Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symbolic Computational Techniques for Solving Games ⋆

Rajeev Alur, P. Madhusudan, Wonhong Nam

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA.
e-mail: {alur, madhusudan, wnam}@cis.upenn.edu

Received: date / Revised version: date

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 either a reachability game of the form "*p until q*" (for state predicates p and q) or a safety game of the form "*Always p*".

For reachability games, the first technique employs symbolic fixed-point computation using ordered binary decision diagrams [9]. 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 also show how some of these techniques can be adopted to solve safety games. We compare the various approaches by evaluating them on two examples for reachability games, and on an interface synthesis example for a fragment of TinyOS [15] for safety games. We use existing tools such as MOCHA [4], MUCKE [7], SEMPROP [19], QUBE [12] and BERKMIN [13], and contrast the results.

⋆ Supported in part by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382. A preliminary version of this paper was presented at the *BMC'03: Workshop on Bounded Model Checking*, and appears in *Electronic Notes in Theoretical Computer Science* 89(4), 2003.

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 [11]. Games have since then become popular in formal methods with various applications including control of discrete event systems [23], realizability and synthesis, and model-checking μ-calculus formulae [26]. 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 [18], in compatibility checking of formal interfaces for modules [10], in approaches to compositional verification [1,3], and more recently, in applications to synthesizing dynamic interfaces for Java classes [2].

Research and related applications have led to a variety of game formulations such as infinite games on finite graphs, concurrent multi-player games and games on pushdown systems [26]. However, the simplest games that most solutions computationally rely on are the two-player reachability and safety games on a finite graph. A reachability 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. In a safety game, a winning strategy for the system must make sure the game avoids a set of bad states forever.

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 and safety games 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 input representation.

The standard attractor-set approach to solve reachability games is a simple fixed-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 symbolic model checker especially optimized to handle μ-calculus formulas.

In order to use propositional solvers for reachability games, 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 [19], QUAFFLE [28] and QUBE [12] to decide 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 reachability 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 [13] and zCHAFF [22].

In the strategy tree above, several nodes of the tree may 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 that encodes a strategy for the system and that witnesses the fact that the system wins the game. Given a parameter n to bound the size of such a witness set, our reduction checks if there exists a set of at most n positions such that the system can force the game to be within this set 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 [21]). However, it turns out that BDDs still outperform the SAT and QBF methods.

We then turn to solving safety games, for which the symbolic techniques using BDDs are again straightforward. However, since in a safety game the objective is to stay away from a set of bad states forever, strategies that bound the length of plays by a constant are not interesting. However, the idea of guessing a bounded witness set generalizes with the modification that cycles are now permitted, and we are guessing a strategy that can be encoded as a bounded automaton. To study these two approaches, we apply these two methods to synthesize an interface for the radio layer of TinyOS [10]. Synthesizing interfaces can be reformulated as a safety game problem and hence is a more realistic example for evaluating games in the formal methods context. In this setting also, the symbolic approach using BDDs outperform the SAT-based strategy synthesis technique.

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 [25] and for universal planning using BDDs [16] (see also [6]). More recently, in [27], the authors investigate solving infinite games for synthesizing controllers using BDDs. 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 reachability 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 reachability 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. Section 5 outlines the approaches we follow for safety games. We present our experimental results for the two reachability games and the Tiny OS interface synthesis example in Section 6, and Section 7 contains some concluding remarks.

2 Games

In this section, we define the required terminology. Let X be a finite set of variables. Each variable x will be associated with a finite domain D_x; in the sequel the domain for each variable will be either evident or stated explicitly. We write $X' = \{x' \mid x \in X\}$ for the set of primed variables of X; the domain of a primed variable x' will be the same as that of x. We denote by $Val(X)$ the set of all total functions that map every variable x in X to a value in its domain D_x. If X is a set of variables, then a predicate over X is a boolean combination of relations over domains, and is typically an expression constructed using variables and constants. We denote the set of all predicates over X as $\mathcal{P}(X)$.

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* 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 *system move variables*[1] that determine the next move of the system and M_E is a finite set of *environment move variables* that 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.

