification (LPS) format. Linearisation removes concatenation of actions, parallel composition, hiding, etc. An LPS consists of a single process with *summands* of the form “$\sum_{d:\,D}$ guard $\rightarrow$ action . next state” and an initial state of the process. The LPS gives rise to a Labelled Transition System (LTS). The summands model the nondeterministic choice in the system.

Definition 2.1 (Linear Process)**.** A *linear process* has the following structure:

$$\textbf{proc}\ P(x_p\colon D_p) = \sum_{i\in I}\sum_{y\colon E_i} c_i(x_p,y) \rightarrow a_i(f_i(x_p,y))\ .\ P(g_i(x_p,y))$$

The specification consists of m summands $i \in I$ with $I = \{1,\dots,m\}$. Each summand may sum over a data sort E_i, has a guard c_i, a parameterised action a_i, that is enabled if the guard is satisfied, and the specification of the behaviour after having executed the action, specified by the recursive definition $P(g_i(x_p,y))$. In examples the +-operator will be used for combining summands instead of the sum notation.

Example 2.2. As an example we give a model of a buffer with two cells:

$$\begin{aligned}
&\textbf{proc}\ \ \mathsf{Buffer}(q\colon \mathrm{List}(D)) = \\
&\qquad\qquad \sum_{d\colon D} (\#q < 2) \rightarrow \mathrm{read}(d)\ .\ \mathsf{Buffer}(q \triangleleft d) \\
&\qquad\qquad + (q \neq [\,]) \rightarrow \mathrm{send}(\mathrm{head}(q))\ .\ \mathsf{Buffer}(\mathrm{tail}(q)); \\
&\textbf{init}\ \ \ \mathsf{Buffer}([\,]);
\end{aligned}$$

---

[3] Instructions on how to install and use the tools and the files used in the case studies are available at `http://www.cs.utwente.nl/~kant/graphite2014/`.

The process Buffer has a data parameter q, which is a list of elements from D, modelling the contents of the buffer. The size of q is denoted $\#q$. The process can either read a value d and proceed with $\mathsf{Buffer}(q \triangleleft d)$ (the same process, but with d appended to q), or send the first element of q and proceed with $\mathsf{Buffer}(\mathrm{tail}(q))$. The system is initialised to the Buffer process with $q = [\,]$, the empty list, as parameter.

2.2 First order modal μ-calculus

The first order modal μ-calculus is Hennesy-Milner logic extended with fixpoint operators, quantifiers and data parameters. Formulae are defined by the grammar:

$$\phi ::= b \mid \neg b \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \mid \mathsf{Q}d\colon D\,.\,\phi \mid \mathsf{Z}(e) \mid \sigma\mathsf{Z}(x\colon D := d)\,.\,\phi$$

where b is a boolean data expression, $\mathsf{Q} \in \{\forall, \exists\}$, $\sigma \in \{\mu, \nu\}$ is a minimal (μ) or maximal (ν) fixpoint operator, with the restriction that each propositional variable Z occurs positively in φ in an equation $\sigma\mathsf{Z}(x\colon D := d)\,.\,\varphi$, i.e., within the scope of an even number of negations. α is some language for specifying predicates on actions. The class of first order modal μ-calculus formulae is denoted $\mathcal{M}$. The semantics of formulae interpreted over LTSs is presented in, e.g., [11].

2.3 Parameterised Boolean Equation Systems

Satisfaction of a formula by a linear process specification is first translated to a Parameterised Boolean Equation System (PBES), a system of first order boolean equations. We propose a new translation in Section 3; here we first define what a PBES is.

Definition 2.3. *Predicate formulae* ξ are defined by the following grammar:

$$\xi ::= b \mid \mathsf{X}(\vec{e}) \mid \neg\xi \mid \xi \oplus \xi \mid \mathsf{Q}d\colon D\,.\,\xi$$

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$, $\mathsf{Q} \in \{\forall, \exists\}$, b is a data term of sort Bool, $\mathsf{X} \in \mathcal{X}$ 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 2.4. A *First Order Boolean Equation* is an equation of the form: $\sigma\mathsf{X}(\vec{d}\colon D) = \xi$ where $\sigma \in \{\mu, \nu\}$ is a fixpoint operator, $\vec{d}$ is a vector of data variables of sort D, and ξ is a predicate formula. The class of first order boolean equations is denoted $\mathcal{E}$.

Definition 2.5. A *Parameterised Boolean Equation System (PBES)* is a sequence of First Order Boolean Equations: $\mathcal{S} = (\sigma_1\mathsf{X}_1(\vec{d_1}\colon D_1) = \xi_1)\ \ldots\ (\sigma_n\mathsf{X}_n(\vec{d_n}\colon D_n) = \xi_n)$

The semantics and solution of PBESs are described in, e.g., [12]. The order of the equations does matter.

2.4 Parity games

A *parity game* is a game between two players, player **0** (also called ∃loise or player *even*) and player **1** (also called ∀belard 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 [4] and [15].

