Solving μ-Calculus Parity Games by Symbolic Planning

Marco Bakera, Stefan Edelkamp, Peter Kissmann, and Clemens D. Renner

Department of Computer Science
Dortmund University of Technology
{firstname.lastname}@cs.tu-dortmund.de

Abstract. This paper applies symbolic planning to solve parity games equivalent to μ-calculus model checking problems. Compared to explicit algorithms, state sets are compacted during the analysis. Given that $diam(G)$ is the diameter of the parity game graph G with node set V, for the alternation-free model checking problem with at most one fixpoint operator, the algorithm computes at most $O(diam(G))$ partitioned images. For d alternating fixpoint operators, $O(d \cdot diam(G) \cdot (\frac{|V|+(d-1)}{d-1})^{d-1})$ partitioned images are required in the worst case.

Practical models and properties stem from data-flow analysis, with problems transformed to parity game graphs, which are then compiled to a general game playing planner input.

1 Introduction

Symbolic μ-calculus model checking with BDDs [8] has been applied as a general framework for various verification problems like model checking of LTL and CTL formulas or testing for bi-simulation equivalence and language containment. One successful tool is μcke [3]. On the other hand, μ-calculus model checking problems have been converted to parity games [22]. Different tools like Omega [40] and MetaGame [43] have been developed.

Specialized game playing is one of the major successes in AI [29]. In general game playing [26], strategies are computed domain-independently without knowing which game is played. Best policies result in perfect play. The opponents can take actions alternately and independently and attempt to maximize the outcome. The game description language (GDL) is designed for use in defining complete information games. It is a subset of first order logic, using the syntax of the knowledge interchange format (KIF) [18].

This paper attempts to close the gap between general symbolic game playing and model checking, which is based on checking the satisfiability of formulas [25,4]. Here, we refer to symbolic exploration in the context of using binary decision diagrams (BDDs) [6].

For parity games, strategy improvement [39] and progress measure algorithms [23] are prominent. The latter one has been translated to a symbolic setting by providing an algorithm with $O(|V|^{d+3} \log(|V|))$ (ADD) images [7]. In this paper, we improve the results for the μ-calculus to $O(d \cdot \mathit{diam}(G) \cdot (\frac{|V|+(d-1)}{d-1})^{d-1})$ possibly partitioned (BDD) images, where *diam* is the diameter of G and d is the fixpoint alternation depth of the formula. We also provide a theoretically faster algorithm for full alternation.

The paper is structured as follows. First, we review the symbolic classification algorithm for two-player zero-sum games that is included in our general game playing tool. Next, we introduce the basics of game-based model checking and the transformation of μ-calculus model checking problems to parity games. The transformation is illustrated with a simple example. We then show how the classification algorithm solves the problem of parity games that are generated by formulas in the alternation free μ-calculus. The extension to larger fragments of the μ-calculus is discussed together with a transformation of an existing explicit-state strategy synthesis algorithm. Independent proofs of correctness are given for both algorithms. In the empirical part, we analyze model checking problems from data-flow analysis and convert them to parity games and general game playing inputs, on which our planner is applied. Finally, we draw conclusions.

2 Symbolic Analysis of Two-Player Games

A two-player zero-sum game (with perfect information) is given by a set of states S, move-rules to modify states and two players, called player 0 and player 1. Since exactly one player is active at any given time, the entire state space of the game is $S \times \{0, 1\}$. A game has an initial state and some predicate *Goal* to determine whether the game has come to an end. For now, we assume that every path from the initial state is finite. Assuming optimal play and starting with all lost goal positions of one player, all previous lost positions have to be computed. A position is lost if all moves lead to an intermediate position in which the other player can force a move back to a lost position.

In symbolic search with BDDs, states are manipulated in form of sets by computing images. The image of a state set *States* wrt. the transition relation $\mathit{Trans}(x, x')$ is equal to computing $\mathit{WeakImage}(\mathit{Trans}, \mathit{States}) := \exists x. \mathit{Trans}(x, x') \wedge \mathit{States}(x)$, where x and x' are vectors of Boolean state variables. The result of this image operation is a representation of all states reachable from *States* in one step. In order to repeat the process, we substitute x with x'. In an interleaved representation this operation reduces to a textual replacement of node labels in the BDD. For computing the image, a monolithic transition relation is not required. Instead, a sub-relation Trans_a is stored together with every move $a \in \{1, \ldots, k\}$. The image of a state set *States* is partitioned into $\mathit{WeakImage}(\mathit{Trans}, \mathit{States}) = \exists x.(\mathit{Trans}_1(x, x') \wedge \mathit{States}(x)) \vee \ldots \vee \exists x.(\mathit{Trans}_k(x, x') \wedge \mathit{States}(x))$.

In contrast to reachability analysis (which can be invoked to initialize the classification algorithm), the direction of the symbolic retrograde analysis is *backwards*. Fortunately, symbolic backward search causes no problem, as the representation of all moves is defined as a relation. With symbolic search, two-player games with perfect information can be classified iteratively using BDDs. For it we furthermore need to calculate strong preimages: $\mathit{StrongPreImage}(\mathit{Trans}, \mathit{States}) := \forall x'.$

Algorithm 1. Symbolic classification of two-player zero-sum games

```
Data: Transition Relation Trans, Initial State Set Init, Goal Sets Goal, Leaf
      evaluation Eval.
Result: Four Classification Sets.
1  $(Reached, L(0), L(1)) \leftarrow Reachable(Init, Trans, Goal, Eval)$;
2  foreach $i \in \{0,1\}$ do
3      $New \leftarrow Lose(i) \leftarrow L(i)$;
4      $Win(1-i) \leftarrow \bot$;
5      repeat
6          $Weak \leftarrow Move(1-i) \wedge WeakPreImage(Trans, New) \wedge Reached$;
7          $Win(1-i) \leftarrow Win(1-i) \vee Weak$;
8          $Strong \leftarrow Move(i) \wedge StrongPreImage(Trans, Win(1-i)) \wedge Reached$;
9          $New \leftarrow Strong \wedge \neg Lose(i)$;
10         $Lose(i) \leftarrow Lose(i) \vee New$;
11     until $New = \bot$ ;
12 return $(Win(0), Lose(0), Win(1), Lose(1))$;
```

$Trans(x,x') \Rightarrow States(x')$. Fortunately, with it a partitioned computation also applies. Since $StrongPreImage(Trans, States) = \neg\, WeakPreImage(Trans, \neg States)$ with $WeakPreImage(Trans, States) := \exists x'.Trans(x,x') \wedge States(x')$, we can induce $StrongPreImage(Trans, States) = \forall x'.(Trans_1(x,x') \Rightarrow States(x')) \wedge \ldots \wedge \forall x'.(Trans_k(x,x') \Rightarrow States(x'))$.

