Electronic Notes in Theoretical Computer Science 89 No. 4 (2004)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 15 pages

Symbolic Computational Techniques for Solving Games

P. Madhusudan, Wonhong Nam and Rajeev Alur [1]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA.

Abstract

Games are useful in modular specification and analysis of systems where the distinction among the choices controlled by different components (for instance, the system and its environment) is made explicit. In this paper, we formulate and compare various symbolic computational techniques for deciding existence of winning strategies. The game structure is given implicitly, and the winning condition is of the form "*p until q*" for state predicates p and q. The first technique employs symbolic fixpoint computation using ordered binary decision diagrams [8]. The second technique checks for the existence of strategies that ensure winning within k steps, for a user specified bound k, by reduction to the satisfiability of quantified boolean formulas. Finally, the bounded case can also be solved by reduction to satisfiability of ordinary boolean formulas, and we discuss two techniques, one based on encoding the strategy tree, and one based on encoding a witness subgraph, for reduction to SAT. We compare the various approaches on two examples using existing tools such as MOCHA [3], MUCKE [7], SEMPROP [17], QUBE [11], BERKMIN [12].

1 Introduction

The motivation for solving games in formal analysis originated with Church's synthesis problem in the context of automatically synthesizing circuits from specifications [10]. Games have since then become popular in formal methods with various applications including control of discrete event systems [21], realizability and synthesis, and model-checking μ-calculus formulae [24]. In formal verification, they have several applications in verifying reactive systems where the agents comprising the system are viewed as players of a game: in modular verification [16], in synthesis of formal interfaces to modules [9] and in approaches to compositional verification [1,2].

[1] Email: {madhusudan, wnam, alur}@cis.upenn.edu

Research and related applications have led to variety of game formulations such as infinite games on finite graphs, concurrent multi-player games and games on pushdown systems [24]. However, the simplest game that most solutions computationally rely on is the two-player reachability game on a finite graph. Such a game is played between two players, the *system* and the *environment*, and the game problem is to check whether the system has a winning strategy that will force the game from the initial position to some goal position, no matter how the environment plays.

Though the theoretical complexity of solving various games in the literature is well understood, there has been relatively less effort spent in identifying how the powerful symbolic techniques used in model-checking fare in solving games with large state-spaces. In this paper, we initiate such an effort by a comparative and experimental study of solving simple reachability games (augmented with a safety condition) using techniques that use Bdds, Qbf-solvers and Sat-solvers. We model games symbolically using boolean variables and succinct boolean expressions describing the transitions — the explicit game this defines would be typically exponential in the size of the definition.

The standard attractor-set approach to solve reachability games is a simple fix-point algorithm that can easily be implemented using Bdds. There are two kinds of Bdd-based solvers we use: Mocha which is a model-checker that can directly handle specifications in a game logic called alternating-time temporal logic (Atl) and Mucke which is a μ-calculus model checker especially tuned and extended to handle μ-calculus formulas.

For propositional solvers, we consider bounded reachability games. We first consider games where we ask whether the system has a strategy that will ensure the game reaches the goal within k steps, where k is a user-specified parameter. The natural way to encode this as a propositional satisfiability problem is using a quantified boolean formula, where there is a prefix of alternating quantifiers of length $2k$ that capture a strategy for the system followed by a boolean formula that checks whether the strategy is indeed winning for the system. We then use Qbf solvers Semprop [17], Quaffle [25] and Qube [11] to solve these formulas.

In recent years, there has been a significant interest in engineering Sat-solvers that has resulted in very efficient solvers, while the effort in speeding up Qbf solvers has been relatively less. We hence also consider encodings of games into Sat problems, in two different ways. In the first approach, we use Sat to guess a winning strategy tree of depth k (the tree is exponential in k). This can be seen essentially as "unwinding" the alternating quantification in the Qbf formula above into a tree of existential quantifications, by converting each universal choice to all possible choices. We hence get an exponential-sized Sat formula which is satisfiable if and only if there is a strategy that wins in k steps, and we use the Sat-solvers Berkmin [12] and zChaff [20].

In the strategy tree guessed above, several nodes of the tree could represent the same position in the game and the tree encodes the strategies from these

nodes separately. Since reachability games have zero-memory strategies, we need not guess separate strategies from these nodes. In the second reduction to Sat, we consider a variation where we essentially guess a *directed acyclic graph* of positions of the game which encode a strategy for the system and which witnesses the fact that the system wins the game. Given a parameter n on the size of such a witness set, our reduction checks whether the system has a strategy such that there is a set of positions bounded by n within which the system can force the game to be within and reach the goal. This is perhaps the more natural generalization of bounded model-checking to games.