Definition 2.6 (Parity Game). A *parity game* is a graph $\mathcal{G} = \langle V, E, V_{\mathbf{0}}, V_{\mathbf{1}}, v_I, \Omega\rangle$, with

- V the set of vertices (nodes or places or states);

- $E\colon V \times V$ the set of transitions;
- $V_p \subseteq V$ the set of places owned by player p, for $p \in \{\mathbf{0},\mathbf{1}\}$, with $V_\mathbf{0} \cup V_\mathbf{1} = V$ and $V_\mathbf{0} \cap V_\mathbf{1} = \emptyset$;
- $v_I \in V$ the initial state of the game;
- $\Omega\colon V \to \mathbb{N}$ assigns a priority $\Omega(v)$ to each vertex $v \in V$.

The vertices in the graph represent the instantiated variables from the equation system. The edges represent possible moves of the token (initially placed on v_I) and encode dependencies between variables. In the parity game, player **0** owns the vertices that represent disjunctions, player **1** the vertices that represent conjunctions.

Player **0** is the winner of a play π if π is a finite play $v_0 v_1 \cdots v_r \in V^+$ and $v_r \in V_\mathbf{1}$ and no move is possible from v_r; or π is an infinite play and $\min(\mathrm{Inf}(\Omega(\pi)))$, the minimum of the priorities that occur infinitely often in π, is even. A (memoryless) *strategy* for player a is a function $f_a\colon V_a \to V$. A play $\pi = v_0 v_1 \cdots$ is *conform* to f_a if for every $v_i \in \pi,\ \ v_i \in V_a \Rightarrow v_{i+1} = f_a(v_i)$. 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**.

3 Translating modal μ-calculus and LPS to a modular PBES

In this section we describe the adapted version of the translation from first order modal μ-calculus formulae and LPSs to PBESs. The original translation for μCRL (of which mCRL2 is an extention) was published in [11]. It translated the satisfaction of a formula by an LPS to a system of equations that consists of one equation per propositional variable in the formula. In order to be able to generate a parity game from the PBES efficiently, the equations have to be partitioned in relatively independent parts. In this section we present a translation that ensures that the structure of the equation system reflects the structure of the specification, for the modal operators in the formula. This allows to partition the equations based on the structure of the input specification. To achieve that, we changed the original translation for the modal operators $\langle\alpha\rangle$ and $[\alpha]$. In our translation, for these operators new equations are introduced for every summand in the specification.

The translation is defined for a fixed process specification P, as defined in Def. 2.1, and for formulae of the form $\varphi_0 = \sigma\mathsf{X}(x_f\colon D_f := d)\,.\,\varphi$, i.e., formulae with a fixpoint operator as outmost operator.

Satisfaction of the formula φ_0 by process P is defined as $\mathsf{T}(\varphi_0)$, with the function T as defined below.

Let $\mathcal{D}$ be the set of data variables. The function $\mathsf{T}\colon \mathcal{M} \to \mathcal{E}^*$ generates a sequence of first order boolean equations for a formula $\varphi \in \mathcal{M}$. The function T uses a function $\mathsf{RHS}\colon \mathcal{M} \times \mathcal{D}^* \times \{\nu,\mu\} \to \mathcal{F} \times \mathcal{E}^*$ (defined below) that produces the right hand side of the equation and a sequence of equations that are introduced to be used in the right hand side. $\mathsf{RHS}(\varphi,\vec{v},\varsigma)$ has a μ-calculus formula φ, a sequence of data variables $\vec{v}$ and a fixpoint operator ς as arguments. The latter two are needed for the newly introduced equations.

The function T is specified as follows for the fixpoint operator:

$$\mathsf{T}(\sigma\mathsf{X}(x_f\colon D_f := d)\,.\,\varphi) \overset{\mathrm{def}}{=} (\sigma\widetilde{\mathsf{X}}(\vec{v}) = \psi) \mathbin{+\!\!\!+} Z \mathbin{+\!\!\!+} \mathsf{T}(\varphi)$$

with $\langle\psi,Z\rangle = \mathsf{RHS}(\varphi,\vec{v},\sigma)$ and $\vec{v} = [x_f\colon D_f, x_p\colon D_p]$. The operator $\mathbin{+\!\!\!+}$ denotes concatenation of sequences. A new equation is produced with ψ as right hand side, which is the result of RHS applied to the formula. The resulting sequence of equations consists of the new equation together with Z, the equations generated by RHS, and the result of T applied to the remainder of the formula, $\mathsf{T}(\varphi)$.

Table 1: Definition of the RHS function that generates a PBES from a μ-calculus formula, defined for a process P as in Definition 2.1.