We define a *reachability game* as a tuple $G = \langle S, I, \mathcal{G}, \mathcal{S}\rangle$ with a game structure S, an *initial state*[2] $I \in Val(X)$, 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 moves $m_S \in Val(M_S)$ 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 $f : Q^+ \to Val(M_S)$ 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, $plays(f)$, to be the set of paths that 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.

A *zero-memory* strategy is a strategy that depends only on the last state of the play, i.e. a strategy f such that $f(\lambda q) = f(\lambda' q)$, for every $\lambda, \lambda' \in Q^*$.

Given a reachability game $G = \langle S, I, \mathcal{G}, \mathcal{S}\rangle$, a strategy 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 *reachability game problem* is to check whether the system has a winning strategy in the game G. We postpone the definition of *safety games* to Section 5.

Example 1.

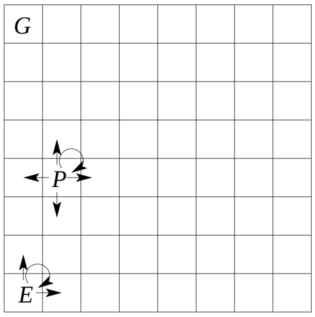
Consider the reachability game between an *evader E* and a *pursuer P* on an $n \times n$ grid as shown in Figure 1. The evader tries to reach the predefined *goal* position without being caught by the pursuer. The evader chooses one amongst five moves: *up, down, left, right* and *stay* in every step. 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 shown in the figure.

3 Solving Reachability Games Using BDDs

In this section, we solve reachability games using binary decision diagrams (BDDs). The standard attractor-set method to solve games is a fixed-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

[1] 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.

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



The game is $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, where $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ is given by:

- $X_S = \{x_e, y_e\}$ where x_e and y_e ranging over $\{0, \cdots, n-1\}$ are the x-y coordinates of the evader, and $X_E = \{x_p, y_p, \mathit{clock}\}$ 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 $\{\mathit{up}, \mathit{down}, \mathit{left}, \mathit{right}, \mathit{stay}\}$.
-
$$T_S \equiv \left[\begin{array}{l} \Big((x_e > 0) \wedge (m_e = \mathit{left}) \wedge (x'_e = x_e - 1) \wedge (y'_e = y_e)\Big) \vee \Big((x_e < n-1) \wedge (m_e = \mathit{right}) \wedge (x'_e = x_e + 1) \wedge (y'_e = y_e)\Big) \\ \vee\Big((y_e < n-1) \wedge (m_e = \mathit{up}) \wedge (x'_e = x_e) \wedge (y'_e = y_e + 1)\Big) \vee \Big((y_e > 0) \wedge (m_e = \mathit{down}) \wedge (x'_e = x_e) \wedge (y'_e = y_e - 1)\Big) \\ \vee\Big((m_e = \mathit{stay}) \wedge (x'_e = x_e) \wedge (y'_e = y_e)\Big) \end{array}\right].$$
-
$$T_E \equiv \left[\begin{array}{l} \Bigg((\mathit{clock} = 1) \wedge \Bigg(\Big((x_p > 0) \wedge (m_p = \mathit{left}) \wedge (x'_p = x_p - 1) \wedge (y'_p = y_p)\Big) \\ \qquad \vee\Big((x_p < n-1) \wedge (m_p = \mathit{right}) \wedge (x'_p = x_p + 1) \wedge (y'_p = y_p)\Big) \\ \qquad \vee\Big((y_p < n-1) \wedge (m_p = \mathit{up}) \wedge (x'_p = x_p) \wedge (y'_p = y_p + 1)\Big) \\ \qquad \vee\Big((y_p > 0) \wedge (m_p = \mathit{down}) \wedge (x'_p = x_p) \wedge (y'_p = y_p - 1)\Big)\Bigg)\Bigg) \\ \vee\Big((m_p = \mathit{stay}) \wedge (x'_p = x_p) \wedge (y'_p = y_p)\Big) \end{array}\right] \wedge (\mathit{clock}' \neq \mathit{clock}).$$
- If the initial position of the evader is $(x = 0, y = 0)$ and the initial position of the pursuer is $(x = 1, y = 3)$, then $I \equiv ((x_e = 0) \wedge (y_e = 0) \wedge (x_p = 1) \wedge (y_p = 3) \wedge (\mathit{clock} = 0))$. $\mathcal{G}$ is *true* if the x-y coordinates of the evader coincide 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

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,

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

In the algorithm, sets of states and the transition relation are represented by BDDs [9]. Both the ATL model checker and μ-calculus model checker use this algorithm.