Algorithm 1 shows the classification algorithm for computing strategies in turn-taking games as mentioned in [12]. The idea of attractors, however, goes back to [44]. First of all, we calculate all the reachable states through forward reachability analysis; a backward exploration can result in states that are unreachable from the initial state. Next, we construct four sets: The lost states $Lose(i)$ for each player and the won states $Win(i)$ for each player. The lost states for player i are initialized with the BDDs $L(i)$. From these we only take those goal states that are reachable. Won states are initialized with the BDD $\bot$ for the *false* function, representing the empty set. Now we construct the predecessors of the lost states. Here, the last move has to be made by player $(1-i)$, the opponent of player i; this predicate is denoted by $Move(1-i)$. These predecessors are then added to the won states of the opponent. Starting from those won states we calculate their predecessors. Here, the last move has to be made by the current player i. These new states are added to the lost states. If there are no new states at this point, the calculation terminates (for the current player).

Once the algorithm has ended for both players, we can simply check in which set the initial state resides. If it is in one of the sets of won states, the corresponding player can assure a victory; if it is in one of the lost states, the opponent can assure victory (independent of the other player's moves). If the initial state is in none of these four sets, a finite game surely ends in a draw – always assuming both players perform optimal play.

The number of images for determining all reachable states and the number of images for their classification is linear in the maximal BFS layer, known as the radius of the problem r. By introducing *no-ops*, one can transform any (non simultaneous) game into a turn-taking game by at most doubling the game graph. For games with loops and no draws, the classification algorithm (started for each player) still might leave a set of positions unclassified. These sets correspond to an infinite game play without further progress. Note that by applying retrograde analysis, some states may be classified despite the fact that they lie on a cycle.

Extensions of the algorithm to games with arbitrary costs are proposed in [14,15].

3 Preliminaries: Model Checking Based on Parity Games

Model checking is a procedure for the automated verification of software and hardware systems. The system model (e.g., in form of a Kripke transition system[1]) is checked for the validity of a formula in temporal logic.

3.1 μ-Calculus Model Checking

Modal μ-calculus formulas ϕ are built from propositions (basic properties of the system's states) $p \in AP$, standard Boolean operators, $\langle a\rangle\phi$ (*possibility*) and $[a]\phi$ (*necessity*) modal operators on actions $a \in A$, as well as minimal and maximal fixpoint operators $\mu X.\phi$ and $\nu X.\phi$. The μ and ν operators act as binders for fixpoint variables. The following (minimal) syntax denotes the modal μ-calculus with X being a fixpoint variable (the dual operators can be derived from this base set of operators):

$$\phi ::= \mathit{true} \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \langle a\rangle\phi \mid X \mid \mu X.\phi$$

For brevity, we write $\langle\cdot\rangle\phi := \bigvee_{a\in A}\langle a\rangle\phi$ and $[\cdot]\phi := \bigwedge_{a\in A}[a]\phi$.

3.2 Parity Games

A parity game graph $G = (V_\Diamond, V_\Box, E, p)$ is composed of two disjoint sets of vertices $V_\Diamond$ and $V_\Box$, an edge set $E \subseteq V \times V$, where $V = V_\Diamond \cup V_\Box$, and a priority function $p : V \to \{1, 2, \ldots, d\}$, for some integer d, defined on its vertices. The game is played by two players: *diamond* and *box*. The game starts at some vertex $v_0 \in V$. The players construct a possibly infinite path as follows: Let u be the last vertex added so far to the path. If $u \in V_\Diamond$, then diamond chooses an edge $(u, v) \in E$. Otherwise, if $u \in V_\Box$, then box chooses an edge $(u, v) \in E$. In either case, vertex v is added to the path, and a new edge is then chosen by either diamond or box. Let $v_0, v_1, \ldots$ be the path constructed by the two players, and $p(v_0), p(v_1), \ldots$ the sequence of the priorities of the vertices on the path. Diamond wins the game if the path ends in a leaf node in $V_\Box$ or the smallest priority seen infinitely many times is even, while box wins otherwise.

[1] The model $M := (S, A, AP, \to, I)$ is composed of a set of states S, a set of actions A (from labeled transition systems), a set of atomic propositions AP (from Kripke structures), a transition relation $\to \subseteq S \times A \times S$, and an interpretation function $I : S \to 2^{AP}$ (assigning propositions to states).

4 Solving Parity Games for Alternation-Free Formulas

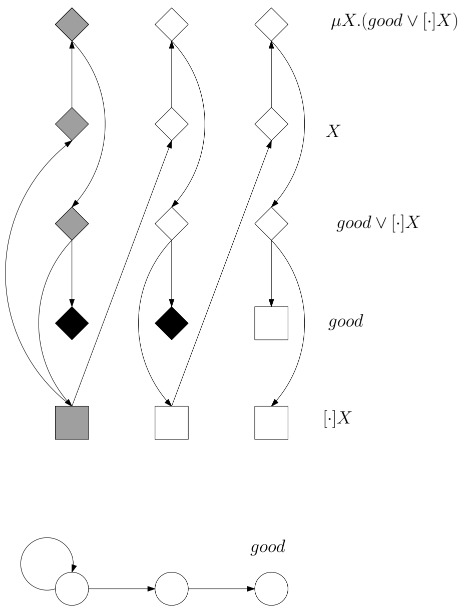
It has been shown that model checking systems with property specification in the μ-calculus is equivalent to solving a parity game, with the maximal priority roughly corresponding to the alternation depth of the μ-formula ϕ [9]. For each play there is a unique partitioning of the parity game in two winning sets, see for example [16].