$$
\begin{array}{lll}
\mathsf{RHS}(b,\vec{v},\varsigma) & \overset{\text{def}}{=} \langle b,[\,]\rangle & \\
\mathsf{RHS}(\neg b,\vec{v},\varsigma) & \overset{\text{def}}{=} \langle \neg b,[\,]\rangle & \\
\mathsf{RHS}(\varphi_1 \land \varphi_2,\vec{v},\varsigma) & \overset{\text{def}}{=} \langle \psi_1 \land \psi_2, Z_1 \mathbin{+\!\!+} Z_2\rangle & \text{with } \langle \psi_i, Z_i\rangle = \mathsf{RHS}(\varphi_i,\vec{v},\varsigma) \text{ for } i \in \{1,2\}. \\
\mathsf{RHS}(\forall x\colon D \,.\, \varphi,\vec{v},\varsigma) & \overset{\text{def}}{=} \langle \forall x\colon D \,.\, \psi, Z\rangle & \text{with } \langle \psi, Z\rangle = \mathsf{RHS}(\varphi,\vec{v} \mathbin{+\!\!+} [x],\varsigma) \\
\mathsf{RHS}([\alpha]\varphi,\vec{v},\varsigma) & \overset{\text{def}}{=} \Big\langle \widetilde{\mathsf{Y}}(\vec{v}), & \\
 & \quad \langle\ \varsigma\widetilde{\mathsf{Y}}(\vec{v}) = \bigwedge_{i \in I} \widetilde{\mathsf{X}}_{\mathsf{i}}(\vec{v}), & \\
 & \qquad \varsigma\widetilde{\mathsf{X}}_1(\vec{v}) = \mathsf{ApplySummand}(1,\psi) & \\
 & \qquad \cdots, & \\
 & \qquad \varsigma\widetilde{\mathsf{X}}_{\mathsf{m}}(\vec{v}) = \mathsf{ApplySummand}(m,\psi) & \\
 & \qquad Z\ \rangle\ \Big\rangle & \\
 & \text{with } \langle \psi, Z\rangle = \mathsf{RHS}(\varphi,\vec{v},\varsigma) & \\
\mathsf{RHS}(\mathsf{X}(d),\vec{v},\varsigma) & \overset{\text{def}}{=} \langle \widetilde{\mathsf{X}}(d,x_p),[\,]\rangle & \\
\mathsf{RHS}(\sigma\mathsf{X}(x_f\colon D_f := d) \,.\, \varphi,\vec{v},\varsigma) & \overset{\text{def}}{=} \langle \widetilde{\mathsf{X}}(d,x_p),[\,]\rangle &
\end{array}
$$

For other operators, the function T is applied to the subformulae recursively, where the resulting sequences of equations are concatenated.

The function RHS generates the right hand sides of the equations as defined in Table 1 with the translation for the modal operator defined per summand i as:

$$\mathsf{ApplySummand}(i,\psi) = \forall_{y\colon E_i}(a_i(f_i(x_p,y)) \in [\![\alpha]\!] \land c_i(x_p,y)) \Rightarrow \psi[x_p := g_i(x_p,y)]$$

The cases for boolean conditions b, negation and conjunction are straightforward. For universal quantification the quantified variable is added to the list of parameters $\vec{v}$ for equations generated by RHS. For propositional variables and fixpoint subformulae the corresponding variable is used, where the notation $\widetilde{\mathsf{X}}$ is used to introduce a fresh variable that is guaranteed to be unique in the equation system.

For the modal operator $[\alpha]\varphi$, the RHS function generates a new equation for each summand and results in an expression that is a conjunction of the propositional variables that are the left hand sides of these equations. Every equation generated for a summand i for a formula $[\alpha]\varphi$, has a right hand side generated by $\mathsf{ApplySummand}(i,\psi)$, where c_i is the guard of summand i, g_i is the function that describes the transition to the next state, ψ is the result of applying RHS to the formula φ, and $a_i(f_i(x_p,y)) \in [\![\alpha]\!]$ means that the action of summand i satisfies the action formula α.

The cases for disjunction, existential quantification and possibility are similar to those for conjunction, universal quantification and necessity, respectively.

Example 3.1. For the two place buffer in Example 2.2, we want to verify that if a message d is read through action 'read', it will eventually be sent through action 'send', which is expressed by the formula:

$$\nu\mathsf{Y} \,.\, (\forall d\colon D \,.\, ([\mathrm{read}(d)](\mu\mathsf{X} \,.\, (\langle \mathbf{true}\rangle \mathbf{true} \land [\neg\mathrm{send}(d)]\,\mathsf{X})))) \land [\mathbf{true}]\,\mathsf{Y}$$

Applying the function T results in the equation system in Figure 2. The equation Y represents the $\nu\mathsf{Y}$ part of the formula (which is the whole formula), X represents the $\mu\mathsf{X}$ part. The system is initialised to the toplevel variable (Y) with the LPS parameters set to their initial value in the specification ($q = [\,]$).