Algorithm [Symbolic model checking for solving reachability games]
Input: a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$.
Output: the answer to the reachability game.
$\rho := \mathit{False}$;
$\tau := \mathcal{G}$;
while $\tau \not\rightarrow \rho$ **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 algorithm for reachability games

ATL Model Checking

MOCHA [4] is a verification environment for modular verification against specifications written in 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 in the modeling language *reactive modules* [4] where the system and its environment are described as separate modules, and specify the desired winning condition 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 reachability game problem. For Example 1, we specify the evader and the pursuer as separate modules, and specify the reachability game property as the ATL specification, $\langle\langle \mathit{Evader}\rangle\rangle(\mathit{safe}\ \mathcal{U}\ \mathit{goal})$. Then, we use symbolic ATL model checking of MOCHA, which implements the algorithm shown in Figure 2.

μ-calculus Model Checking

The μ-calculus [5] is propositional modal logic extended with the least fixed-point operator and is interpreted over Kripke structures. While μ-calculus 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:

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

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

Since least fixed-point 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 (such as allocating fixed variable orderings for variables computing fixed-points, frontier set simplification, etc.).

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

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

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

4 Solving Bounded Reachability Games

Symbolic model checking [20] has been acknowledged as an efficient verification technique. Many symbolic model checkers use BDDs [9] as representations for sets of states and transition relation. However, the size of BDDs may increase exponentially as the number of variables, and the memory requirements during the fixed-point computation are unpredictable.

Recently, a new model checking technique, *bounded model checking* using boolean satisfiability solvers [8,14], 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 which is satisfiable if and only if the formula f is violated along some path of length k in S.

4.1 QBF Methods

Given a reachability game $G = \langle S, I, \mathcal{G}, \mathcal{S}\rangle$, a strategy f and a bound k, $\mathit{plays}_k(f)$ 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 reachability game G if for all $\lambda = q_0, \cdots, q_k \in \mathit{plays}_k(f)$ are winning, i.e., there exists a position $0 \leq i \leq k$ such that $\mathcal{G}[q_i] = \mathit{true}$ and for all positions $0 \leq j < i$, $\mathcal{S}[q_j] = \mathit{true}$. The *bounded reachability game problem* is, given a reachability game G and a bound k, to check whether the system has a k-winning strategy in the reachability 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 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 describing that there exists a series of system's moves to guarantee that for all corresponding environment's moves, the goal region is reached through the safe region as long as the environment proceeds according to the transition relation. $\Phi^1_{G,k}$ is defined as:

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

where,

- $\phi_1 \equiv \bigwedge_{i=0}^{k-1} T_S(X_i, M_S^i, X_S^{i+1})$,
- $\psi_1 \equiv \bigwedge_{i=0}^{k-1} T_E(X_i, M_E^i, X_E^{i+1})$ and
- $\rho \equiv \bigvee_{i=0}^{k} \big(\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j)\big)$.

In the above formula $\Phi^1_{G,k}$, the subformulas, ϕ_1 and ψ_1 force the consecutive states along the path to 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 $O\big(k \cdot (|T_S| + |T_E| + |\mathcal{G}| + |\mathcal{S}|) + |I_S| + |I_E|\big)$ where $|\mathcal{G}|, |\mathcal{S}|, |T_S|, |T_E|, |I_S|$ and $|I_E|$ are the lengths of the respective formulas. In this expression, $k \cdot (|T_S| + |T_E|)$ is the dominant factor because $|T_S|$ and $|T_E|$ are quadratic in N while $|\mathcal{G}|$ and $|\mathcal{S}|$ are 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 $O\big(k \cdot (|\mathcal{G}| + |\mathcal{S}|) + |T_S| + |T_E| + |I_S| + |I_E|\big)$. $\Phi^2_{G,k}$ is given by:

$$\Phi^2_{G,k} \equiv \exists X_S^0, M_S^0 . \forall X_E^0, M_E^0 . \cdots \exists X_S^k . \forall X_E^k .$$
$$\forall Y, Y_M, Y', Z, Z_M, Z' .$$
$$I_S(X_S^0) \wedge \phi_2 \wedge \Big(\big(I_E(X_E^0) \wedge \psi_2\big) \rightarrow \rho\Big)$$

where,

- $\phi_2 \equiv \bigvee_{i=0}^{k-1} \big((X^i = Y) \wedge (M_S^i = Y_M) \wedge (X_S^{i+1} = Y')\big) \rightarrow T_S(Y, Y_M, Y')$,
- $\psi_2 \equiv \bigvee_{i=0}^{k-1} \big((X^i = Z) \wedge (M_E^i = Z_M) \wedge (X_E^{i+1} = Z')\big) \rightarrow T_E(Z, Z_M, Z')$ and
- $\rho \equiv \bigvee_{i=0}^{k} \big(\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j)\big)$.

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 [19], QUAFFLE [28] and QUBE [12] in order to decide these quantified boolean formulas.

4.2 SAT Method using Strategy Trees

The bounded reachability 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 reachability game problem to a boolean formula that has only existential quantification, in order to use SAT solvers.

For translating the quantified formula for $\Phi^1_{G,k}$ in the previous section into an existentially quantified boolean formula, we need to eliminate universal quantification by introducing extra copies of variables in order to specify explicitly all cases the universal quantification can quantify over; for example, $\forall x . \exists y . (x \wedge y) \equiv \exists y_1, y_2 . \big((\mathit{true} \wedge y_1)\big) \wedge (\mathit{false} \wedge y_2)\big)$. Figure 3 shows relations between successors and predecessors in the QBF and SAT methods, in the case where the environment has four choices of moves in each step. In the 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 a boolean 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 [13] and ZCHAFF [22] in order to check if the boolean formula $\Phi^3_{G,k}$ is satisfiable.

4.3 SAT Method using Witness Sets

In the strategy-tree based SAT method, we constructed a tree which is a witness for a k-bounded reachability game problem. The tree, however, could have many identical states and we check for a strategy from these identical states independently. Since reachability games have zero-memory strategies, we need not guess separate strategies from these nodes. 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 reachability 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. For each element m_i of the set $\{m_1, m_2, \cdots, m_{max}\}$ of the environment's moves, define $T_i(X, M_S, X')$ to be 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 reachability game problem as follows. Given a reachability 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 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 $T_j[q_i, m_S^i, q_l] = \mathit{true}$.

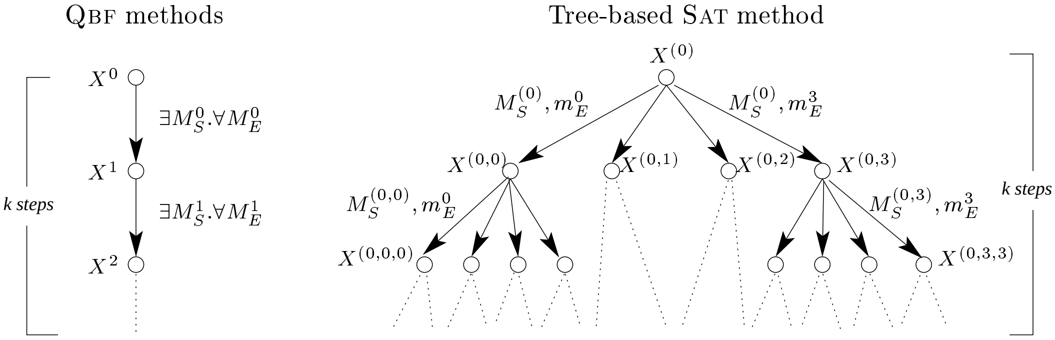


Fig. 3. Tree-based SAT Method

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

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

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 move at the state q. The definition of a witness set forces every copy q_i that is not a goal-state to 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 moves. We again use BERKMIN and zCHAFF in order to check if the boolean formula $\Phi^4_{G,n}$ is satisfiable.