We compare all the above methods and the different encodings described above using two examples that can be scaled. The first example is a pursuer-evader game where the objective is to guide a robot from one end of a grid to another while evading another slower robot that moves arbitrarily in the grid. Since our results show that Bdd methods outperform both Sat and Qbf methods by a large margin for this example, we consider in the second example a game which is known to be hard for Bdds (using the *swap* example from [19]). However, it turns out that Bdds still outperform the Sat and Qbf methods. We postpone a more detailed discussion of the results to the concluding section.

Our aim in this paper is to have a common platform to specify symbolic games so as to compare various symbolic techniques and evaluate them. The games we consider involve continuous interaction between the two players, as is common in most games studied in formal methods. The use of symbolic methods to solve problems related to games is not new. Symbolic methods have been proposed and studied in the area of planning in AI, for example, in conditional planning using Qbf methods [23] and for universal planning using Bdds [14] (see also [5]). However, we do not know of any comparative study of solving games using different symbolic approaches.

The paper is organized as follows. Section 2 lays out the precise definition of symbolic two-player reachability games. In Section 3 we outline two approaches using Bdds to solve games, one using Atl specifications in Mocha and the other using μ-calculus specifications in Mucke. Section 4 deals with solving bounded versions of the game problem, using reductions to satisfiability of Qbf and Sat formulas. For the Sat reduction we outline both the strategy-tree approach as well as the witness-graph approach. We present our experimental results for two game examples in Section 5 and conclude in Section 6.

2 Games

In this section, we define the required terminology. Let X be a finite set of variables. We write $X' = \{x' |\ x \in X\}$ for the set of primed variables of X. We denote by $Val(X)$ the set of all total functions that map every variable in X to a value in its domain. The set of all predicates over X is denoted by $\mathcal{P}(X)$. Given $p \in Val(X)$ and a predicate φ over $X = \{x_1, \cdots, x_n\}$, we write $\varphi[p] = \varphi[p(x_1)/x_1, \cdots, p(x_n)/x_n]$ for the truth value obtained by replacing

each variable $x_i \in X$ in φ with the value $p(x_i)$.

We model a *game* [24] between a *system* and its *environment* using a *game structure* $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ with the following components:

- X_S is a finite set of *variables* the system controls, and X_E is a finite set of *variables* the environment controls with $X_S \cap X_E = \emptyset$. We write $X = X_S \cup X_E$ for the set of system and environment variables, and $Q = Val(X)$ for the set of states of S.
- M_S is a finite set of *move* variables[2] which determine the next *move* of the system and M_E is a finite set of *move* variables which determine the next *move* of the environment. We assume $M_S \cap M_E = \emptyset$, $M_S \cap X = \emptyset$ and $M_E \cap X = \emptyset$.
- $T_S \in \mathcal{P}(X, M_S, X'_S)$ is a *transition predicate* for the system variables. For each $q \in Val(X)$, $m_S \in Val(M_S)$ and $q'_S \in Val(X'_S)$, if $T_S[q, m_S, q'_S] = true$ then q'_S is the next valuation of variables in X_S when the system picks the move m_S at the state q. Similarly, $T_E \in \mathcal{P}(X, M_E, X'_E)$ is a *transition predicate* for the environment variables. For each $q \in Val(X)$, $m_E \in Val(M_E)$ and $q'_E \in Val(X'_E)$, if $T_E[q, m_E, q'_E] = true$ then q'_E is the next valuation of variables in X_E when the environment picks the move m_E at the state q.

Now, we define a *game* $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with a game structure S, an *initial state*[3] $I \in Val(X)$, a *goal predicate* $\mathcal{G} \in \mathcal{P}(X)$ and a *safe predicate* $\mathcal{S} \in \mathcal{P}(X)$ where for each $q \in Val(X)$, if $\mathcal{G}[q] = true$ then the state q is in the *goal* region, and if $\mathcal{S}[q] = true$ then the state q is in the *safe* region. The game starts in the initial state and in every step, the system and the environment pick a move simultaneously and the state evolves according to this choice. If the goal region is reached then the system wins. If the current state is not in the safe region, the environment wins. Otherwise, the game continues forever.

For two states p and q, we say that q is the *successor* of p if there are $m_S \in Val(M_S)$ and $m_E \in Val(M_E)$ such that $T_S[p, m_S, q'_S] = true$, $T_E[p, m_E, q'_E] = true$ and $q = q_S \cup q_E$. We assume that there exists at least one successor at every state. A *path* of S is a finite or infinite sequence $\lambda = q_0, q_1, \cdots$ of states such that for all positions $i \geq 0$, q_{i+1} is a successor of q_i. For a path λ and a position $i \geq 0$, we use $\lambda[i]$ and $\lambda[0, i]$ to denote the i-th state of λ and the finite prefix $q_0, q_1, \cdots, q_i$ of λ, respectively. A *strategy* for the system is a function $f : Q^+ \rightarrow Val(M_S)$ which maps every nonempty finite state sequence $\lambda \in Q^+$ to a move $f(\lambda) \in Val(M_S)$. Given a strategy f, we define the *plays* of f to be the set $plays(f)$ of paths which are possible when the system follows the strategy f; that is, a path $\lambda = q_0, q_1, \cdots$ is in $plays(f)$ if for all positions $i \geq 0$, there are $m_S \in Val(M_S)$ and $m_E \in Val(M_E)$ such that $m_S = f(\lambda[0, i])$ and q_{i+1} is the $\langle m_S, m_E \rangle$ successor of q_i. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, a strategy

