SAT-based Strategy Extraction in Reachability Games*

Niklas Een
UC Berkeley

Alexander Legg
NICTA† and UNSW Australia

Nina Narodytska
University of Toronto and NICTA

Leonid Ryzhyk
Carnegie Mellon University

Abstract

Reachability games are a useful formalism for the synthesis of reactive systems. Solving a reachability game involves (1) determining the winning player and (2) computing a winning strategy that determines the winning player's action in each state of the game. Recently, a new family of game solvers has been proposed, which rely on counterexample-guided search to compute winning sequences of actions represented as an abstract game tree. While these solvers have demonstrated promising performance in solving the winning determination problem, they currently do not support strategy extraction. We present the first strategy extraction algorithm for abstract game tree-based game solvers. Our algorithm performs SAT encoding of the game abstraction produced by the winner determination algorithm and uses interpolation to compute the strategy. Our experimental results show that our approach performs well on a number of software synthesis benchmarks.

1 Introduction

Two-player games are a useful formalism for synthesis and analysis of reactive systems (Bloem et al. 2007; Cassez et al. 2009; Ryzhyk et al. 2014). A two-player game involves two parties, referred to as the *controller* and the *environment*, which perform actions over a finite state space of the game. The controller aims to satisfy a predefined game *objective*, while the environment tries to prevent it from doing so.

We consider games with *reachability* objectives, where the controller must force the system into a given *goal* region. Consider, for example, the problem of synthesising the packet transmission function of a network controller device driver. The problem can be formulated as a game played over the state space of the device automaton, where controller actions model software commands to the device and environment actions model non-deterministic hardware events, such as error conditions. The game objective is to force the device into a state where it successfully completes packet transfer (Ryzhyk et al. 2014).

*This research is supported by a grant from Intel Corporation.

†NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Game-based synthesis involves solving two related problems: (1) *determining the winner* and (2) *generating a winning strategy*. The former identifies the player who can always win the game by choosing correct actions in every state (any reachability game has a unique winner); the latter computes a strategy function which maps states of the game into winning actions. The strategy can then be converted into a hardware circuit or a software program.

The traditional approach to game solving iteratively constructs a sequence of state sets from which the controller can win the game in $1, 2, \ldots$ rounds, until a fixed point is reached. Given this sequence, the strategy can be extracted using one of several existing techniques (Lonsing and Biere 2010; Jiang, Lin, and Hung 2009; Ehlers, Könighofer, and Hofferek 2012). Both parts of this two-step process are carried out symbolically using BDD, SAT or QBF-based methods. All three approaches suffer from similar performance problems. In particular, the winner determination step becomes inefficient if there does not exist a compact BDD or CNF representation of the winning sets or if such a representation is hard to compute. Furthermore, the strategy extraction step starts with an intermediate strategy representation that defines a potentially very large number of winning transitions in every state. Determinising such a strategy proved costly (Bloem et al. 2014a).

To address these limitations, a new type of SAT and QBF-based game solvers was proposed recently (Janota et al. 2012; Narodytska et al. 2014). Instead of computing winning regions, they use counterexample-guided search to compute winning *sequences of actions* represented as a compact *abstract game tree*. These solvers proved to be efficient at solving the winner determination problem on several families of benchmarks where the traditional approach does not scale (Narodytska et al. 2014).

In this work, we focus on the strategy generation problem, which is crucial for practical application of game solvers. Our contribution is two-fold:

1. We propose the first strategy generation method for game solvers based on abstract game trees, thus enabling the use of such solvers as part of a complete synthesis methodology.

2. We demonstrate that abstract game trees enable low-overhead strategy generation, thus providing additional

compelling motivation for further research on this new class of solvers. Our method reuses game abstractions computed by the winner determination algorithm and, as a result, introduces low computational overhead on top of the winner determination step.

2 Background

SAT and QBF A satisfiability problem ϕ consists of a set of Boolean variables $\mathcal{X}$ and a set of clauses $\mathcal{C} = \{C_1, \ldots, C_m\}$. A clause is a disjunction of literals, where a literal is either a variable $x_i \in \mathcal{X}$ or its negation $\overline{x_i}$. We denote the set of variables in the formula ϕ by $vars(\phi)$.