5 Solving Safety Games

We now describe the two methods we use to solve safety games. A *safety game* is a tuple $G = \langle S, I, \mathcal{S} \rangle$, where the components are exactly as in a reachability game (note that the goal-predicate is missing). The notions of strategies for the system and the set *plays*(f) are as before. For a safety game $G = \langle S, I, \mathcal{S} \rangle$, a strategy f is *winning* if for all $\gamma = q_0, q_1, \cdots \in \mathit{plays}(f)$ with $q_0 = I$, for all states q_i, $\mathcal{S}(q_i) = \mathit{true}$.

A safety game $G = \langle S, I, \mathcal{S} \rangle$ can be solved using the related reachability game $G' = \langle S, I, \mathcal{G}, \mathcal{S}' \rangle$ where $\mathcal{G} = \neg \mathcal{S}$ and $\mathcal{S}' \equiv \mathit{true}$, and where the players' roles are reversed. More precisely, the system has a winning strategy in G if and only if the environment has a winning strategy in G', where the environment is required to force the game to the unsafe region. The game can hence be solved by reversing the roles of the players, and solving G'. Consequently, the BDD-based approach easily extends to solving safety games.

The restriction to bounded length plays, however, is not useful for safety winning conditions. Though it is true that the system does win the safety game provided it can remain safe for k steps, where k is the number of positions in the game, this value of k is too large (as it is the state-space of the game) and impractical.

The witness based technique (of Section 4.3) however does extend to the safety setting. The witness set described for reachability games is essentially a strategy automaton that encodes a winning strategy for the system. For safety games, we can define a similar strategy automaton that encodes a set W of safe states (which includes the initial state) such that from any state q in W, there is a system move m_S such that for every (valid) environment move m_E, the state from q on (m_S, m_E) belongs to W as well. Note that in this setting, we do not need a notion of progress towards the goal states and hence we do not require the priorities, encoded by indices, to increase as in the reachability case.

Formally, given a safety game $G = \langle S, I, \mathcal{S} \rangle$ and a user supplied $n \in \mathbb{N}$, we generate a boolean formula $\Phi^5_{G,n}$ which is satisfiable if and only if we can generate a strategy automaton with n states. As in the reachability case, for each element m_i of the set $\{m_1, m_2, \cdots, m_{max}\}$ of the environment's moves, let $T_i(X, M_S, X')$ be the predicate obtained from $T_S(X, M_S, X'_S) \wedge T_E(X, M_E, X'_E)$ by replacing each of the variables $v \in M_E$ with the value $m_i(v)$. Now, we define a strategy automaton as follows. Given a safety game $G = \langle S, I, \mathcal{S} \rangle$ 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 strategy automaton for the game G if and only if

- for the initial predicate I of G, $I[q_1] = \mathit{true}$, and
- for each $q_i \in W$, $\mathcal{S}[q_i] = \mathit{true}$ and there exists a system move m^i_S 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 $1 \leq l \leq n$ such that $T_j[q_i, m^i_S, q_l] = \mathit{true}$.

The formula $\Phi^5_{G,n}$ is:

$$\Phi^5_{G,n} \equiv I(X^1) \wedge \bigwedge_{i=1}^{n} \Big(\mathcal{S}(X^i) \wedge$$

$$\bigwedge_{j=1}^{max} \left(V_j(X^i) \rightarrow \bigvee_{1\leq l\leq n} T_j(X^i, M_S^i, X^l)\right)\Big)$$

where $V_j \in \mathcal{P}(X)$, for every $1 \leq j \leq max$, is the validity predicate for the environment: $V_j[q] = true$ if and only if m_j is a valid environment move at the state q. The bounds on the size of $\Phi^5_{G,n}$ are same as the ones for $\Phi^4_{G,n}$, and SAT solvers are used to check its satisfiability.

6 Experiments

Apart from the pursuer-evader game, we consider a second example for reachability games, which is known to be hard for BDDs [21]. Then we present a third example which is a safety game that corresponds to interface synthesis.

Example 2.