[2] In many examples, M_S and M_E will contain a single variable but in general, if a system has multiple components then there can be a move variable for each component.

[3] We can handle multiple start states by introducing a new state as an initial state with moves to all the start states.

f is a *winning strategy* in the game G if for all $\lambda = q_0, q_1, \cdots \in plays(f)$ such that $q_0 = I$, there exists a position $i \geq 0$ such that $\mathcal{G}[q_i] = true$ and for all positions $0 \leq j < i$, $\mathcal{S}[q_j] = true$. Finally, given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, the *game problem* is to check whether the system has a winning strategy in the game G.

Example 1.
Consider the game between an *evader* E and a *pursuer* P on an $n \times n$ grid as shown in Figure 1. The evader tries to reach the predefined *goal* position G without being caught by the pursuer. The evader chooses one amongst five moves: *up, down, left, right* and *stay* in every step. The pursuer, however, chooses one such move only in every odd step and it must stay stationary in every even step. Considering the evader as the system player and the pursuer as the environment player, we can define the game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ as follows.

First, we model the game structure by $S = (X_S, X_E, M_S, M_E, T_S, T_E)$.

- $X_S = \{x_e, y_e\}$ where x_e and y_e ranging over $\{0, \cdots, n-1\}$ are the x-y coordinates of the evader, and $X_E = \{x_p, y_p, clock\}$ where x_p and y_p ranging over $\{0, \cdots, n-1\}$ are x-y coordinates of the pursuer and *clock* ranging over $\{0, 1\}$ is a toggle specifying when the pursuer can change its position.
- $M_S = \{m_e\}$ and $M_E = \{m_p\}$ where m_e and m_p range over $\{up, down, left, right, stay\}$.
- $$T_S \equiv \begin{bmatrix} \big((x_e > 0) \wedge (m_e = left) \wedge (x'_e = x_e - 1) \wedge (y'_e = y_e)\big) \\ \vee \big((x_e < n-1) \wedge (m_e = right) \wedge (x'_e = x_e + 1) \wedge (y'_e = y_e)\big) \\ \vee \big((y_e < n-1) \wedge (m_e = up) \wedge (x'_e = x_e) \wedge (y'_e = y_e + 1)\big) \\ \vee \big((y_e > 0) \wedge (m_e = down) \wedge (x'_e = x_e) \wedge (y'_e = y_e - 1)\big) \\ \vee \big((m_e = stay) \wedge (x'_e = x_e) \wedge (y'_e = y_e)\big) \end{bmatrix}.$$

$$T_E \equiv \begin{bmatrix} \Big((clock = 1) \wedge \Big(\big((x_p > 0) \wedge (m_p = left) \wedge (x'_p = x_p - 1) \wedge (y'_p = y_p)\big) \\ \qquad \vee \big((x_p < n-1) \wedge (m_p = right) \wedge (x'_p = x_p + 1) \wedge (y'_p = y_p)\big) \\ \qquad \vee \big((y_p < n-1) \wedge (m_p = up) \wedge (x'_p = x_p) \wedge (y'_p = y_p + 1)\big) \\ \qquad \vee \big((y_p > 0) \wedge (m_p = down) \wedge (x'_p = x_p) \wedge (y'_p = y_p - 1)\big)\Big)\Big) \\ \vee \big((m_p = stay) \wedge (x'_p = x_p) \wedge (y'_p = y_p)\big) \end{bmatrix}$$

$$\wedge (clock' \neq clock).$$

$I \equiv \{x_e = 0, y_e = 0, x_p = 1, y_p = 3\}$ if the initial position of the evader is $(x = 0, y = 0)$ and the initial position of the pursuer is $(x = 1, y = 3)$. $\mathcal{G}$ is *true* if the x-y coordinates of the evader are same with the predefined goal position. $\mathcal{S}$ is *true* if the x-y coordinates of the evader are different from the pursuer's: $\mathcal{S} \equiv (x_e \neq x_p) \vee (y_e \neq y_p)$.