A Quantified Boolean Formula (QBF) is an extension of a satisfiability problem with quantifiers with the standard semantics. A formula $\forall x.\phi$ is satisfied iff $\phi\mid_{x=0} \wedge \phi\mid_{x=1}$ is satisfied. Similarly, a $\exists x.\phi$ is satisfied iff $\phi\mid_{x=0} \vee \phi\mid_{x=1}$ is satisfied. We assume that QBFs are in closed prenex form: $Q_1 X_1 \ldots Q_n X_n \phi$, where $Q_i \in \forall, \exists$ and $Q_i \neq Q_{i+1}$ and $X_i \cap X_j = \emptyset$, $i \neq j$. The propositional part ϕ of a QBF is called the *matrix* and the rest—the *prefix*.

Interpolants Let A and B be propositional formulas such that $A \wedge B = \bot$. Then there exists a propositional formula $\mathcal{I}$, called an *interpolant* of A and B such that $A \implies \mathcal{I}$ and $\mathcal{I} \wedge B = \bot$ and $vars(\mathcal{I}) = vars(A) \cap vars(B)$. An interpolant can be efficiently obtained from a proof of unsatisfiability of $A \wedge B$ produced by a SAT solver (Pudlak 1997).

Games and Strategies A reachability game $\mathcal{G} = \langle S, L_c, L_u, O, \delta \rangle$ consists of a set of states S, controllable actions L_c, uncontrollable actions L_u, a set $O \in 2^S$ of goal states, and a transition function $\delta : (S, L_c, L_u) \to S$. The game proceeds in a sequence of rounds, starting from the initial state. In each round, the controller picks an action $\mathbf{c} \in L_c$. The environment responds by picking an action $\mathbf{u} \in L_u$, and the game transitions to a new state $\delta(\mathbf{s}, \mathbf{c}, \mathbf{u})$.

Example 1 *Consider a game whose transition function δ is defined by the DFA in Figure 1(a). In the first round the controller stays idle (i.e., executes a special idle action i) and the environment chooses a number 1 or 2. The controller must match 1 by playing a in the second round and 2 by playing b and c in the second and third rounds. The goal consists of a single state s_4. Hence, $S = \{s_0, \ldots, s_4\}, L_c = \{i, a, b, c\}, L_u = \{0, 1, 2\}, O = \{s_4\}$. We make a simplifying assumption here that each player chooses one of the actions available in the current state of the game automaton, e.g., the controller must play i in the first round even though the set of all possible actions is $\{i, a, b, c\}$.* □

For a set $I \subseteq S$ and an action $a \in L_c$, we define the a-successor of I:

$$succ(I, a) = \{\mathbf{s}' \mid \exists \mathbf{s} \in I, \mathbf{u} \in L_u . \mathbf{s}' = \delta(\mathbf{s}, a, \mathbf{u})\} \quad (1)$$

One way to solve a reachability game is to reduce it to a sequence of *bounded reachability games*, that impose a bound on the number of rounds in which the controller must force the game into the goal. By increasing the bound we are guarantied to either find a strategy or prove that it does not exist. In this work we focus on solving bounded games.

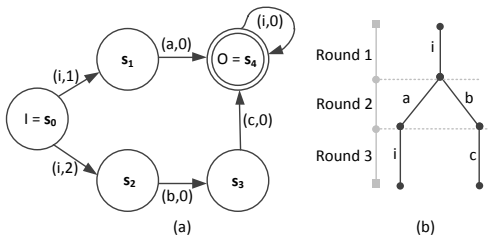


Figure 1: (a) Transition relation δ in Example 1; (b) Example abstract game trees.

Given a reachability game $\mathcal{G}$, function $\pi : S \to L_c$ is a *winning strategy* of length n on behalf of the controller from initial set $I \subseteq S$ if any sequence $(\mathbf{s}_0, \mathbf{c}_0, \mathbf{s}_1, \mathbf{c}_1, ..., \mathbf{s}_n)$, such that $\mathbf{s}_0 \in I$ and, $\mathbf{c}_k = \pi(\mathbf{s}_k)$, $\mathbf{s}_{k+1} \in succ(\{\mathbf{s}_k\}, \mathbf{c}_k)$, visits the goal set: $\exists j \in [0, n].\mathbf{s}_j \in O$. Dually, $\rho : S \times L_c \to L_u$ is a *spoiling environment strategy* of length n from I if there exists state $\mathbf{s}_0 \in I$ such that any sequence $(\mathbf{s}_0, \mathbf{c}_0, \mathbf{u}_0, \mathbf{s}_1, \mathbf{c}_1, \mathbf{u}_1, ..., \mathbf{s}_n)$ such that $\mathbf{u}_k = \rho(\mathbf{s}_k, \mathbf{c}_k)$, and $\mathbf{s}_{k+1} = \delta(\mathbf{s}_k, \mathbf{c}_k, \mathbf{u}_k)$, stays outside of the goal set: $\forall j \in [0, n].\mathbf{s}_j \notin O$. We say that the game is *winning* from I in n rounds if there exists a winning controller strategy for it. Otherwise, the game is *losing* and there exists a spoiling environment strategy.