The second example is *swap* introduced in [21]. We change the example into a reachability game problem. There is an array $A[\ ]$ with n elements that 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 for reachability games using the Examples 1 and 2. For QBF methods, our program first generates a Boolean circuit [17] file, which is a more succinct format than CNF. Then we use BC2CNF [17] to translate the Boolean circuit into a CNF formula. 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 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 the experiment 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 fixed-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 pursuer 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) gave 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 again outperformed the 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.

Example for safety games: Interface synthesis for a module in TinyOS

In this section, we apply the safety games to solve the problem of *interface synthesis* on an abstraction of a piece of software code. Apart from illustrating this application of games, this example illustrates how solving games using BDDs and the strategy-automaton based approach using SAT perform in more realistic examples than those discussed above.

The Tiny microthreading Operating System (TinyOS) [15] is an event-driven operating system for supporting operations required by networked sensors. Figure 4 illustrates a communication layer of a network where packets are sent and received through a low-power radio channel. The higher levels send commands to lower layers; events originate at the lowest hardware layer and propagate up.

We modeled the Radio Byte and RFM components in MOCHA, treating the upper level that issues commands as the environment and handling the generation of events in the hardware as nondeterministic system events.

The code of TinyOS is from an early version; Figure 4 gives a schematic view of the components of Radio Byte and RFM and how they communicate. The Radio Byte layer handles requests to send data from the upper level and communicates at a bit level to the lower RFM modules. If the radio channel detects messages, an interrupt causes an event that RFM propagates to the Radio Byte layer which in turn signals to the upper layer and manages the receiving of the channel data. The radio can also be turned to low-power mode where messages cannot be received or sent (to conserve power) and woken up when required.

These modules were also isolated and studied in [10] where the authors discovered an incompatibility error. It turns out that turning the radio to low-power mode does not set the RFM components to low-power and the

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

Table 1. The results for Example 1

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

Table 2. The results for Example 2

waking up phase is hence unavailable in RFM after a low-power command.

Our goal in this experiment is not to check for compatibility, but rather to generate the most general environment that can issue commands to the radio modules such that all issued commands can be correctly processed by the modules. In other words, we want to synthesize the most general assumption these modules make on the interface that connects them to the rest of the operating system. For example, since turning the radio to low-power results eventually in an unavailability, a correct interface must capture this by demanding that the upper layer never turns the radio to low-power mode.

This problem can be set up naturally as a safety game problem: the components Radio Byte and RFM constitute one player and the upper level (environment) takes the role of the second player. The problem of designing the most general interface can then be stated as follows: find the most general strategy for the environment (player 0) to issue commands such that any command issued can be handled by the radio modules (player 1). Interrupts generated by the hardware (which determine when bits succeed in getting sent and whether information is being received by the channel) are nondeterministic—the environment must meet its goal of issuing available commands no matter how these interrupts are generated.

We model an abstraction of the code for the radio modules Radio Byte and RFM. In particular, various details such as timing issues, encoding of data for error correction, byte-to-bit conversions, etc., are abstracted away. Only the control flow (and the state variables relevant to it) are retained, and this abstraction causes nondeterminism in the model.

We implemented the safety game using MOCHA to evaluate the BDD-based approach. The MOCHA model has 29 variables. Unavailability of commands was modeled by setting a variable "*Error*" to true. The game problem captured by the specification "$\langle\langle Env \rangle\rangle G(\neg Error)$"