$$
\begin{array}{lll}
\mathbf{pbes} & \nu\mathsf{Y}(q\colon \mathrm{List}(D)) & = (\forall d\colon D\,.\,(\#q < 2) \Rightarrow \mathsf{X}(q \triangleleft d, d)) \wedge \mathsf{Y}_1(q); \\
 & \nu\mathsf{Y}_1(q\colon \mathrm{List}(D)) & = \mathsf{Y}_{11}(q) \wedge \mathsf{Y}_{12}(q); \\
 & \nu\mathsf{Y}_{11}(q\colon \mathrm{List}(D)) & = (\forall d'\colon D\,.\,(\#q < 2) \Rightarrow \mathsf{Y}(q \triangleleft d')) \\
 & \nu\mathsf{Y}_{12}(q\colon \mathrm{List}(D)) & = ((q \neq [\,]) \Rightarrow \mathsf{Y}(\mathrm{tail}(q))); \\
 & \mu\mathsf{X}(q\colon \mathrm{List}(D), d\colon D) & = \mathsf{X}_1(q) \wedge \mathsf{X}_2(q,d); \\
 & \mu\mathsf{X}_1(q\colon \mathrm{List}(D)) & = \mathsf{X}_{11}(q) \vee \mathsf{X}_{12}(q); \\
 & \mu\mathsf{X}_{11}(q\colon \mathrm{List}(D)) & = (\#q < 2) \\
 & \mu\mathsf{X}_{12}(q\colon \mathrm{List}(D)) & = (q \neq [\,]) \\
 & \mu\mathsf{X}_2(q\colon \mathrm{List}(D), d\colon D) & = \mathsf{X}_{21}(q,d) \wedge \mathsf{X}_{22}(q,d); \\
 & \mu\mathsf{X}_{21}(q\colon \mathrm{List}(D), d\colon D) & = (\forall d'\colon D\,.\,(\#q < 2) \Rightarrow \mathsf{X}(q \triangleleft d', d)) \\
 & \mu\mathsf{X}_{22}(q\colon \mathrm{List}(D), d\colon D) & = (\mathrm{head}(q) = d \vee q = [\,] \vee \mathsf{X}(\mathrm{tail}(q), d)); \\
\mathbf{init} & \mathsf{Y}([\,]); &
\end{array}
$$

Figure 2: Example PBES.

The equation Y_1 represents the "[**true**] Y" part (at the end) of the formula, a conjunction of Y_{11} and Y_{12}, each representing a set of transitions from one of the summands of the LPS, followed by Y with the parameters updated to reflect the new state after the transitions. The equation X_1 represents "⟨**true**⟩ **true**" ('some action is enabled'), a disjunction of X_{11} and X_{12}, representing that of one of the two summands is enabled. The equation X_2 represents "$[\neg\mathrm{send}(d)]\,\mathsf{X}$" ('$\mathsf{X}$ should hold after every action other than send(d)'), a conjunction of X_{21} and X_{22}, representing the transitions that match the action formula $\neg$send(d).

Having separate equations representing the different summands now allows the tools to make a partition based on the summands, thus reflecting the structure of the LPS.

4 Generating Symbolic Parity Games

Instantiation to a parity game is similar to generating the reachable state space from a specification; both involve generating a large graph out of an abstract description. That is the reason we implemented instantiation of PBESs to parity games as an extension of the high performance verification toolset LTSMIN [3]. LTSMIN is modular in the sense that the core algorithms are separated from the input languages by using a generic interface in between. We extended LTSMIN with a PBES language module for generating symbolic parity games. We discuss instantiation in this section briefly. Details on the language module, the dependency matrix for PBESs and instantiation to parity games can be found in [14].

The result of instantiation is a symbolic parity game, in which MDDs are used to represent the set of reachable states of the game, for each player and for each priority the set of states owned by that player respectively with that priority, and the relations encoding the moves, partitioned in transition groups.

4.1 Parameterised Parity Games

For the instantiation of PBESs to Parity Games, we assume PBESs to be in a specific form: the *Parameterised Parity Game* (PPG), an equation system where every equation is either conjunctive or disjunctive. Not every PBES generated from an LPS and a formula (by the translation in Section 3) is a PPG, but any PBES can be transformed to an equivalent PPG by a transformation described in [14]. During instantiation every variable associated with a conjunctive expression will belong to player **1**, variables with disjunctive expressions will be owned by player **0**.

4.2 State Vectors and the Partitioned Transition Relation

Instantiated variables (predicate variables with concrete parameters) are the *states* in the generated parity game. A variable and its data parameters are stored in what in LTSmin is called a *state vector*. State vectors are used to encode states in LTSs and parity games, and are vectors of integers $\langle x_0, x_1, \cdots, x_K \rangle$ (other value types are stored in a database).

Logical dependencies as expressed by the equations are encoded as *transitions* in the generated parity game and are computed by a successor function Next.

During generation, LTSmin builds a symbolic transition relation E from the transitions that are computed by the language module. In LTSmin, E is a partitioned transition relation $E = E_1 \cup \ldots \cup E_M$, consisting of parts E_g which are called *transition groups*. The parts E_g are stored as MDDs; the composite relation E is not stored. For every newly encountered state its successors are computed for every transition group.

Applying the partitioned transition relation can be much more efficient than applying a monolithic transition relation if the partition is chosen well, as is well known in the literature (see, e.g., [5], [16]).

Computing the successors of a set of states V is then defined as $\textsc{Next}(V) = \bigcup_{1 \leq g \leq M} \textsc{Next}_g(V)$, i.e., iterating over the transition groups, where $\textsc{Next}_g(V)$ is the result of applying the relation E_g to V (and renaming the variables).

It is known that the order in which the different parts of the transition relation are applied often does matter for performance. For instance, saturation [6] is a technique for optimising the order of application to minimise the size of the intermediate decision diagrams. In LTSmin several of such techniques are available.