Example 2 *Suppose that the bound n is 3, thus the controller must reach state s_4 in three rounds. The following strategy is winning for the controller from the initial set $I = \{s_0\}$ in 3 rounds: $\pi(s_0) = \pi(s_4) = i$, $\pi(s_1) = a$, $\pi(s_2) = b$ and $\pi(s_3) = c$.* □

In this paper we deal with *symbolic games* defined over three sets of Boolean variables s, c, and u. Each state in $\mathbf{s} \in S$ represents a valuation of variables s, each action in $\mathbf{c} \in L_c$ ($\mathbf{u} \in L_u$) represents a valuation of variables c (u). Sets relations of a symbolic game are represented by their characteristic formulas, e.g., the objective O is given as a Boolean formula $O(s)$ over variables s. The transition relation δ of the game is given as a Boolean formula $\Delta(s, c, u, s')$ over state, action, and next-state variables.

Games as QBF A bounded reachability game can be encoded as a QBF, where existentially and universally quantified variables model opponent moves. To this end, we introduce $n + 1$ copies of state variables s (where n is the bound on the length of the winning strategy): $s_0, \ldots, s_n$, where s_i encode the state of the game after i rounds. We also introduce n copies of controllable and uncontrollable action variables, c_i and u_i, $i = 0, \ldots, n-1$.

The following formula is true iff there exists a spoiling environment strategy for $\mathcal{G}$ of length n from I (equivalently, the formula is false iff the game is winning in n rounds):

$$E(I) = \exists s_0 \forall c_0 \exists u_0 ... \forall c_{n-1} \exists u_{n-1} \exists s_1 .. s_n . I(s_0) \wedge \psi_0 \quad (2)$$
$$\psi_i = \neg O(s_i) \wedge \Delta(s_i, c_i, u_i, s_{i+1}) \wedge \psi_{i+1}, i \in [0, n-1]$$
$$\psi_n = \neg O(s_n)$$

Intuitively, ψ_i encodes a play starting at round i. It is true iff this play is losing for the controller, i.e., it always stays outside of the goal region. Formula $E(I)$ holds if for some

initial state $s_0 \in I$, there exists an environment response u_i to any controller action c_i at every round of the game, such that the resulting run of the game is losing for the controller.

3 Solving games using ∀-expansion

One way to solve a QBF formula is to first convert it to CNF via ∀-*quantifier expansion*. Consider the formula $\exists x \forall y \exists z \phi(x, y, z)$. Expanding the ∀-quantifier while keeping the formula in the prenex form generates the following formula: $\exists x \exists z^1 z^2 [\phi(x, z^1)|_{y=0} \wedge \phi(x, z^2)|_{y=1}]$, where z^1 and z^2 are fresh copies of variables z.

Straightforward application of quantifier expansion introduces a large number of variables and is practically infeasible. A more pragmatic approach is to perform *partial expansion*, i.e., to consider only a subset of possible assignments to each universally quantified variable block. If the resulting CNF is unsatisfiable, then so is the original QBF.

From the game solving perspective, partial expansion of formula (2) corresponds to constructing an abstraction of the game that restricts moves available to the controller. If the controller wins the restricted game (i.e, the resulting CNF is unsatisfiable), then it also wins the original game. Specifically, we restrict the controller to choosing actions from a tree, called *abstract game tree*, which is a special case of ∀-expansion trees (Janota and Marques-Silva 2013).

Definition 1 (Abstract game tree) *An abstract game tree for a reachability game $\mathcal{G}$ and bound n is a rooted tree with uniform depth n where every edge is labelled with a total assignment to controllable action variables and where all outgoing edges of a node have distinct labels.*

Example 3 *Figure 1(b) shows an abstract game tree for the running example. The tree restricts the controller to playing two possible sequences of actions: (i, b, c) and (i, a, i).* □

Given an abstract game tree T, a node v of the tree, and a successor edge e of v, we denote $height(T)$ the distance from the root to a leaf of T, $root(T)$ the root node, $edges(v)$ the list of tuples (e, a, v') that represent outgoing edges of v, along with their associated actions and child nodes (v').