Local model checking approaches like [9] generate a node for each state in the system and each sub-formula of ϕ. In the following we generally assume alternation-free μ-formulas. The transformation of ν-formulas is dual (changing the roles of box and diamond).

We transform the model and the μ-calculus formula into a parity game as implemented in the tool GEAR [1,2]. Figure 1 shows a simple model with respect to an alternation-free μ-calculus formula $\mu X.(\mathit{good} \vee [\cdot]X)$ and its translation to a parity game graph (nodes are of type either box or diamond, node priorities are all 1). The color shading will be explained below.

Marking a node won corresponds to a definite win for player diamond, marking it lost corresponds to a definite win for player box (assuming optimal play). For the recursive winning set computation we may apply the following rules (assuming player diamond's point of view).

- v is a $\diamond$ node
  - v is a leaf $\Rightarrow$ v is marked lost



$\mu X.(\mathit{good} \vee [\cdot]X)$
X
$\mathit{good} \vee [\cdot]X$
good
$[\cdot]X$
good

Fig. 1. A model (bottom) wrt. the alternation-free μ-calculus formula $\mu X.(\mathit{good} \vee [\cdot]X)$ and the classified parity game graph (top). Here, *good* is an atomic proposition.

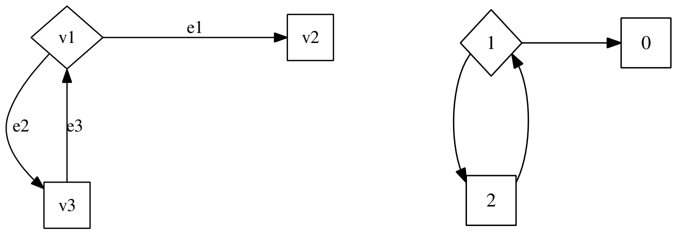


Fig. 2. Example game graph (left) and backward BFS layers (right)

  - there is a successor u of v marked won $\Rightarrow$ mark v won
  - all successors u of v are marked lost $\Rightarrow$ mark v lost
- v is a $\Box$ node
  - v is a leaf $\Rightarrow$ v is marked won
  - there is a successor u of v marked lost $\Rightarrow$ mark v lost
  - all successors u of v are won $\Rightarrow$ mark v won

If there is a lasso in the graph, it is not immediate to determine a strategy defined as a subset of edges completely classifying optimal play of player diamond. Consider the parity game graph given in Fig. 2 (left). All states belong to the diamond player's winning set, but if he chooses to take the vertical edge leading to the state on the bottom of the graph, the box player will win, as the game results in an infinite cycle, which is won by the box player.

The idea to remedy this is to store the backward BFS layer each state was classified in. The player then has to take an edge that leads to a state that was detected earlier in the backward search and thus is stored in a smaller layer (smaller by 1). In the example game in Fig. 2 (right), the numbers in the nodes denote the backward layers. Following the strategy to take only actions to layers with smaller numbers, the diamond player easily wins – he just has to take the horizontal edge to the terminal state.

Theorem 1. *For the case of alternation-free parity games, given the backward layers of the classification from the calculation of the winning sets, it is possible to calculate the diamond player's strategy.*

Proof. Let L_i denote the set of states found in the ith layer of the backward search and $W_\Diamond = \bigcup_{i>0} L_i$ the winning set of the diamond player as determined by the algorithm. We define the optimal strategy for player diamond as $E_\Diamond := \{(u,v) \in W_\Diamond \times W_\Diamond \mid \exists i > 0.\ u \in L_i \wedge v \in L_{i-1}\}$.

We prove the correctness of this by induction on the BFS layer $i > 0$. For the states from the set L_1, it is clear that the diamond player will win by taking the edge to the terminal state (within L_0). For the states in L_i, the player can take an edge leading to a state in L_{i-1}. From there, we inductively already have a strategy.

Formulas in Hennessy-Milner logic [20] have no fixpoint operators. The corresponding game is finite and the classification algorithm will end up with a complete strategy for both players. The search for a strategy in a parity game matches the one in a two-player zero-sum game given that one player is box and the other player is diamond.

As only one reachability step and one call to the classification algorithm is executed, backward analysis is restricted to the reachable set in form of a DAG which is traversed bottom-up. Thus we obtain that the symbolic classification algorithm for parity games arising from model checking problems with temporal logic properties in Hennessy-Milner logic is correct and amounts to $O(\mathit{radius}(G, \mathit{Init}))$ possibly partitioned (BDD) images, where $\mathit{radius}(G, \mathit{Init})$ is the maximum BFS-layer of the forward search of G starting from *Init*.

If one μ fixpoint operator is present in the according parity game, we have to search for strategies in which diamond must avoid a cycle. Otherwise, box can establish a cycle and wins the game. The translation to an ordinary game graph for cyclic solutions may become involved, as infinite games are played. In order to detect cycles, one might try adopting the state recording method for transforming safety into liveness [32]. However, an application is not immediate.

Next, we recognize that all states that remain unmarked lie on a cycle that player diamond cannot avoid; according to the winning condition, these states can be marked lost. Figure 1 displays the result of such a complete classification of the parity game graph. Black shading indicates that player box wins from there, white indicates nodes won by player diamond and gray denotes nodes won by player box due to an infinite cycle. The respective other player loses due to the zero-sum character of the game.

For alternation-free formulas, all priorities in the graph are the same. W.l.o.g. we consider μ formulas only, such that all priorities are 1 and player box wins the game (assuming optimal play) if he[2] can force player diamond to stay on a cycle. The remaining unclassified nodes all lie on a lasso[3]. Moreover, we need to show that player diamond cannot escape the cycle unless he navigates to some black node (where he loses). Let us assume that one of the remaining nodes is not on a lasso. This implies that the node would be on a path to a sink that can be marked won for one of the players. Therefore, the node itself could have been marked as well and would not be unclassified, contrary to our assumption.