There are several ways of choosing transition groups for PPGs; we distinguish two. First, choosing entire equations to form a transition group (which we call *simple*). This approach is best when the equation system is generated by the translation in Section 3: if the summands of the original LPS are relatively independent, then also the equations in the PBES will be relatively independent. Second, splitting conjunctive equations in conjuncts and disjunctive equations in disjuncts (which we call *splitting*) is an option when such independence is not present in the equation system (e.g., when using the previous `lps2pbes` translation). However, as will be demonstrated in the experiments, splitting conjuncts and disjuncts does not per se result in a good partition.

4.3 Dependency matrix

On top of the partitioning of the transition relation into *transition groups*, we use a *dependency matrix* to store information about the *dependence* of transition groups on the *parts* x_i of the state vector. A group g is *dependent* on part i if the variable that is stored in slot i is read or changed by the expression of group g. Independence is also referred to as *locality*: when the dependency matrix is sparse, the effect of transition groups is relatively local. A detailed description is in [14], we will explain it here using an example.

Example 4.1. Suppose we have an LPS that models the Tic Tac Toe game as a process with parameters $b_1 \ldots b_9$ to encode the board configuration and parameter p to encode whose turn it is. Suppose that the LPS has separate summands for every position on the board, i.e., a summand for placing a piece on position b_1, one for b_2, etc. We want to check the property that player X has a winning strategy: $\mu \mathsf{Z} \,.\, [\mathrm{wins}(O)]\,\mathbf{false} \wedge \langle \mathrm{move}(X) \rangle (\langle \mathrm{wins}(X) \rangle\, \mathbf{true} \vee [\mathrm{move}(O)]\, \mathsf{Z})$, which for clarity of the example we present as a modal equation system:

$$\begin{aligned} &\mu \mathsf{Z} = \mathsf{A} \wedge \mathsf{B} && \mu \mathsf{B} = \langle \mathrm{move}(X) \rangle\, \mathsf{C} \\ &\mu \mathsf{A} = [\mathrm{wins}(O)]\, \mathbf{false} && \mu \mathsf{C} = \langle \mathrm{wins}(X) \rangle\, \mathbf{true} \vee [\mathrm{move}(O)]\, \mathsf{Z} \ . \end{aligned}$$

The resulting PBES for this property is:

$$\begin{aligned} \mathbf{pbes}\ &\mu \mathsf{Z}(b_1,\ldots,b_9,p) &&= \mathsf{A}(b_1,\ldots,b_9,p) \wedge \mathsf{B}(b_1,\ldots,b_9,p); \\ &\mu \mathsf{A}(b_1,\ldots,b_9,p) &&= \mathsf{A}_1(b_1,\ldots,b_9,p) \wedge \mathsf{A}_2(b_1,\ldots,b_9,p) \wedge \cdots \\ &\ldots \\ &\mu \mathsf{B}(b_1,\ldots,b_9,p) &&= \mathsf{B}_1(b_1,\ldots,b_9,p) \vee \mathsf{B}_2(b_1,\ldots,b_9,p) \vee \cdots \\ &\mu \mathsf{B}_1(b_1,\ldots,b_9,p) &&= (b_1 = -) \wedge \mathsf{C}(p,b_2,\ldots,b_9,\mathit{Opponent}(p)) \\ &\ldots \\ &\mu \mathsf{B}_9(b_1,\ldots,b_9,p) &&= (b_9 = -) \wedge \mathsf{C}(b_1,\ldots,b_8,p,\mathit{Opponent}(p)) \\ &\mu \mathsf{C}(b_1,\ldots,b_9,p) &&= \ldots \\ &\ldots \\ \mathbf{init}\ &\mathsf{Z}(-,-,-,-,-,-,-,-,-,\mathrm{X}); \end{aligned}$$

Equation A is a conjunction of the equations $\mathsf{A_i}$ for $1 \leq i \leq 9$, where each equation $\mathsf{A_i}$ means that $[\mathrm{wins}(O)]\,\mathbf{false}$ holds for summand i in the LPS; in other words, that there is no action wins(O) enabled in that summand. Equation B is a disjunction of the equations $\mathsf{B_i}$ for $1 \leq i \leq 9$, where each equation $\mathsf{B_i}$ means that $\langle \mathrm{move}(X) \rangle\, \mathsf{C}$ holds for summand i; in other words, that there is an action move(X) enabled, which is true if $b_i = -$, and that afterwards C holds for the state resulting from the action.

Each of B_1 to B_9 represents a single move on the board, and is computed by a single transition group touching only a small number of parameters.

The resulting dependency matrix is in Figure 3. For the transition groups B_1 to B_9, repectively the board parameters b_1 to b_9 are marked as dependent (+). The group of equation Z only changes the predicate variable Var and none of the parameters. This way the matrix is very sparse and transitions can be encoded efficiently.