We define expansion E_T of the game formula (2) induced by T. To this end, we introduce a copy of state variables s_v for every node v and a copy of action variables u_e and c_e for every edge e of T.

$$E_T(I) = I(s_{root(T)}) \wedge \psi_{root(T)} \tag{3}$$

$$\psi_v = \neg O(s_v) \wedge \bigwedge_{(e,a,v') \in edges(v)} [\Delta(s_v, c_e, u_e, s_{v'}) \wedge c_e = a \wedge \psi_{v'}]$$

It is easy to see that E_T is a partial expansion of E, with existential quantifiers removed; hence if E is true, then there exists a satisfying assignment to E_T. It follows that if we can find T such that E_T is unsatisfiable, then E is false and hence the game is winning. Such T serves as a *certificate* of existence of a winning strategy.

Definition 2 (Certificate tree) *Given a set of states I and an abstract game tree T, such that $E_T(I)$ is unsatisfiable, we say that T is a certificate tree for the set I.*

Proposition 1 *The following statements are equivalent:*

- *$\mathcal{G}$ is winning from I in n rounds.*
- *There exists a certificate tree T for I of height n.*

In this paper we deal with game solvers that produce such certificate trees. In theory, any generic QBF solver that produces $\forall Exp + Res$ refutations or refutations that can be p-simulated by $\forall Exp + Res$ (Janota and Marques-Silva 2013; Beyersdorff, Chew, and Janota 2014) falls into this category. One example is the RAREQS QBF solver (Janota et al. 2012). In practice, generic QBF solvers do not scale well in solving real-world games (Ansótegui, Gomes, and Selman 2005; Sabharwal et al. 2006; Narodytska et al. 2014). This motivated the development of specialised game solvers that exploit domain-specific properties of reactive games, such as EVASOLVER (Narodytska et al. 2014).

In the rest of the paper we assume that the game is winning and the solver returns a certificate tree for it. All techniques presented here can be easily adapted to compute a spoiling strategy in case the game is losing.

4 Strategy generation

We use the certificate tree computed by the game solver as a starting point for strategy generation. We know that the controller can win the game in n rounds by picking actions from the tree; however we do not yet know exactly which actions should be picked in every state.

The strategy generation algorithm takes an initial set I and its certificate tree T, computed by the game solver, and generates a winning controller strategy from I. To this end, it first partitions I into subsets I_i, one for each outgoing branch of T (Figure 3), such that the controller can win from I_i by picking action a_i along the ith branch of the tree. This partitioning defines the local winning strategy in the root node of T. Next, for each partition I_i, the algorithm computes the set of a_i-successor states of I_i, obtaining the set I_i' of states winning in the child subtree T_i' of T_i (Figure 3). The algorithm is then invoked recursively for each subtree T_i'.

Example 4 *Figure 2 illustrates operation of the algorithm on the winning abstract game tree returned by the game solver for our running example (Figure 1(b)). The algorithm starts at the root of the tree and the initial set $I = \{s_0\}$. The game tree only defines one winning action is the root node, hence this action is winning is all states of I and no partitioning is required. We compute the successor set reachable by playing action i in I: $I' = succ(I, i) = \{s_1, s_2\}$ (see the game automaton in Figure 1(a)).*

Next, we descent down the i branch of the tree and consider subtree T' and its initial set I' (Figure 2(b)). We partition I' into subsets $I_1' = \{s_1\}$ and $I_2' = \{s_2\}$ that are winning for the left and right subtrees of T' respectively, i.e., the controller must play action a in state s_1 and b in s_2. Consider the resulting subtrees T_1' and T_2' with initial sets I_1' and I_2' (Figure 2(c)). We have $I_1'' = succ(I_1', a) = \{s_4\}$, $I_2'' = succ(I_2', b) = \{s_3\}$. Finally, we obtain two subtrees T_1'' and T_2'' with initial sets I_1'' and I_2'' (Figure 2(d)). Both subtrees have one branch; hence corresponding actions i and c are winning in I_1'' and I_2'' respectively.