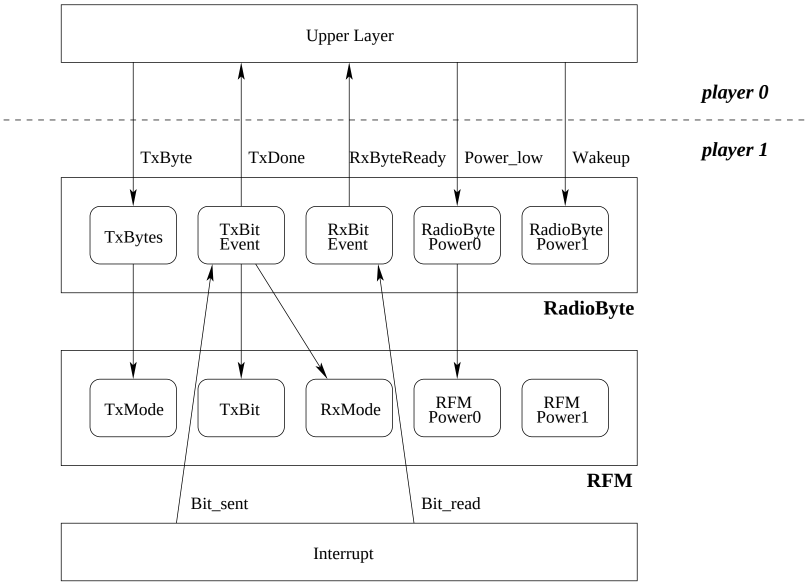


Fig. 4. The radio layer in TinyOS

("does the environment have a strategy such that *Error* is never set to true"), turned out to be true and was checked within one second.

The model however does not satisfy the CTL property "$AG(\neg Error)$" as the run where the environment turns the radio to low-power and then wakes it up causes an error. This check could be done in one second by MOCHA.

The winning strategy for the environment in the game always sends data on the channel and receives data when it comes on the channel, but never turns the radio to low-power.

We also implemented the above safety game problem using the strategy-automaton based technique using SAT. However, with the complete model, the SAT tools could not generate a winning strategy automaton. We then experimented with a sub-model of the TinyOS module where the receiving of events is disabled. For this model, the BERKMIN tool succeeded in synthesizing a safe interface having 13 states (the CNF formula corresponding to this instantiation had 45,645 variables and 227,889 clauses and the tool took 318 seconds to finish).

In summary, the BDD-based approach was faster and was able to handle the full model.

7 Conclusions

We have presented various symbolic methods using BDDs, QBF-solvers and SAT-solvers to solve symbolically presented succinct games and evaluated them on three examples. One of these examples was a realistic verification example, an interface synthesis example for a TinyOS module.

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.

Finally, McMillan has developed a technique to perform unbounded model checking using SAT solvers, where SAT-solvers are exploited to manipulate sets of states stored as boolean formulas [21]. It would be interesting to to explore whether games can 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, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. To appear in *Proceedings of the 32nd Symposium on Principles of Programming Languages*, 2005.

3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.
4. R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the 10th Int'l Conf. on Computer-Aided Verification*, LNCS 1427, pp. 521–525. Springer-Verlag, 1998.
5. A. Arnold and D. Niwinski. Rudiments of μ-calculus. Studies in Logic and the Foundations of Mathematics 146. 2001.
6. 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.
7. A. Biere. μcke - Efficient μ-calculus model checking. In *Proc. of the 9th Int'l Conf. on Computer-Aided Verification*, LNCS 1254, pp. 468–471. Springer-Verlag, 1997.
8. 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.
9. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
10. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proc. of the 14th Int'l Conf. on Computer-Aided Verification*, LNCS 2404, pp. 428–441. Springer-Verlag, 2002.
11. A. Church. Logic, arithmetics, and automata. In *Proc. of the International Congress of Mathematicians, 1962*, pages 23–35, Institut Mittag-Leffler, 1963.
12. 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.
13. 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.
14. 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.
15. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.
16. 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.
17. T. Junttila. Boolean circuit package version 0.20. `http://www.tcs.hut.fi/~tjunttil/circuits/index.html`
18. O. Kupferman and M. Y. Vardi. Module checking. In *Proc. of the 8th Int'l Conf. on Computer-Aided Verification*, LNCS 1102, pp. 75–86. Springer-Verlag, 1996.
19. R. 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.
20. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
21. K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. of the 14th International Conf. on Computer-Aided Verification*, LNCS 2404, pages 250–264. Springer-Verlag, 2002.
22. 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.
23. P.J.G. Ramadge, and W.M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77:81-98, 1989.
24. 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.
25. J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352,1999.
26. W. Thomas. Infinite games and verification. In *Proc. of the 14th Int'l Conf. on Computer-Aided Verification*, LNCS 2404, pp. 58–64. Springer-Verlag, 2002.
27. N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proc. of the 8th Int'l Conf. on the Implementation and Application of Automata, CIAA'03*, LNCS 2759, pages 11-22. Springer, 2003.
28. 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.