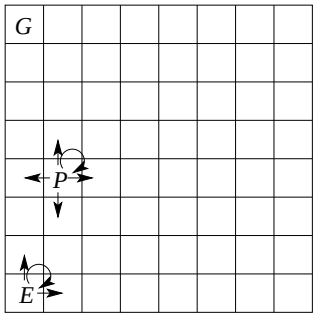

Fig. 1. Pursuit-evasion Game

Algorithm [Symbolic model checking for game problems]

Input: a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$.
Output: the answer to the model checking problem for the game.

$\rho := \mathit{False}$;
$\tau := \mathcal{G}$;
while $\tau \not\equiv \rho$ **do**
    $\rho := \rho \vee \tau$;
    $\tau := \mathit{Pre}^G(\rho) \wedge \mathcal{S}$;
od;
if $\rho(I)$ **then** return *true*;
else return *false*;

Fig. 2. Symbolic model checking algorithm for game problems

3 Solving games using BDDs

In this section, we solve games using binary decision diagrams (BDDs). The standard attractor-set method to solve games is a fix-point algorithm that can be implemented using BDDs. Figure 2 shows a symbolic model checking algorithm for our game problem, which manipulates state sets of S. Given a goal region and a safe region, we compute all states from which there is a winning strategy for the system. Note that the function Pre^G is different from the pre-image function of CTL model checkers. The function Pre^G, when given a predicate $\rho(X_S, X_E)$, returns a predicate $\mathit{Pre}^G(\rho) \in \mathcal{P}(X)$ for the set of states p such that from p, the system enforces the next state to satisfy ρ no matter how the environment behaves. Formally,

$$\mathit{Pre}^G(\rho) \equiv \exists M_S, X'_S . \forall M_E, X'_E . T_S(X, M_S, X'_S) \wedge (T_E(X, M_E, X'_E) \rightarrow \rho(X'_S, X'_E)).$$

In the algorithm, sets of states and the transition relation are represented by BDDs [8]. Both the ATL model checker and μ-calculus model checker use this algorithm.

ATL Model Checking

MOCHA[3] is a verification environment for modular verification for alternating-time temporal logic, which is a game logic extension of CTL.

Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, we specify a game structure S as reactive modules [3] where the system and its environment are described in separate modules, and specify $\mathcal{G}$ and $\mathcal{S}$ as an ATL formula using the *until* operator $\mathcal{U}$. The logic ATL admits a formula $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$, where ϕ and ψ are state predicates and A is a subset of players. The formula $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$ asserts that the players in A can cooperate to keep satisfying ϕ until satisfying ψ no matter how the remaining players behave. Considering A as the system, the semantics of $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$ is exactly same as the game problem. For Example 1, we specify the evader and the pursuer as separate modules, and specify the game property as the ATL specification, $\langle\langle Evader \rangle\rangle (safe \, \mathcal{U} \; goal)$. Then, we use symbolic ATL model checking of MOCHA, which implements the algorithm shown in Figure 2.

μ-calculus Model Checking

The μ-calculus [4] is propositional modal logic extended with the least fixpoint operator and is interpreted over Kripke structures. While μ-caculus model-checking can be seen to be equivalent to evaluating *infinite parity games* on finite graphs, the μ-calculus also trivially encodes solutions to reachability games. In our context, the μ-calculus formula:

$$\mu X.(goal \vee (safe \wedge \bigvee_{m_s \in Val(M_S)} \bigwedge_{m_e \in Val(M_E)} \langle m_s, m_e \rangle X))$$

computes the winning area for player S, as it stands for the least set X containing the goal configurations as well as those configurations from which the system can force a move into X.

Since least fixpoint computations can be performed symbolically, we can use symbolic μ-calculus model checkers to solve games using BDDS. The model-checker we consider is Biere's model checker MUCKE (μCKE) [7], which is developed with an aim to be a μ-calculus model checker that performs as well as symbolic model-checkers like SMV on the CTL fragment. MUCKE is a BDD-model checker optimized for the μ-calculus using techniques similar to those employed in model-checkers for CTL (like allocating fixed variable orderings for variables computing fixpoints, frontier set simplification, etc.).

When coding games into μ-calculus, we can also implement *early termination*, i.e. terminating the above fix-point computation when we reach an initial state. This can be encoded as:

$$\mu X.(goal \vee (\exists \bar{x} \in X : I\bar{x}) \vee (safe \wedge \bigvee_{m_s \in Val(M_S)} \bigwedge_{m_e \in Val(M_E)} \langle m_s, m_e \rangle X))$$

In the above, if an initial state is reached, the set X immediately gets set to the entire set of states and the fixpoint terminates.

4 Solving Bounded Games

Symbolic model checking [18] has been acknowledged as an efficient verification technique. Many symbolic model checkers use Bdds [8] as representations for sets of states and transition relation. However, the size of Bdds may increase exponentially as the number of variables.