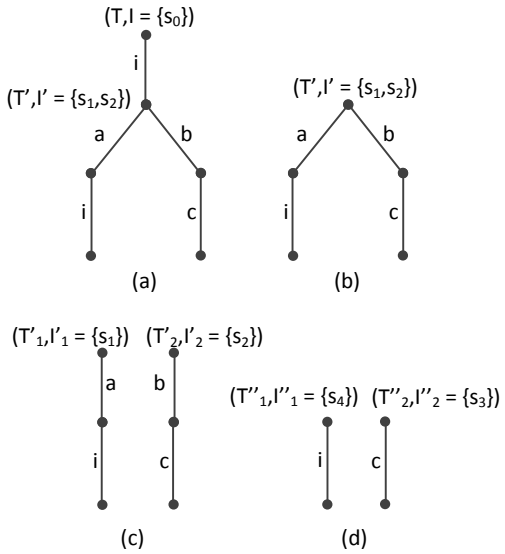


Figure 2: Operation of the strategy extraction algorithm on the running example.

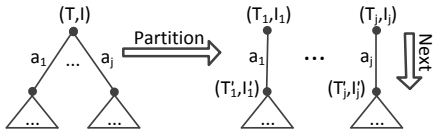


Figure 3: The PARTITION function.

Putting together fragments of the winning strategy computed above, we obtain the following strategy for this example: $\pi(s_0) = i$, $\pi(s_1) = a$, $\pi(s_2) = b$, $\pi(s_3) = c$, $\pi(s_4) = i$.

The above algorithm involves two potentially costly operations: winning set partitioning and successor set computation. If implemented naïvely, these operations can lead to unacceptable performance. The key insight behind our solution is that both operations can be efficiently approximated from the proof of unsatisfiability of the formula $E_T(I)$, with the help of interpolation, as described below. The resulting approximations are sound, i.e., preserve the correctness of the resulting strategy.

Algorithm 1 shows the pseudocode of the strategy generation algorithm. The algorithm proceeds in two phases: the first phase (GENLOCALSTRATS) computes local strategies in nodes of T; the second phase (COMPILESTRAT) compiles all local strategies into a winning strategy function.

The GENLOCALSTRATS function recursively traverses the certificate tree T, starting from the root, computing local strategies in each node. The main operation of the algorithm, called PARTITION, splits (T, I) into j pairs (T_i, I_i), as shown in Figure 3. Each tree T_i is a copy of a single branch of T. The partitioning is constructed in such a way that the action a_i that labels the root edge of T_i is a winning controller action in I_i.

Next we consider each pair (T_i, I_i) (lines 10-14). We descend down the tree and compute the controller strategy in the child subtree T_i' of T_i (right-hand side of Figure 3). To do so, we first compute the set of a_i-successors of I_i: More precisely, we compute an overapproximation $I_i' \supseteq succ(I_i, a_i)$, such that T_i' is a certificate tree for I_i'. Such an overapproximation is returned by the NEXT function in line 11. We can now recursively invoke the strategy generation function to compute a winning strategy for the pair (T_i', I_i') (line 12).

The algorithm returns the set of tuples (W, a, k). Each tuple represents a fragment of the strategy in some tree node, where W is the winning set in this node, a is the controller action to play in this set, and k is the distance from the node to the bottom of the tree (i.e., distance to the goal).

Algorithm 1 Computing a winning strategy

```
1:  function GenStrategy(T, I)
2:      Strat ← GenLocalStrats(T, I)
3:      return CompileStrat(Strat)
4:  end function

5:  function GenLocalStrats(T, I)
6:      v ← root(T)
7:      [(e_1, a_1, v_1), ..., (e_j, a_j, v_j)] ← edges(v)
8:      [(T_1, I_1), ..., (T_j, I_j)] ← Partition(T, I ∧ ¬O)
9:      Strat ← {(I_i, a_i, height(T)) | i ∈ [1, j]}
10:     for i = 1 to j do
11:         (T_i', I_i') ← Next(T_i, I_i)
12:         Strat_i ← GenLocalStrats(T_i', I_i')
13:         Strat ← Strat ∪ Strat_i
14:     end for
15:     return Strat
16: end function
```

Algorithm 2 Partitioning winning states

```
1:  function Partition(T, I)
2:      v ← root(T)
3:      Î ← I, T̂ ← T
4:      for i = 1 to j do
5:          (T_i, T̃) ← split(T̂)
6:          A ← E_T̃(Î)
7:          B ← E_T_i(⊤)
8:          I(s_v) ← Interpolant(A, B)
9:          I_i ← I(s) ∧ Î
10:         Î ← Î ∧ ¬I(s), T̂ ← T̃
11:     end for
12:     return [(T_1, I_1), ..., (T_j, I_j)]
13: end function
```

PARTITION

The PARTITION function (Algorithm 2) computes a local strategy in the root of an abstract game tree. It takes a pair (T, I), such that T is a certificate tree for set I and partitions I into subsets I_i such that the controller can win by choosing action a_i in I_i.