Why can diamond not escape from this cycle (lasso)? Any diamond node v on the cycle must have at least one unclassified successor. It cannot have a white successor (marked 'won by diamond') because in that case v would have been marked 'won by diamond' as well. Diamond would not pick a black successor as this implies that he will lose eventually. The case for player box is analogous (no black successors, white successors are avoided). Therefore, only unclassified nodes are played. As an infinite path in a finite graph will eventually lead to a cycle, the unclassified nodes can be marked black. Therefore, all nodes are classified.

[2] For the sake of simplicity, we stay with *he* and *him* instead of *he / she* and *him / her*.

[3] A lasso is a cycle and a prefix of nodes (stem) leading to that cycle.

As only one reachability analysis and one call to the classification algorithm is executed, and given that backward analysis is restricted to the reachable set, the complexity of the algorithm in the number of images is linear in $diam(G)$ and $diam(G^{-1}) = diam(G)$, where G^{-1} is the inverse graph of G (all edges are reversed).

Therefore, we observe that the symbolic classification algorithm for parity games arising from model checking problems wrt. temporal logic properties in alternation-free μ-calculus formulas is correct and amounts to $O(diam(G))$ possibly partitioned (BDD) images, where *diam* is the maximal shortest path length between every two states.

5 Extension to Full Alternation Depth

We now address an extension of our symbolic planning approach to cover parity games for μ-calculus model checking problems with alternation. No polynomial-time algorithm is to be expected, since according to [41] only the solution of games with so-called Büchi winning condition can currently be done in polynomial time. There are sub-exponential algorithms [5,24] with a complexity of $n^{O(\sqrt{n/\log n})}$, which are to be preferred only if the alternation depth is larger than $\Omega(\sqrt{n})$. Even specialized problems like finite-state controller synthesis for r request-response constraints [41] require an exponential time algorithm in r.

In adaptation of the notation used in the literature [42], we define the alternation depth d as the number of alternations between the fixpoint operators ν and μ plus 1 (resulting in the counter-intuitive result that alternation-free μ-calculus formulas have a value of $d = 1$).

In the following, we devise an efficient symbolic algorithm following the explicit-state strategy synthesis algorithm documented in [42], which re-assembles ideas from [9] and [28]. The strategy synthesis algorithm has a time complexity of $O(|E|(|V|/d)^{d-1})$ (assuming a uniform distribution of priorities). Compared to the algorithm of [23], the exponent $d - 1$ matches the value $\lceil d/2 \rceil$ as obtained in the small progress measure algorithm for $d = 2$ and $d = 3$. With further refinements, the results of [42] indicate that for these cases the strategy improvement algorithm will be faster for formulas with small alternation depth (which appear in practice).

Let V_i be the set of nodes for player i. The strategy synthesis algorithm relies on an iterative calculation of *forcing sets*. A forcing set for some subset $V' \subseteq V$ towards some fixed node set $A \subseteq V$ for player $i \in \{0, 1\}$ is defined by the condition that for each node u in V' player i can force player $(1 - i)$ to play towards the node set A. A maximal forcing set from V' to A for player i does not include an edge (v, w) with $v \in V' \cap V_{1-i}$ and $w \in V \setminus (V' \cup A)$.

On acyclic game graphs, the computation of winning sets reduces to the computation of forcing sets as mentioned above. Otherwise, cycles are handled in the synthesis algorithm 5.

First, the nodes are partitioned with respect to their priority (l. 5). There is no need for deeper recursion and refinement of the resulting winning sets when

Algorithm 2. Main

```
Result: Winning Sets $W_0$ and $W_1$.
1 $(R, W_0, W_1) \leftarrow \textit{Initialize}()$;
2 $(W_0, W_1) \leftarrow \textit{Synthesize}(R \wedge \neg(W_0 \vee W_1))$;
3 return $(W_0, W_1)$;
```

Algorithm 3. Initialize

```
Result: Reachable set $R$, winning Sets $W_0$ and $W_1$.
1 $(R, W_1, W_0) \leftarrow \textit{Reachable}(\textit{Init}, \textit{Trans}, \textit{Goal}, \textit{Eval})$;
2 $W_0 \leftarrow \textit{Force}(R, W_0, 0)$;
3 $W_1 \leftarrow \textit{Force}(R, W_1, 1)$;
4 return $(R, W_0, W_1)$;
```

all nodes share the same priority. In this case, winning sets are computed that respect the player that is currently predominating the game when considering priorities (ll. 5-5). Afterwards, an assumption of the winning set for the other player is made (ll. 5-5). The subsequent repeat loop tries to consolidate this assumption (ll. 5-5). The partitioning that emerges either breaks down into exactly two or more classes of nodes with the same priority. In the former case, the assumption is computed as in the cycle-free case (ll. 5-5). The latter case requires refining the assumption – by a new assumption based on the current one – in the recursive call (ll. 5-5). The remainder of the loop (ll. 5-5) collects the results and assigns them to the appropriate player until there is no more need for refinement (l. 5).

Following the presentation in [42], we show symbolic equivalents of the algorithms *Main* (Algorithm 2), *Initialize* (Algorithm 3), *Force*, (Algorithm 4), and *Synthesize* (Algorithm 5). For the ease of presentation, we assume that the parity game graph is *consistent*, such that priorities are consecutive (there is no gap). For game graphs that are translated from model checking tasks, this assumption is necessarily true. We further assume the transition relation *Trans*, the initial state set *Init*, the goal predicate *Goal*, the priority evaluation function *Priority*, and the leaf evaluation predicate *Eval* to be globally accessible.

After initialization, the synthesis algorithm refers to computing the forcing sets and a recursive call to itself. As shown before, all explicit state operations to determine the winning sets for both players can be performed symbolically. The initialization that computes the maximal forcing set for both players on the entire graph towards the terminal nodes matches the classification in Algorithm 1. For computing the forcing sets, the classification algorithm needs to be executed only for one player. Computing the subgraph can either be done in the game description language by specifying different graph and goal conditions (see Appendix B for an example) or via restricting the disjunctive representation of the transition relation to the part that corresponds to the remaining edges.