Recently, a new type of model checking technique, *bounded model checking* with satisfiability solving [6,13], has led to promising results. In bounded model checking, given a transition system S, a temporal logic formula f and a user-supplied bound $k \in \mathbb{N}$, we construct a propositional formula $[\![S, f]\!]_k$ which is satisfiable if and only if the formula f is valid along some path of length k. Then, we solve the formula $[\![S, f]\!]_k$ using a Sat solver.

4.1 QBF Methods

For solving bounded games, we need more definitions. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, a strategy f and a bound k, $plays_k(f)$ is the set of plays of length k which are possible when the system follows the strategy f. A strategy f is a *k-winning strategy* in a game G if for all $\lambda = q_0, \cdots, q_k \in plays_k(f)$ are winning, i.e., there exists a position $0 \leq i \leq k$ such that $\mathcal{G}[q_i] = true$ and for all positions $0 \leq j < i$, $\mathcal{S}[q_j] = true$. The *bounded game problem* is, given a game G and a bound k, to check whether the system has a k-winning strategy in the game G. Consequently, we want to construct a boolean formula $\Phi^1_{G,k}$ which is satisfiable if and only if the system has a k-winning strategy in the game G.

Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ and a bound k, we denote, for every $0 \leq i \leq k$, the i-th copy of X, X_S, X_E by X^i, X^i_S, X^i_E, respectively. we divide I into I_S and I_E which are the initial values for X_S and X_E, respectively. However, unlike bounded model checking, we need alternations of existential quantification and universal quantification in order to solve a bounded game problem. Therefore, the formula $\Phi^1_{G,k}$ is a quantified boolean formula beginning with a prefix $\exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^{k-1}_S, M^{k-1}_S . \forall X^{k-1}_E, M^{k-1}_E . \exists X^k_S . \forall X^k_E$. $\Phi^1_{G,k}$ describes that there exists a series of system's moves to guarantee that for all series of environment's moves, the goal region is reached through the safe region as long as the environment proceeds according to the transition relation. $\Phi_{G,k}$ is as follows.

$$\Phi^1_{G,k} \equiv \exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^{k-1}_S, M^{k-1}_S . \forall X^{k-1}_E, M^{k-1}_E . \exists X^k_S . \forall X^k_E .$$
$$I_S(X^0_S) \wedge \phi_1 \wedge \Big((I_E(X^0_E) \wedge \psi_1) \rightarrow \rho \Big)$$

where,

- $\phi_1 \equiv \bigwedge_{i=0}^{k-1} T_S(X_i, M^i_S, X^{i+1}_S)$,
- $\psi_1 \equiv \bigwedge_{i=0}^{k-1} T_E(X_i, M^i_E, X^{i+1}_E)$ and

- $\rho \equiv \bigvee_{i=0}^{k} (\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j))$.

In the above formula $\Phi^1_{G,k}$, the subformulas, ϕ_1 and ψ_1, force that the next states along the path should obey the transition relation and ρ encodes reachability to the goal region through the safe region within k steps.

The total number of variables in $\Phi^1_{G,k}$ is $O(k \cdot N)$ where $N = |X \cup M_S \cup M_E|$, and the length of $\Phi^1_{G,k}$ (after some simplification) is $O(k \cdot (|T_S| + |T_E| + |\mathcal{G}| + |\mathcal{S}|) + |I_S| + |I_E|)$ where $|\mathcal{G}|, |\mathcal{S}|, |T_S|, |T_E|, |I_S|$ and $|I_E|$ are the lengths of formulas. In this expression, $k \cdot (|T_S| + |T_E|)$ is the dominant factor because $|T_S|$ and $|T_E|$ are quadratic in N, but $|\mathcal{G}|$ and $|\mathcal{S}|$ and is linear in N.

We define a new formula $\Phi^2_{G,k}$ which has three extra copies of the variables $X \cup M_S \cup M_E$, but which is shorter than the previous formula $\Phi^1_{G,k}$ since it has only one occurrence of T_S and T_E. The trick is to have an additional universal quantification after the k alternating quantifiers and to treat these as temporary variables and check that if they match the i^{th} and $(i+1)th$ copy of the original variables, then they satisfy the predicates T_S and T_E. Subsequently, the total number of variables in $\Phi^2_{G,k}$ is $O(k \cdot N)$ and the length of $\Phi^2_{G,k}$ (after some simplification) is $O(k \cdot (|\mathcal{G}| + |\mathcal{S}|) + |T_S| + |T_E| + |I_S| + |I_E|)$. $\Phi^2_{G,k}$ is given by:

$$\Phi^2_{G,k} \equiv \exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^k_S . \forall X^k_E . \forall Y, Y_M, Y', Z, Z_M, Z' .$$
$$I_S(X^0_S) \wedge \phi_2 \wedge \Big((I_E(X^0_E) \wedge \psi_2) \rightarrow \rho \Big)$$

where,

- $\phi_2 \equiv \bigvee_{i=0}^{k-1} ((X^i = Y) \wedge (M^i_S = Y_M) \wedge (X^{i+1}_S = Y')) \rightarrow T_S(Y, Y_M, Y')$,
- $\psi_2 \equiv \bigvee_{i=0}^{k-1} ((X^i = Z) \wedge (M^i_E = Z_M) \wedge (X^{i+1}_E = Z')) \rightarrow T_E(Z, Z_M, Z')$ and
- $\rho \equiv \bigvee_{i=0}^{k} (\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j))$.