At every iteration, the algorithm splits the tree into the leftmost branch T_i and the remaining tree (Figure 4). It then computes the set I_i where the controller wins by following the branch T_i and removes I_i from the initial set I. At the

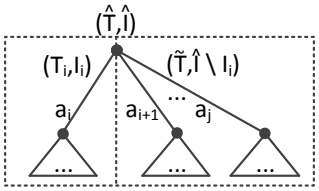


Figure 4: Splitting of T in the PARTITION function.

next iteration it considers the leftover tree $\tilde{T}_i$ and the shrunk initial set $\hat{I}$.

The algorithm maintains the invariant that $\hat{T}$ is a certificate tree for $\hat{I}$ and hence $E_{\hat{T}}(\hat{I})$ is unsatisfiable. We decompose this formula into two conjuncts $E_{\hat{T}}(\hat{I}) = A \wedge B$ such that A and B only share state variables s_v in the root node v of T and that the interpolant $\mathcal{I}$ of A and B consists of states where the controller can win by following the T_i subtree. Hence $\mathcal{I} \wedge \hat{I}$ gives us the desired set I_i.

Informally, A is a partial expansion of the game formula induced by $\tilde{T}$. It is satisfiable iff there exists a spoiling environment strategy from $\hat{I}$ against abstract game tree $\tilde{T}$. B is a partial expansion of the game induced by T_i. It is satisfiable iff there exists a spoiling environment strategy against T_i. Both A and B can be satisfiable individually, but their conjunction is unsatisfiable.

The interpolant $\mathcal{I}$ of A and B implies $\neg B$, i.e., for any state in $\mathcal{I}$, a_i is a winning move. $\mathcal{I}$ is also implied by A, i.e., it contains all states in $\hat{I}$ where the controller cannot win by picking moves from $\tilde{T}$ as a subset. Equivalently, for any state in $I_i \wedge \neg\mathcal{I}$, the controller *can* win by following $\tilde{T}$, i.e., $\tilde{T}$ is a certificate tree for $I_i \wedge \neg\mathcal{I}$, and we can apply the decomposition again to $\tilde{T}$ and $I_i \wedge \neg\mathcal{I}$ at the next iteration.

We prove useful properties of the PARTITION function. First, we show that A and B indeed form a conjunctive decomposition of $E_{\hat{T}}(\hat{I})$.

Proposition 2 $A \wedge B = E_{\hat{T}}(\hat{I})$.

Proof: $A \wedge B = E_{\tilde{T}}(\hat{I}) \wedge E_{T_i}(\top) = \hat{I} \wedge \psi_{root(\tilde{T})} \wedge \top \wedge \psi_{root(T_i)} = \hat{I} \wedge (\psi_{root(\tilde{T})} \wedge \psi_{root(T_i)}) = \hat{I} \wedge \psi_{\hat{T}} = E_{\hat{T}}(\hat{I})$. □

Proposition 3 *The following invariant is maintained throughout the execution of* PARTITION*: $\hat{T}$ is a certificate tree for $\hat{I}$.*

Proof: We prove by induction. The invariant holds for the initial assignments of $\hat{T}$ and $\hat{I}$. By Proposition 2 and induction hypothesis, $A \wedge B = E_{\hat{T}}(\hat{I})$ is unsatisfiable. Hence the interpolation operation in line 8 is well defined. By the properties of interpolants, $(A \implies \mathcal{I}(s_v))$, hence $(\neg\mathcal{I}(s_v) \implies \neg A)$ or equivalently $(\neg\mathcal{I}(s_v) \implies \neg E_{\tilde{T}}(\hat{I}))$.