Algorithm 4. Force

```
Data: Set $V'$, Target Set $A$, Player $i$.
Result: Forcing Set $F$.
1  $\mathit{Trans}' \leftarrow \mathit{Trans} \wedge (V' \times (V' \cup A))$;
2  $\mathit{New} \leftarrow \mathit{Lose}(1-i) \leftarrow (A \wedge \mathit{Move}(1-i))$;
3  $\mathit{Win}(i) \leftarrow (A \wedge \mathit{Move}(i))$;
4  repeat
5      $\mathit{Weak} \leftarrow \mathit{Move}(i) \wedge \mathit{WeakPreImage}(\mathit{Trans}', \mathit{New}) \wedge V'$;
6      $\mathit{Win}(i) \leftarrow \mathit{Win}(i) \vee \mathit{Weak}$;
7      $\mathit{Strong} \leftarrow \mathit{Move}(1-i) \wedge \mathit{StrongPreImage}(\mathit{Trans}', \mathit{Win}(i)) \wedge V'$;
8      $\mathit{New} \leftarrow \mathit{Strong} \wedge \neg \mathit{Lose}(1-i)$;
9      $\mathit{Lose}(1-i) \leftarrow \mathit{Lose}(1-i) \vee \mathit{New}$;
10 until $(\mathit{New} = \bot)$ ;
11 return $(\mathit{Lose}(1-i) \vee \mathit{Win}(i))$;
```

Lemma 1. *The worst-case number of partitioned images of the symbolic classification algorithm for parity games with d alternating fixpoint operators is bounded by $2 \cdot \mathit{diam}(G) \cdot d \cdot \prod_{k=1}^{d-1}(|\mathit{Level}_k| + 1)$.*

Proof. Let $T(G)$ denote the running time for the synthesis algorithms on the graph with $V = \mathit{Support}$, for the number of images we have $T(G) = 2 \cdot \mathit{diam}(G)$, if $d = 1$.

Given that $\mathit{Upper} \cup \mathit{Lower} \subseteq \mathit{Layer}$ and $W_1 \subseteq \mathit{Layer}$ we obtain a recursive equation for the asymptotic complexity of

$$T(G) = 1 + \sum_{j=0}^{r} 2 \cdot \mathit{diam}(G|_{\mathit{Layer}_j}) + T(G|_{\mathit{Upper}_j})$$

images, where r is the number of iterations of the repeat loop, Layer_j is the set representing Layer and Upper_j is the set representing $\mathit{Upper} \wedge \neg W$ in the j-th iteration. Both calls to the *Force* function induce at most $\mathit{diam}(G|_{\mathit{Layer}_j})$ many images and the plus 1 is due to the weak pre-image in line 5. $G|_{V'}$ is the subgraph restricted to $V' \subseteq V$, more precisely $G|_{V'} := (V', E \cap (V' \times V'))$.

The next step is to rewrite the equation to avoid recursion. Following the inductive argument in [42], see Appendix A, we have

$$T(G) \leq 2 \cdot \sum_{j=1}^{d-1} \mathit{diam}(G) \cdot \prod_{k=1}^{j}(|\mathit{Level}_k| + 1)$$

many images in the worst case, where Level_k is the set of nodes in graph G with priority k.

$$T(G) \leq 2 \cdot \sum_{j=1}^{d-1} \mathit{diam}(G) \cdot \prod_{k=1}^{d-1}(|\mathit{Level}_k| + 1)$$

Algorithm 5. Synthesize

```
**Data**: Node Set $Support$.
**Result**: Winning Sets $Z_0$, $Z_1$.
if $(Support = \bot)$ then
    return $(\bot, \bot)$;
$m \leftarrow MinPriority(Priority \wedge Support)$;
$m' \leftarrow m + 1$;
$i \leftarrow (m \bmod 2)$;
if $MaxPriority(Priority \wedge Support) = m$ then
    if $(i = 0)$ then
        return $(Support, \bot)$;
    else
        return $(\bot, Support)$;
$Layers \leftarrow Support$;
$Lower \leftarrow Priority_{<m'} \wedge Layers$;
$Upper \leftarrow Support \wedge \neg Lower$;
repeat
    if $(Lower = \bot)$ then
        $W \leftarrow Force(Upper, Lower, i)$;
    else
        $W \leftarrow \bot$;
    if $(i = 0)$ then
        $(W_0, W_1) \leftarrow Synthesize(Upper \wedge \neg W)$;
    else
        $(W_1, W_0) \leftarrow Synthesize(Upper \wedge \neg W)$;
    $Upper \leftarrow Upper \wedge \neg W_1$;
    $Layers \leftarrow Layers \wedge \neg W_1$;
    $Z_1 \leftarrow Z_1 \vee W_1$;
    if $(Lower \neq \bot \wedge W_1 \neq \bot)$ then
        $W' \leftarrow Force(Layers, W_1, 1 - i)$ ;
        $Lower \leftarrow Lower \wedge \neg W'$;
        $Upper \leftarrow Upper \wedge \neg W'$;
        $Layers \leftarrow Layers \wedge \neg W'$;
    else
        $W' \leftarrow \bot$;
until $(W' = \bot)$ ;
$Z_0 \leftarrow W \vee W_0 \vee (Lower \wedge Move(i) \wedge WeakPreImage(Trans, W \vee W_0 \vee Lower))$;
if $(i = 0)$ then
    return $(Z_0, Z_1)$;
else
    return $(Z_1, Z_0)$;
```

$$\leq 2 \cdot diam(G) \cdot d \cdot \prod_{k=1}^{d-1} (|Level_k| + 1).$$

Theorem 2. *The symbolic classification algorithm for parity games arising from model checking problems wrt. temporal logic properties in μ-calculus formulas with d alternating fixpoint operators is correct. For $d > 1$, the worst-case number of partitioned images is $O(d \cdot diam(G) \cdot (\frac{|V|+(d-1)}{d-1})^{d-1})$.*

Proof. The correctness of the algorithm is inherited from the correctness of the explicit-state variant documented in [42].

The number of images for initialization is $O(radius(G, Init))$ for computing the reachable set and $O(diam(G))$ for computing the two forcing sets. Let $T(G)$ denote the running time for the synthesis algorithms on the graph with $V = Support$.