If a transition group g is only dependent on, for instance, parameters 1 and 3 (as for B_2 in the example), then a transition $\langle \mathbf{0},0,\mathbf{1},1,1,\ldots\rangle \rightarrow_g \langle \mathbf{2},0,\mathbf{2},1,1,\ldots\rangle$ is simply stored as a vector of tuples of old and new values for dependent parameters: $\langle\langle 0,2\rangle,\langle 1,2\rangle\rangle$ Also, once a state has been visited with values $\langle 0,2\rangle$ for parameters 1 and 3, for future fresh states with the same values for these parameters, transitions for group g do not have to be computed for that state again. The MDDs used to store the partitioned transition relation only use these shorter vectors of integers, allowing for a compact representation of the transition relation.

g	Var	b_1	b_2	b_3	$\ldots$	b_9	p
Z	+	−	−	−		−	−
A	+	−	−	−		−	−
$\ldots$							
B	+	−	−	−		−	−
B_1	+	+	−	−		−	+
B_2	+	−	+	−		−	+
$\ldots$							
B_9	+	−	−	−		+	+
C	+	−	−	−		−	−
$\ldots$							

Figure 3: Dependency matrix for the Tic Tac Toe game.

5 Symbolic Parity Game Solver

We implemented the recursive algorithm for solving parity games by Zielonka [17] for symbolic parity games. The recursive algorithm is widely used in practice and easy to implement symbolically. Although its worst-case complexity is worse than some other algorithms, its performance is very good in practice [8] (at least for explicit representations of the game). Our solver is similar to the symbolic parity game solver by Bakera et al. [1]. Our own implementation allows us to reuse the partitioned transition relation from the instantiation tool directly.

The algorithm returns the set of winning states for player **1** and the set of winning states for player **0** as MDDs. The algorithm makes heavy use of the successor function Next and predecessor function Prev, which use the partitioned transition relation in the tool. However, in the solver currently available in `pbes2lts-sym` no saturation or similar techniques are used; the relations of transition groups are always applied in the order $1..M$.

6 Experiments

We performed experiments to compare our new tool to existing methods for solving PBESs (available in the mCRL2 toolset), previous versions of our tool and NuSMV 2 [7].

As benchmarks we verified properties for two models. First, we created a model of the well known *Connect Four* game (`four`) in both mCRL2 and NuSMV with different board sizes (the original is 7×6). We verified whether player Yellow has a winning strategy: $\mu \mathsf{X} \,.\, [\text{wins}(\mathit{Red})]\, \mathbf{false} \wedge \langle \text{move} \rangle\, (\langle \text{wins}(\mathit{Yellow}) \rangle\, \mathbf{true} \vee [\text{move}]\, \mathsf{X})$. For NuSMV, we used an SMV model and a CTL property equivalent to the μ-calculus property: $\mathbf{EX}(\text{yellowwins} \vee \mathbf{AX}(\neg\text{redwins} \wedge \mathbf{EX}(\text{yellowwins} \vee \mathbf{AX}(\ldots))))$.

Second, we verified several properties for *Finite State Machines* (FSMs) that are used in the Compact Muon Solenoid (CMS) detector, part of the Large Hadron Collider (LHC) at CERN. These state machines are used to control all the components of the detector, which are organised in a hierarchical manner. Components send commands to their children, which send status updates to their parent, asynchronously. In the experiments, we used the `wheel` subsystem, consisting of 8 FSMs, which we checked for four properties: absence of deadlocks (`nodeadlock`), absence of intermediate states in the *when* phase (`absence`), `progress` and `responsiveness`. The FSMs, the translation from FSMs to mCRL2, and the properties have been reported in [13]. We did not compare to NuSMV for this model.

The mCRL2 models are translated to PBESs using the function T, described in Section 3, which preserves the structure of the LPS (`structured`), and using the unstructured earlier version of the translation (`unstructured`). Both are available in the mCRL2 toolset. The structured version can be used by passing the `-s` option to the `lps2pbes` tool. We compared three tool combinations[4]:

- `pbes2bool`, one of the explicit state PBES solvers in the mCRL2 toolset, which instantiates to a Boolean Equation System (BES) and solves the BES using approximation. We used the unstructured `lps2pbes` as the tool performed better with that translation.
- The LTSmin toolset – in particular the tool `pbes2lts-sym` in combination with our new symbolic parity game solver `spgsolver`. We compared three combinations of `lps2pbes` translations and transition partitioning (see Section 4.2): `simple`: unstructured `lps2pbes` with one group per equation; `split`: unstructured `lps2pbes` with the equations split into conjuncts/disjuncts; and `structured`: structured `lps2pbes` with one group per equation.
- NuSMV 2.5.4 with the `-dynamic` option to enable dynamic reordering of variables, which appeared to give better performance.

The experiments were performed on a machine with two quad-core Intel Xeon E5520 CPUs @ 2.27 GHz and 24GB memory. Every tool was given a 20 GB memory limit and a 24 h time limit. We report

---

[4] Details on versions of the tools and options passed to the tools can be found on `http://www.cs.utwente.nl/~kant/lpar2013/`.

Table 2: Experimental results for ConnectFour and the CERN case study. Time is measured in seconds, memory usage in multiples of 1,000 KiB. ‘gen’ indicates time and memory used for generating a parity game, ‘solve’ solving and ‘total’ and ‘max’ indicate the total time and maximum memory used in all steps combined.