After $\hat{T}$ and $\hat{I}$ are updated in line 10, their new values $\hat{T}'$ and $\hat{I}'$ satisfy the following equalities: $E_{\hat{T}'}(\hat{I}') = E_{\tilde{T}}(\hat{I} \wedge \neg\mathcal{I}(s)) = \neg\mathcal{I}(s_v) \wedge E_{\tilde{T}}(\hat{I}) \implies \neg E_{\tilde{T}}(\hat{I}) \wedge E_{\tilde{T}}(\hat{I}) = \bot$ and hence the invariant is maintained.□

Proposition 4 *Let T be a certificate tree for I and let $I \wedge O = \bot$. Then $[(T_1, I_1), \ldots, (T_j, I_j)] =$ PARTITION(T, I) is a local winning strategy in the root of T, i.e., the following properties hold:*

1. *Sets $I_1, \ldots, I_j$ comprise a partitioning of I: $I = \bigvee I_i$ and $\forall i, k.(i \neq k) \implies I_i \wedge I_k = \bot$*
2. *T_i is a certificate tree for I_i, for all $i \in [1, j]$*

Algorithm 3 Successor set

```
1: function NEXT(T, I)
2:     T' ← subtree(T, 1)
3:     v ← root(T)
4:     [(e, a, v')] ← edges(v)
5:     A ← I(s_v) ∧ Δ(s_v, c_e, u_e, s_v') ∧ c_e = a
6:     B ← E_T'(⊤)
7:     I(s_v') ← Interpolant(A, B)
8:     return (T', I(s))
9: end function
```

Proof: At every iteration of the algorithm, we partition $\hat{I}$ into $I_i = \mathcal{I} \wedge \hat{I}$ and $\hat{I} \wedge \neg\mathcal{I}$; hence different sets I_i do not overlap by construction.

At the final iteration of the algorithm, the tree $\tilde{T}$ consists of a single root node without outgoing branches. Hence, $A = E_{\tilde{T}}(\hat{I}) = \hat{I}(s_v) \wedge \neg O(s_v) = \hat{I}(s_v)$. Since $(A \implies \mathcal{I}(s_v))$, we get $(\hat{I} \implies \mathcal{I}(s))$ and therefore $\mathcal{I}(s) \wedge \hat{I} = \hat{I}$, i.e., all states in $\hat{I}$ are included in the final set I_j and hence the partitioning completely covers set I: $I = \bigvee I_i$.

We prove the second statement of the proposition. The set I_i is computed as $\mathcal{I}(s) \wedge \hat{I}$ (line 9) at the ith iteration of the algorithm. Thus, $E_{T_i}(I_i) = E_{T_i}(\mathcal{I}(s) \wedge \hat{I}) = \mathcal{I}(s) \wedge \hat{I} \wedge E_{T_i}(\top)$. By the properties of interpolants, $\mathcal{I}(s) \wedge B = \mathcal{I}(s) \wedge E_{T_i}(\top) = \bot$. Hence $E_{T_i}(I_i) = \bot$, i.e., T_i is a certificate tree for I_i. □

NEXT

The NEXT function (Algorithm 3) takes a set I and its certificate tree T, such that there is exactly one outgoing edge, labelled a, from the root node of T. T has a sole child subtree T' with root node v'. The function computes an overapproximation I' of the a-successor of I, such that I' is winning for the controller and T' is a certificate tree for I'.

Once again, we decompose the unsatisfiable formula $E_T(I)$ into two conjuncts A and B. A encodes one round of the game from the set I, where the controller plays action a. $B = E_{T'}(\top)$ is a partial $\forall$-expansion of the game induced by T'. A and B only share state variables $s_{v'}$ and their interpolant gives the approximation we are looking for.

Proposition 5 *Let T be a certificate tree for I with a single outgoing edge, labelled a in its root node, and let $(T', \mathcal{I}) =$ NEXT(T, I). Then:*

1. *$\mathcal{I}$ is an overapproximation of the a-successor of I, i.e., $\mathcal{I} \supseteq succ(I, a)$*
2. *T' is a certificate tree for I'*

Proof: We rewrite formula (1) in the symbolic form: $succ(I, a) = \exists s_v, u_e.I(s_v) \wedge \Delta(s_v, c_e, u_e, s_{v'}) \wedge c_e = a$.