We denote by *M1* the method which uses the first formula $\Phi^1_{G,k}$ to solve the game, and by *M2* the method which uses $\Phi^2_{G,k}$. We use QBF solvers such as SEMPROP [17], QUAFFLE [25] and QUBE [11] in order to solve the above quantified boolean formulas.

4.2 SAT Method using Strategy Tree

The bounded game problem is naturally translated to a QBF solving problem as we saw in Section 4.1 and we must use QBF solvers. However, several SAT solvers have recently shown promising results. In the next two subsections, we show how to translate the bounded game problem to a boolean formula only with existential quantification in order to use SAT solvers.

For translating the quantified formula for $\Phi^1_{G,k}$ in the previous section into a boolean formula, we need to eliminate universal quantification by introducing extra copies of variables in order to specify explicitly all cases without universal quantification; for example, $\forall x . \exists y . (x \wedge y) \equiv \exists y_1, y_2 . ((true \wedge y_1)) \wedge (false \wedge y_2))$. Figure 3 shows relations between successors and predecessors in QBF and

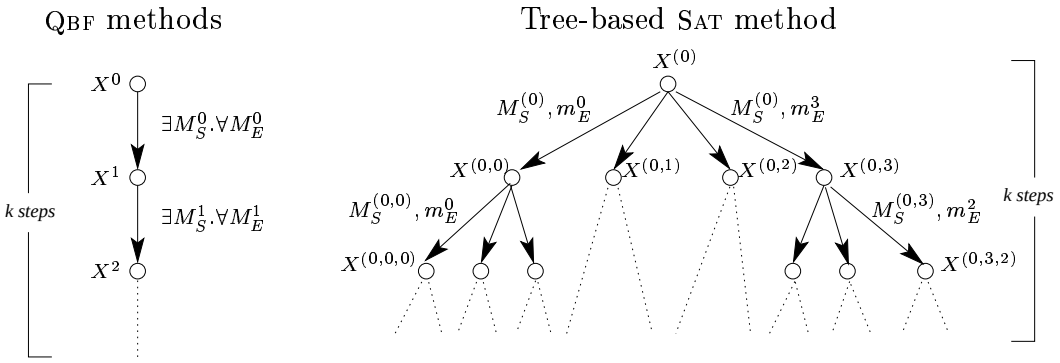

Fig. 3. Tree-based SAT Method

SAT methods. In tree-based SAT method, we introduce explicitly one copy of variables for every node in the tree. Thus, the number of copies is exponential in the bound k.

Every path of the tree-based SAT method corresponds to a play of length k and we just need to write a formula to check that the paths stay in the *safe* region until they reach the *goal* region, which we write as the formula $\Phi^3_{G,k}$.

The number of variables in $\Phi^3_{G,k}$ is $O(N \cdot m^k)$ where m is the maximum number of environment's moves and the length of $\Phi^3_{G,k}$ is $O(m^k \cdot (|T_S| + |T_E| + k \cdot (|\mathcal{S}| + |\mathcal{G}|)) + |I_S| + |I_E|)$. We then use BERKMIN [12] and ZCHAFF [20] in order to solve the boolean formula $\Phi^3_{G,k}$.

4.3 SAT Method using Witness Set

In the strategy-tree based SAT method, we constructed a tree which is a witness for a bounded game problem with a bound k. The tree, however, could have many identical states and we check the strategy from the identical states many times. In this section, we introduce a method that can generate a witness set with less copies of variables. The main idea is to construct a set which witnesses the fact that the system wins. Thus, given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ and a user supplied $n \in \mathbb{N}$, we generate a boolean formula $\Phi^4_{G,n}$ which is satisfiable if and only if we can generate a witness set with n states. First, we define $T_i(X, M_S, X')$ as a predicate for the next state when the environment's move is fixed. For each element m_i of the set $\{m_1, m_2, \cdots, m_{max}\}$ of the environment's moves, $T_i(X, M_S, X')$ is the predicate obtained from $T_S(X, M_S, X'_S) \wedge T_E(X, M_E, X'_E)$ (where $X' = X'_S \cup X'_E$) by replacing each of the variables $v \in M_E$ with the value $m_i(v)$. Now, we define a witness set for a bounded game problem as follows. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ and user supplied number n, $W = \{q_1, q_2, \cdots, q_n\}$ is a witness set for the game G if and only if

- for the initial predicate I of G, $I[q_1] = \mathit{true}$, and
- for each $q_i \in W$, $\mathcal{G}[q_i] = \mathit{true}$, or, $\mathcal{S}[q_i] = \mathit{true}$ and there exists a system

move m_S^i such that for each valid move m_j in the set $\{m_1, m_2, \cdots, m_{max}\}$ of environment moves at q_i, there exists $i < l \leq n$ such that $T_j[q_i, m_S^i, q_l] = \mathit{true}$.