			MDD	Trans.		Time (s)			Memory (×1,000 KiB)	
System	Tool	#States	nodes	nodes	gen	solve	total	gen	solve	max
`four.5x4`	NuSMV						74			67
	`pbes2bool`	$1.8 \cdot 10^6$					118			1,678
	LTSmin `simple`	$1.3 \cdot 10^7$	$5.4 \cdot 10^5$	$2.0 \cdot 10^6$	1,136	171	1,307	215	298	298
	LTSmin `split`	$2.1 \cdot 10^7$	$8.5 \cdot 10^4$	$3.1 \cdot 10^3$	19	84	103	52	118	118
	LTSmin `struct`	$1.3 \cdot 10^7$	$4.5 \cdot 10^4$	$6.6 \cdot 10^2$	7.3	37	44	49	76	76
`four.6x4`	NuSMV						2,657			432
	`pbes2bool`						–			>20,000
	LTSmin `split`	$7.1 \cdot 10^8$	$2.8 \cdot 10^5$	$4.3 \cdot 10^3$	94	1,435	1,529	65	1,242	1,242
	LTSmin `struct`	$3.8 \cdot 10^8$	$1.5 \cdot 10^5$	$8.9 \cdot 10^2$	13	565	586	50	770	770
`four.6x5`	NuSMV						>86,400			–
	LTSmin `split`	$4.6 \cdot 10^{10}$	$1.8 \cdot 10^6$	$6.3 \cdot 10^3$	949	37,697	38,646	328	13,706	13,706
	LTSmin `struct`	$2.6 \cdot 10^{10}$	$1.6 \cdot 10^6$	$1.3 \cdot 10^3$	574	17,139	17,713	324	13,706	13,706
`four.7x5`	NuSMV						>86,400			–
	LTSmin `split`	$3.1 \cdot 10^{12}$	$5.1 \cdot 10^6$	$8.4 \cdot 10^3$	6,988	–	–	1,300	>20,000	>20,000
	LTSmin `struct`	$1.6 \cdot 10^{12}$	$3.3 \cdot 10^6$	$1.7 \cdot 10^3$	1,122	–	–	520	>20,000	>20,000
`four.6x6`	LTSmin `struct`	$1.6 \cdot 10^{12}$	$2.2 \cdot 10^7$	$1.8 \cdot 10^3$	10,750	–	–	5,439	>20,000	>20,000
`four.7x6`	LTSmin `struct`	$2.0 \cdot 10^{14}$	$7.2 \cdot 10^7$	$2.2 \cdot 10^3$	81,116	>86,400	>86,400	14,340	>20,000	>20,000
`wheel nodeadlock`	`pbes2bool`	$4.6 \cdot 10^6$					2,190			9,894
	LTSmin `split`	$4.6 \cdot 10^6$	$2.7 \cdot 10^5$	$2.6 \cdot 10^6$	16,856	247	17,103	340	187	340
	LTSmin `struct`	$1.4 \cdot 10^7$	$3.1 \cdot 10^5$	$1.2 \cdot 10^4$	190	47	237	98	118	118
`wheel absence`	`pbes2bool`	$5.9 \cdot 10^6$					4,420			14,779
	LTSmin `split`	$2.0 \cdot 10^7$	$1.2 \cdot 10^6$	$1.1 \cdot 10^7$	28,769	477	29,246	889	770	889
	LTSmin `struct`	$2.4 \cdot 10^7$	$8.5 \cdot 10^5$	$1.3 \cdot 10^4$	1,142	172	1,314	215	118	215
`wheel progress`	`pbes2bool`						–			>20,000
	LTSmin `split`	$3.2 \cdot 10^7$	$1.4 \cdot 10^7$	$1.3 \cdot 10^8$	61,156	9,078	70,234	9,050	8,474	9,050
	LTSmin `struct`	$6.5 \cdot 10^7$	$3.1 \cdot 10^5$	$1.2 \cdot 10^4$	1,266	2,471	3,737	142	299	299

the number of states of the generated symbolic parity game, the number of MDD nodes used to store the set of states and the number of MDD nodes used for storing the relations.

6.1 Results

The results are in Table 2. The models are ordered by their number of states. When a tool was not able to complete within the constraints, the larger model was skipped for that tool. For the `responsiveness` property the `pbes2bool` did not complete within the 20 GB memory bound and `pbes2lts-sym` could not generate the parity game within 24 h. Also, `four.7x6` could not be solved by any of the tools.

We can make the following observations. From the results we see that of the different options for `pbes2lts-sym`, our new approach performs always best, both in time and memory, compared to the splitting and simple approach. For both Connect Four and the Wheel model, `pbes2bool` is up to 9 times slower and uses up to more than 80 times more memory than the structured approach with `pbes2lts-sym`, for cases where both tools finished within the constraints. Comparison with NuSMV shows a mixed picture. Structured `pbes2lts-sym` is up to 4.5 times faster, but NuSMV uses up to 1.9 times less memory (considering the structured approach). Also, NuSMV exceeds the time limit for `four.6x5` and larger and `pbes2lts-sym` exceeds the memory limit for `four.7x5` and larger. That NuSMV is slower can be explained by the the conjunctive partitioning that is used in the tool, while the non-determinism in the Connect Four gives rise to a disjunctive specification, which is exploited by the disjunctive partitioning in `pbes2lts-sym`. The better memory performance of NuSMV is due to the encoding of the board using two bits per field, where `pbes2lts-sym` uses a 32-bit integer for every field.