For $d > 1$, we have

$$\begin{aligned} T(G) &\le 2 \cdot diam(G) \cdot d \cdot \prod_{k=1}^{d-1} (|Level_k| + 1) \\ &\le 2 \cdot diam(G) \cdot d \cdot \left(\frac{\sum_{k=1}^{d-1} (|Level_k| + 1)}{d-1} \right)^{d-1} \\ &= 2 \cdot diam(G) \cdot d \cdot \left(\frac{|V| + (d-1)}{d-1} \right)^{d-1} \end{aligned}$$

partitioned images in the worst case. For the penultimate step we used the inequality for the geometric wrt. the arithmetic mean.

The worst-case number of BDD images beats the value of $O(|V|^{d+3} \log(|V|))$ images obtained by [7]. For the important subclass $d = 2$, our algorithm reduces to only $O(diam(G) \cdot |V|)$ (BDD) images compared to $O(|V|^5 \log(|V|))$ (ADD) images [7].

6 Empirical Analysis

We draw experiments with our general game playing planning tool [14], which itself uses CUDD[4] by Fabio Somenzi as the underlying BDD library. The models and formulas were generated from data-flow analysis problems and translated into parity game graphs using GEAR. The export format of GEAR was adapted to suit the game-based planner. Moreover, we introduced no-operators to allow the game to be turn-taking. Parts of the specification of the GDDL encoding[5] for the example problem is provided in appendix B.

One important fact about our tool is the minimization of the state encoding by building groups of mutually exclusive propositions [13,19]. As a result, we can apply a binary state encoding. This is the key to a space-efficient representation of the states, since in a BDD many states share nodes and exponentially many nodes may be represented in a polynomially sized graph.

[4] http://vlsi.colorado.edu/~fabio/CUDD

[5] GDDL is a language introduced by [14]; a hybrid of GDL (Game Description Language) and PDDL (Planning Domain Definition Language).

6.1 Data-Flow Analysis as Model Checking

Data-flow analysis (DFA) is one step at the compile time of a program, prior to its optimization. Many DFA demands have been transformed into model checking problems [34]. The main idea is to interpret control flow graphs as Kripke transition systems with program steps labeling nodes and edges. Basic propositions at a node are *isDefined*(x), denoting that variable x is written or changed, and *isUsed*(x), denoting that variable x is read. A variable is *live* if it is used and was not redefined before – in terms of temporal logics this can be expressed as $\mu X.\mathit{isUsed}(x) \vee \neg\mathit{isDefined}(x) \wedge \langle\cdot\rangle X$.

Many such formulas are free of alternation as shown in [31]. This makes data-flow analysis via model checking a good testbed for our search algorithms.

6.2 Experiments

We have performed three experiments, obtaining matching results wrt. GEAR.

The first example is based on the Java byte code of an implementation of the *Fast Fourier Transformation.* The byte code has been transformed into a control-flow graph using Soot[6]. For its liveness analysis, 749 states are reachable in 366 steps. For these, 141 BDD nodes are needed. The classification algorithm can classify 517 states: 343 states are won for the diamond player, which are represented by 155 BDD nodes. These are found after 39 iterations through the loop. For the box player, 174 states are classified as won. Here, 11 iterations and 112 BDD nodes are needed. These states contain the initial state, i.e., it is surely won for the box player. The remaining 232 states are not classified by our algorithm, so they must lie on one or more lassos from which the diamond player cannot escape. Thus, they are also won for the box player. The total runtime of the forward and backward analysis was 0.8 seconds.

The second example consists of automatically generated code as described in [21], also used as an input to Soot. We expect that this leaves more room for data-flow analysis wrt. possible optimizations. We reached a total of $4,590$ states after $3,086$ steps and need 619 BDD nodes to represent them all. In this case, all states are also reachable in backward direction: The algorithm classifies $3,888$ as won for the diamond player. For these, it needs 128 iterations and 770 BDD nodes. Within two steps it classifies the remaining 702 states as won for the box player using 350 BDD nodes. The initial state is contained within the set of states won for the box player. The total runtime for the forward and backward analysis was 24 seconds.

Instead of source code, the third example for the DFA-MC paradigm considers process graphs edited by the jABC tool[7], where a model is converted to a characteristic formula, which is checked together with a failure specification. The explicit model had 49,141 nodes in the parity graph. The reachability analysis converts the graph into a turn-taking game with 96,616 states and 13,110 BDD nodes, generated in 63 steps. The number of states (BDD nodes) that are won

[6] http://www.sable.mcgill.ca/soot
[7] http://jabc.cs.uni-dortmund.de

for diamond are 22,682 (12,198); the number of states directly won by player box are 26,510 (12,736). The remaining 43,421 non-classified states correspond to the situation that box can enforce player diamond to stay on a cycle. The entire classification took 123 seconds.

7 Conclusion and Discussion

We have seen a fruitful approach for the symbolic analysis of parity games that arise when transforming μ-calculus model checking problems. The algorithms for the Hennessy-Milner and alternation-free μ-calculus are efficient and have been implemented. Testbeds arose during data-flow analysis. Moreover, an implementation for formulas with large alternation depth has been presented.

The historical roots of the work are as follows. As a winning condition for games, the parity condition was already considered by Emerson and Jutla [16]. It was shown that parity games always result in memoryless winning strategies – determinicity of parity games follows directly from the determinicity of Borel games. The algorithmic presentation of McNaughton [28] and the early analyses of Zielonka [44] lay the basis for algorithms based on recursive reachability. No implementation was provided. Thomas [37] realized the importance of parity games. One of the first implementations is the (explicit-state) fixpoint analysis machine [35], which provides a tool based on [9], which was in turn the basis for (explicit-state) strategy synthesis algorithm by [42]. For uniform priorities, [42] shows an advantage wrt. FAM, and $d \leq 3$ (the practical cases) an advantage to Jurdzinski [42]. There are two recent improvements for enumeration we are aware of: the $O(|E| \cdot |V|^{d/3})$ algorithm by [30], and an acceleration for the 3-priorities [11]. A distributed implementation for parity games shows a rather direct adaption of Jurdzinski's small progress measurement algorithm for multi-core architectures based on different state-space partitioning functions [38].