The formula $\Phi^4_{G,n}$ for witness-based SAT method is as follows.

$$\Phi^4_{G,n} \equiv I(X^1) \wedge \bigwedge_{i=1}^{n} \Big(\mathcal{G}(X^i) \vee \Big(\mathcal{S}(X^i) \wedge \bigwedge_{j=1}^{max} (V_j(X^i) \rightarrow \bigvee_{l>i}^{n} T_j(X^i, M_S^i, X^l)) \Big) \Big)$$

where $V_j \in \mathcal{P}(X)$, for every $1 \leq j \leq max$, is a *validity predicate* for the environment: $V_j[q] = \mathit{true}$ if and only if m_j is valid environment's move at the state q. The definition of a witness set forces that every copy q_i which is not a goal, must have a transition to some q_l where l is strictly larger than i. Note that q_n must hence be a goal position and in fact the definition forces all plays encoded in the witness set to end in the goal. In the formula $\Phi^4_{G,n}$, the total number of variables is $O(n \cdot N)$ and the length of the formula is $O(mn^2 \cdot |T_j| + n \cdot (|\mathcal{G}| + |\mathcal{S}|) + |I|)$ where m is the maximum number of environment's moves. We again use BERKMIN and ZCHAFF in order to solve the boolean formula $\Phi^4_{G,n}$.

5 Experimental Results

We also consider a second example, which is known to be hard for BDDS [19].

Example 2.
The second example is *swap* introduced in [19]. We change the example into a game problem. There is an array $A[\ ]$ with n elements which are m-bit binary numbers. We assume that $n \leq 2^m$ so that all elements in the array can be distinct. Initially we have, for all $0 \leq i < n$, $A[i] := i$. At each step, the system chooses a direction between *left* and *right* and the environment chooses an index i, in the range $0, \cdots, n-1$; then the value of $A[i]$ is swapped with that of $A[(i-1) \bmod n]$ or $A[(i+1) \bmod n]$, according to the direction the system picked. The property we want to check is whether the system can eventually make $A[0]$ and $A[1]$ same no matter what the environment does (the system clearly loses).

We compare the methods we addressed using the Examples 1 and 2. For QBF methods, our program first generates a Boolean circuit [15] file, which is a more succinct format than CNF. Then we use BC2CNF [15] to translate the Boolean circuit into CNF. In the process, many intermediate variables are introduced. Finally, our program attaches quantification to the CNF file automatically and we use QBF solvers such as SEMPROP, QUAFFLE and QUBE to solve the CNF formula with quantification.

Also, for SAT methods we generate a Boolean circuit file and translate it to CNF using BC2CNF. We use the SAT solvers BERKMIN and ZCHAFF on the CNF

Table 1
The results for Example 1

Grid size	Bdd methods			Bounded methods				
	Mocha	Mucke		Step(k)	Qbf methods		Sat methods	
		Normal	Early		$M1$	$M2$	Tree	Witness
4×4	0 (12)	3 (7)	3 (3)	4 6 7 15 16	1 172 2030 – –	550 – – – –	0 0 0 17 *	818 (25)
8×8	6 (20)	3 (16)	3 (16)	4 5 15 16	76 32429 – –	– – – –	0 0 117 *	–
16×16	190 (35)	3 (32)	3 (32)	12 15 16	– – –	– – –	29 135 *	–
32×32	6493 (67)	5 (64)	5 (64)	12 15 16	– – –	– – –	58 531 *	–
256×256	–	373 (512)	100 (263)	8 12	– –	– –	– –	–
512×512	–	–	4024 (517)	8 12	– –	– –	– –	–

formula. All experiments were performed on a PC using a 1GHz Pentium III processor, 1.5GB memory and the Linux operating system.

The results for Example 1 are shown in Table 1 where the time shown is the execution time in seconds, '–' means that it did not complete in 10 hours, and * means the size of the input file was too large to execute (over 1GB). In Bdd methods, the number in parenthesis is the number of iterations taken to reach the fix-point while in the witness method, the number in parenthesis is the size of the witness set. For early termination results, the initial position of the purpuer was chosen as $(n/3, 3n/4)$ for the $n \times n$ grids. In this example, Mucke performed better than Mocha. For Qbf method $M1$, Qube (Ver. BJ1.0) worked best and for Qbf method $M2$, Semprop (Ver. 240202) showed the best result. For Sat methods, Berkmin worked best. The results in the table are the results for the tools that performed best. For this example, Bdd-based methods seem better than Qbf, and Sat-based methods seem better than Qbf-methods.