There is a large difference between the different models in that for the Connect Four game most time and memory is spent on solving the generated game, not on generating it; for the `wheel` FSMs it is the other way around.

7 Conclusions

We have presented an improved method for verifying modal μ-calculus for process algebraic specification, consisting of an improved translation of the verification problem to a PBES (`lps2pbes`), an efficient tool for symbolic instantiation of the PBES to a symbolic parity game based on LTSmin (`pbes2lts-sym`), and a new symbolic parity game solver tool (`spgsolver`). The combination of these tools allows for high performance model checking of large systems, using MDDs as data structures. The structure of the specification is used for choosing a good partition of the transition relation, allowing for efficient application of operations on these data structures.

We compared the performance of the new solution to the existing tool `pbes2bool` and the symbolic model checker NuSMV. The new LTSmin based tools perform much better than the previous version and than `pbes2bool` both in execution time and memory usage. NuSMV is more memory efficient, but slower, in comparison to our approach.

We intend to experiment with combinations of disjunctive and conjunctive partitioning in the `pbes2lts-sym` tool and the symbolic parity game solver. We want to extend LTSmin to allow for translation to a parity game for any supported input language. Furthermore, we want to apply optimisations like saturation and the parallel application of MDD operations in the parity game solver.

References

[1] M. Bakera, S. Edelkamp, P. Kissmann & C.D. Renner (2009): *Solving μ-Calculus Parity Games by Symbolic Planning*. In: *MoChArt 2008*, *LNCS* 5348, Springer, doi:10.1007/978-3-642-00431-5_2.

[2] S.C.C Blom & J.C. van de Pol (2008): *Symbolic Reachability for Process Algebras with Recursive Data Types*. In: *ICTAC 2008*, *LNCS* 5160, Springer, doi:10.1007/978-3-540-85762-4_6.

[3] S.C.C. Blom, J.C. van de Pol & M. Weber (2010): *LTSmin: Distributed and Symbolic Reachability*. In: *CAV 2010*, *LNCS* 6174, Springer, doi:10.1007/978-3-642-14295-6_31.

[4] J.C. Bradfield & C. Stirling (2001): *Modal logics and mu-calculi: An introduction*. In J.A. Bergstra, A. Ponse & S.A. Smolka, editors: *Handbook of Process Algebra*, chapter 4, Elsevier, pp. 293–330, doi:10.1016/B978-044482830-9/50022-9.

[5] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan & D.L. Dill (1994): *Symbolic Model Checking for Sequential Circuit Verification*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(4), pp. 401–424, doi:10.1109/43.275352.

[6] G. Ciardo, G. Lüttgen & R. Siminiceanu (2001): *Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation*. In: *TACAS 2001*, *LNCS* 2031, Springer, doi:10.1007/3-540-45319-9_23.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani & A. Tacchella (2002): *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*. In: *CAV 2002*, *LNCS* 2404, Springer, doi:10.1007/3-540-45657-0_29.

[8] O. Friedmann & M. Lange (2009): *Solving Parity Games in Practice*. In: *ATVA 2009*, *LNCS* 5799, Springer, doi:10.1007/978-3-642-04761-9_15.

[9] E. Grädel, W. Thomas & T. Wilke, editors (2002): *Automata Logics, and Infinite Games*. *LNCS* 2500, Springer, doi:10.1007/3-540-36387-4.

[10] J.F. Groote & M.A. Reniers (2001): *Algebraic process verification*. In J.A. Bergstra, A. Ponse & S.A. Smolka, editors: *Handbook of Process Algebra*, chapter 17, Elsevier, pp. 1151–1208, doi:10.1016/B978-044482830-9/50035-7.

[11] J.F. Groote & T.A.C. Willemse (2005): *Model-checking processes with data*. *Science of Computer Programming* 56(3), doi:10.1016/j.scico.2004.08.002.

[12] J.F. Groote & T.A.C. Willemse (2005): *Parameterised boolean equation systems*. *Theoretical Computer Science* 343(3), doi:10.1016/j.tcs.2005.06.016.

[13] Y.L. Hwong, J.J.A. Keiren, V.J.J. Kusters, S. Leemans & T.A.C. Willemse (2013): *Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider*. *Science of Computer Programming*, doi:10.1016/j.scico.2012.11.009.

[14] G. Kant & J.C. van de Pol (2012): *Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games*. In: *Graphite 2012*, *EPTCS* 99, doi:10.4204/EPTCS.99.7.

[15] R. Mazala (2002): *Infinite Games*. In: [9], pp. 197–204, doi:10.1007/3-540-36387-4_2.

[16] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier & C. Pixley (1995): *Efficient BDD Algorithms for FSM Synthesis and Verification*. In: *IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV)*.

[17] W. Zielonka (1998): *Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees*. *Theoretical Computer Science* 200(1–2), doi:10.1016/S0304-3975(98)00009-7. doi:10.1016/S0304-3975(98)00009-7.