Applying symbolic game playing has different advantages. First, the representation of the winning sets (e.g., in a binary encoding of the nodes) is implicit and can be much smaller than the explicit one. Wrt. space consumption for progress measures, no vectors have to be stored together with each state. Moreover, the analysis can be extended to implicit parity game graphs. Last but not least, BDDs show advantages to SAT and QBF solvers in combinatorial games [27].

Given a symbolic parity game graph representation in a BDD, the above algorithm is capable of solving much larger problems. In other words, we cover a more powerful input language, which allows the succinct specification of nontrivial game graphs using Boolean formulas. So far we have extracted explicit-state models from the GEAR model checker [1,2]. In the future we will likely integrate our implementation to the global model checker in the jABC framework to access the model checking problem, prior to the explicit graph construction.

References

1. Bakera, M., Margaria, T., Renner, C.D., Steffen, B.: Game-based model checking for reliable autonomy in space. Journal of the American Institute of Aeronautics and Astronautics (AIAA) (to appear)
2. Bakera, M., Margaria, T., Renner, C.D., Steffen, B.: Verification, diagnosis and adaptation: Tool supported enhancement of the model-driven verification process. In: Revue des Nouvelles Technologies de Information (RNTI-SM-1), pp. 85–98 (to appear) ISBN 2854288148
3. Biere, A.: μcke – efficient μ-calculus model checking. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 468–471. Springer, Heidelberg (1997)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (1999)
5. Björklund, H., Sandberg, S., Vorobyov, S.G.: A discrete subexponential algorithm for parity games. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 663–674. Springer, Heidelberg (2003)
6. Bryant, R.E.: Symbolic manipulation of Boolean functions using a graphical representation. In: ACM/IEEE Design Automation Conference, pp. 688–694 (1985)
7. Bustan, D., Kupferman, O., Vardi, M.Y.: A measured collapse of the modal μ-calculus alternation hierarchy. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 522–533. Springer, Heidelberg (2004)
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
9. Cleaveland, R., Klein, M., Steffen, B.: Faster model checking for the modal μ-calculus. Theoretical Computer Science 663, 410–422 (1992)
10. Cleaveland, R., Steffen, B.: A linear-time model-checking algorithm for the alternation-free modal mu-calculus. Formal Methods in System Design 2(2), 121–147 (1993)
11. de Alfaro, L., Faella, M.: An accelerated algorithm for 3-color parity games with an application to timed games. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 108–120. Springer, Heidelberg (2007)
12. Edelkamp, S.: Symbolic exploration in two-player games: Preliminary results. In: AIPS-Workshop on Model Checking, pp. 40–48 (2002)
13. Edelkamp, S., Helmert, M.: Exhibiting knowledge in planning problems to minimize state encoding length. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 135–147. Springer, Heidelberg (2000)
14. Edelkamp, S., Kissmann, P.: Symbolic exploration for general game playing in PDDL. In: ICAPS-Workshop on Planning in Games (2007)
15. Edelkamp, S., Kissmann, P.: Symbolic classification of general two-player games. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS, vol. 5243, pp. 185–192. Springer, Heidelberg (2008)
16. Emerson, E.A., Jutla, C.S.: Tree automata μ-calculus and determinacy. In: Foundations of Computer Science, pp. 368–377 (1991)
17. Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional mu-calculus. In: Symposium on Logic in Computer Science, pp. 267–278 (1986)
18. Genesereth, M.R.: Knowledge interchange format. In: Second International Conference on Principles of Knowledge Representation and Reasoning, pp. 238–249 (1991)

19. Helmert, M.: A planning heuristic based on causal graph analysis. In: International Conference on Automated Planning and Scheduling, pp. 161–170 (2004)
20. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 299–309. Springer, Heidelberg (1980)
21. Jørges, S., Kubczak, C., Pageau, F., Margaria, T.: Model driven design of reliable robot control programs using the jabc. In: 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe), March 2007, pp. 137–148 (2007)
22. Jurdzinski, M.: Deciding the winner in parity games is UP∩co-UP. Information Processing Letters 68(3), 119–124 (1998)
23. Jurdzinski, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
24. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. In: SODA, pp. 117–123 (2006)
25. Kautz, H., Selman, B.: Pushing the envelope: Planning propositional logic, and stochastic search. In: European Conference on Artificial Intelligence, pp. 1194–1201 (1996)
26. Love, N.C., Hinrichs, T.L., Genesereth, M.R.: General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group (April 2006)
27. Madhusudan, P., Nam, W., Alur, R.: Symbolic computational techniques for solving games. Electronic Notes in Theoretical Computer Science 89(4) (2004)
28. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic 65, 129–284 (1993)
29. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Solving checkers. In: International Joint Conference on Artificial Intelligence, pp. 292–297 (2005)
30. Schewe, S.: Solving parity games in big steps. In: CAV, pp. 449–460 (2007)
31. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: Conference Record of POPL 1998: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, Janary 19–21, 1998, pp. 38–48 (1998)
32. Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. STTT 5(2-3), 185–204 (2004)
33. Seidl, H.: Fast and simple nested fixpoints. Information Processing Letters 59(6), 119–124 (1996)
34. Steffen, B.: Data flow analysis as model checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–365. Springer, Heidelberg (1991)
35. Steffen, B., Classen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 72–87. Springer, Heidelberg (1995)
36. Stirling, C.: Local model checking games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)
37. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
38. van de Pol, J., Weber, M.: A multi-core solver for parity games. In: PDMC 2008 (to appear, 2008)

39. Vöge, J., Jurdzinski, M.: A discrete strategy improvement algorithm for solving parity games. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)
40. Vöge, J., Ulbrand, S., Matz, O., Buhrke, N.: The automata theory package omega. In: Wood, D., Yu, S. (eds.) WIA 1997. LNCS, vol. 1436, pp. 228–231. Springer, Heidelberg (1998)
41. Wallmeier, N., Hütten, P., Thomas, W.: Symbolic synthesis of finite-state controllers for request-response specifications. In: H. Ibarra, O., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759, pp. 11–22. Springer, Heidelberg (2003)
42. Yoo, H.: Fehlerdiagnose beim Model-Checking durch animierte Strategiesynthese. PhD thesis, Universität Dortmund (2007)
43. Yoo, H., Müller-Olm, M.: MetaGame: An animation tool for model-checking games. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 163–167. Springer, Heidelberg (2004)
44. Zielonka, W.: Infinite games on finite coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200, 135–183 (1998)