Table 2 shows the results for Example 2 where the Bdd method outperformed Qbf and Sat methods. Unlike Example 1, the Qbf method was better than the tree-based Sat method. This is perhaps because, in Example 2, the environment has n moves at every stage, which makes the strategy tree very large, while in Example 1, it has at most five moves at any stage.

Table 2
The results for Example 2

Array size	BDD method	Bounded methods		
	MOCHA	Step(k)	QBF method	SAT method
			$M1$	Tree
5	0 (5)	5 6 7	3 32 257	3 188 –
6	1 (6)	5 6 7	9 93 872	23 16718 –
7	9 (7)	5 6 7	22 841 3872	81 – –
8	77 (8)	5 6 7	41 1901 10746	1895 – –
9	518 (9)	5 6	– –	11764 –

6 Conclusions

We have presented various symbolic methods using BDDs, QBF-solvers and SAT-solvers to solve symbolically presented succinct games and evaluated them on two examples. This research is preliminary and one cannot draw hard conclusions yet. From the current results, however, it does seem that BDDs (especially MUCKE) outperform methods that use propositional solvers. The main problem with reduction to SAT seems to be the exponential blow-up in the reduction to game witnesses. Also, just reducing the size of the formula by making it more complex, seems to make SAT and QBF solvers perform worse than with a simple but larger encoding. If one could come up with a very small notion of a witness set for winning games, the propositional solvers may turn out to be more powerful.

There are several issues that are interesting for future study. First, most applications require to solve *partial information games* and it is not clear how to extend the methods to handle this. Also, once we know that the system indeed wins the game, we do not know how hard it is to extract a winning strategy of reasonable size from the above procedures.

Games have been recently used in the extraction of formal interfaces to software modules, in order to check consistency between software components [9]. It would be interesting to try out the above symbolic game solving techniques in such a domain.

Finally, McMillan has a technique to do unbounded model checking using SAT solvers, where SAT-solvers are exploited to manipulate sets of states stored as boolean formulas [19]. We plan to explore whether games can also be solved

using a similar approach.

References

[1] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *Proceedings of the Tenth International Conference on Concurrency Theory*, volume 1664 in LNCS, Springer, pp. 82–97, 1999.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.

[3] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, volume 1427 of LNCS, pages 521–525. Springer-Verlag, 1998.

[4] Arnold A. and Niwinski D. Rudiments of μ-calculus. Studies in Logic and the Foundations of Mathematics 146. 2001.

[5] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso *MBP: a Model Based Planner.* In *Proceedings of IJCAI-2001 workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[6] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 in LNCS, pages 193–207. Springer-Verlag, 1999.

[7] A. Biere. μcke - Efficient μ-calculus model checking. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, volume 1254 of LNCS, pages 468–471. Springer-Verlag, 1997.

[8] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[9] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of LNCS, pages 428–441. Springer-Verlag, 2002.

[10] A. Church. Logic, arithmetics, and automata. In *Proc. of the International Congress of Mathematicians, 1962*, pages 23–35, Institut Mittag-Leffler, 1963.

[11] E. Giunchiglia, M. Narizzano and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 364–369, 2001.

[12] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proc. of Design Automation and Test in Europe (DATE'02)*, pages 142–149, 2002.

[13] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P.N. Ashar. Learning from BDDs in SAT-based bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, 2003.

[14] R. Jensen and M. Veloso. OBDD-based Universal planning for synchronized agents in non-deterministic domains. Journal of Artificial Intelligence Research, 13, pages 189-226, 2000.

[15] Tommi Junttila. Boolean circuit package version 0.20. `http://www.tcs.hut.fi/~tjunttil/circuits/index.html`

[16] O. Kupferman, and M. Y. Vardi. Module checking. In *Proceeding of the 8th*

International Conference on Computer-Aided Verification, volume 1102 of LNCS, pages 75–86. Springer-Verlag, 1996.

[17] Reinhold Lets. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proc. of Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of LNCS, pages 160–175. Springer-Verlag, 2002.

[18] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[19] K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of LNCS, pages 250–264. Springer-Verlag, 2002.

[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[21] P.J.G. Ramadge, and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81-98, 1989.

[22] J. Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1192–1197, Morgan Kaufmann Publishers, 1999.

[23] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352,1999.

[24] W. Thomas. Infinite games and verification. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of LNCS, pages 58–64. Springer-Verlag, 2002.

[25] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design (ICCAD'02)*, 2002.