A Compiling Away the Recursion

We have to show that

$$T(G) = 1 + \sum_{j=0}^{r} 2 \cdot \mathit{diam}(G|_{\mathit{Layer}_j}) + T(G|_{\mathit{Upper}_j}).$$

induces

$$T(G) \leq 2 \cdot \sum_{j=1}^{d-1} \mathit{diam}(G) \prod_{k=1}^{j} (|\mathit{Level}_k| + 1)$$

Proof. For the induction we observe that $r \leq |\mathit{Lower}_0|$ (at least a one-element set is processed in each iteration) and $G|_{\mathit{Layer}_{|\mathit{Lower}_0|}} = \emptyset$ (no forcing sets in the last iteration) we induce

$$\begin{aligned} T(G) &\leq 1 + \sum_{j=0}^{|\mathit{Lower}_0|} 2 \cdot \mathit{diam}(G|_{\mathit{Layer}_j}) + \sum_{j=0}^{|\mathit{Lower}_0|} T(G|_{\mathit{Upper}_j}) \\ &\leq 1 + \sum_{j=0}^{|\mathit{Lower}_0|-1} 2 \cdot \mathit{diam}(G) + \sum_{j=0}^{|\mathit{Lower}_0|} T(G|_{\mathit{Upper}_j}) \\ &\leq 1 + |\mathit{Lower}_0| \cdot 2 \cdot \mathit{diam}(G) + \sum_{j=0}^{|\mathit{Lower}_0|} T(G|_{\mathit{Upper}_j}) \end{aligned}$$

We additionally observe that $\mathit{Level}_1 = \mathit{Lower}_0$ and that $\sum_{j=0}^{|\mathit{Lower}_0|} T(G|_{\mathit{Upper}_j})$ is bounded by $(|\mathit{Lower}_0| + 1) \cdot T(G|_{\mathit{Upper}_s})$ for some $0 \leq s \leq |\mathit{Lower}_0|$. Hence, by inserting the induction hypothesis (Ind.) we have

$$\begin{aligned} T(G) &\leq 1 + |\mathit{Level}_1| \cdot 2 \cdot \mathit{diam}(G) + \sum_{j=0}^{|\mathit{Level}_1|} T(G|_{\mathit{Upper}_s}) \\ &\leq 1 + |\mathit{Level}_1| \cdot 2 \cdot \mathit{diam}(G) + (|\mathit{Level}_1| + 1) \cdot T(G|_{\mathit{Upper}_s}) \\ &\overset{\text{Ind.}}{\leq} (1+|\mathit{Level}_1|) \cdot 2 \cdot \mathit{diam}(G) + (|\mathit{Level}_1|+1) \cdot 2 \cdot \sum_{j=2}^{d-1} \mathit{diam}(G) \prod_{k=2}^{j} (|\mathit{Level}_k| + 1) \\ &\leq (1 + |\mathit{Level}_1|) \cdot 2 \cdot \mathit{diam}(G) + 2 \cdot \sum_{j=2}^{d-1} \mathit{diam}(G) \prod_{k=1}^{j} (|\mathit{Level}_k| + 1) \\ &\leq 2 \cdot \sum_{j=1}^{d-1} \mathit{diam}(G) \prod_{k=1}^{j} (|\mathit{Level}_k| + 1) \end{aligned}$$

B GDDL Encoding

The parity games have been translated to GDDL. The domain model for the problem looks as follows.

```
(define (domain alternation-free)
 (:types state role)
 (:predicates (at ?s - state) (connect ?s1 ?s2 - state)
        (box ?s - state) (diamond ?s - state) (control ?player - role))
(:action move-box
  :parameters (?player - role ?s1 ?s2 - state ?nextplayer - role)
  :precondition (and (at ?s1)(connect ?s1 ?s2)(control ?player) (box ?s1)
                   (= ?player box_player)(not (= ?player ?nextplayer)))
  :effect (and (not (at ?s1)) (at ?s2)
             (not (control ?player))(control ?nextplayer)))
(:action noop-box
  :parameters (?player - role ?s - state ?nextplayer - role)
  :precondition (and (at ?s) (box ?s) (control ?player)
                 (= ?player diamond_player)(not (= ?player ?nextplayer)))
  :effect (and (not (control ?player)) (control ?nextplayer)))
(:action move-diamond ...)
(:action noop-diamond ...)
(:lost (?player - role)
  (exists (?s - state) (and (at ?s) (= ?player diamond_player)
    (not (control ?player)))))
(:won (?player - role)
  (exists (?s - state) (and (at ?s) (= ?player diamond_player)
    (not (control ?player)))))
(:lost ...)
(:won ...)
```

The example problem from Fig. 1 is encoded as follows.

```
(define (problem check)
  (:domain modelcheck)
  (:objects box_player diamond_player - role
    s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 - state)
  (:init
    (diamond s1) (diamond s6) (diamond s11) (diamond s2) (diamond s7)
    (diamond s12) (diamond s3) (diamond s8) (diamond s13) (diamond s4)
    (diamond s9) (box s14) (box s5) (box s10) (box s15)
    (connect s2 s1) (connect s1 s3) (connect s3 s5) (connect s3 s4)
    (connect s5 s7) (connect s5 s2) (connect s6 s8) (connect s7 s6)
    (connect s8 s10) (connect s8 s9) (connect s10 s12) (connect s11 s13)
    (connect s12 s11) (connect s13 s15) (connect s13 s14)
    (at s1) (control box_player))
  (:goal (exists (?s1 - state)
    (and (at ?s1)
      (or (and (control box_player) (box ?s1))
          (and (control diamond_player) (diamond ?s1)))
      (forall (?s2 - state) (not (connect ?s1 ?s2)))))))
```