Algorithm 4 Compiling the winning strategy

```
1: function CompileStrat(Strat)
2:     π ← ⊥,  W ← ⊥
3:     for (I, a, k) ∈ Strat do    ▷ Sorted by ascending k
4:         π ← π ∨ (I ∧ ¬W ∧ (c = a))
5:         W ← W ∨ I
6:     end for
7:     return π
8: end function
```

The matrix of this formula is exactly formula A. Hence $succ(I,a) = \exists s_v, u_e.A$. Since $(A \implies \mathcal{I}(s_{v'}))$, $succ(I,a) \implies \exists s, u.\mathcal{I}(s_{v'})$. Since $\mathcal{I}$ is defined over state variables only, the quantifiers can be removed: $succ(I,a) \implies \mathcal{I}(s_{v'})$ or, in the relational form, $\mathcal{I} \supseteq succ(I,a)$. We prove the second property: $E_{T'}(\mathcal{I}) = \mathcal{I}(s_{v'}) \wedge E_{T'}(\top) = \mathcal{I}(s_{v'}) \wedge B = \bot$. □

Compiling the strategy

Finally, we describe how local strategies computed by GenLocalStrats are combined into a winning strategy for the game. This requires some care, as individual partial strategies can be defined over overlapping sets of states. We want the resulting strategy function to be deterministic; therefore for each partial strategy we only add new states not yet covered by the computed combined strategy. Function CompileStrats (Algorithm 4) achieves this by keeping track of all states W already added to the strategy. For every new tuple (I,a,k), it restricts the set I to $\neg W$, which guarantees that no state can be added to the strategy twice. Furthermore, by considering tuples with smaller values of k first, we resolve the nondeterminism in a way that guarantees progress towards the goal at every round of the game.

Let $rank(s), s \in S$, be the smallest k such that there exists $(I,a,k) \in Strat$, $s \in I$, or ∞ if there is no such k.

Proposition 6 *Let π = CompileStrat($Strat$). For any pair $(s,a) \in \pi$, there exists $(I,a,k) \in Strat$ such that $s \in I$ and $k = rank(s)$.*

Theorem 1 (Correctness of the algorithm) *Let abstract game tree T of height n be a certificate tree for the set I, let π be a partial function returned by the strategy generation algorithm, π = GenStrategy(T, I), and let π' be an arbitrary extension of π to a complete function. Then π' is a winning controller strategy of length n from I.*

Proof: Every state-action pair (s,a) in π is generated by the Partition function and, according to Proposition 4, a is a winning controller move in s. By Proposition 5, all possible a-successors of s are either goal states or are covered by π. Hence, by following the strategy π from I, the controller is guaranteed to stay within the winning region of the game until reaching the goal.

Next, we show that ranks of states visited by following the strategy π decrease monotonically and hence the strategy reaches the goal in at most n steps. According to Proposition 6, for every pair $(s,a) \in \pi$, there exists $(I,a,k) \in Strat$, such that $k = rank(s)$. Therefore, for any $s' \in succ(s,a)$, such that $s' \notin O$, there exists $(I',a',k-1) \in Strat$, $s' \in I'$; hence $rank(s') \leq k-1 < k = rank(s)$. □

5 Evaluation

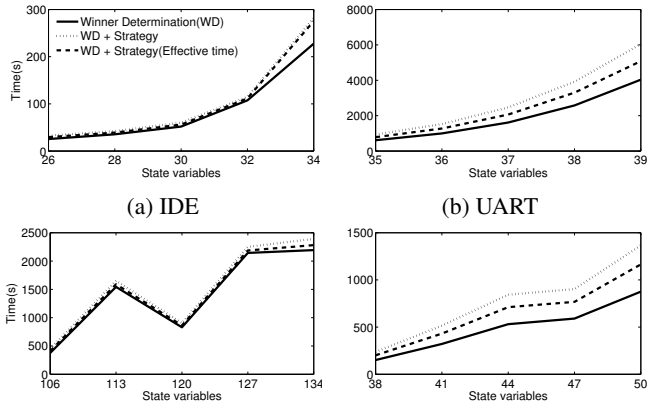
We implemented our strategy generation algorithm as an extension on top of EvaSolver. We use PeRIPLO (Rollini et al. 2014) to generate interpolants in Algorithms 2–3. We use BDDs to compile and store winning sets W in Algorithm 2 (Somenzi 2014). EvaSolver implements an important optimisation whereby it generates multiple certificate trees for fragments of the original game, which enables computational learning of winning states. Our strategy generation algorithm is invoked in an online fashion, whenever EvaSolver computes a certificate tree for a fragment. This results in multiple partial strategies, where each strategy is computed as described in the previous section. We introduce an additional final step to EvaSolver, which combines partial strategies into a global winning strategy for the original game.

We evaluate our implementation on driver synthesis benchmarks from (Narodytska et al. 2014). These benchmarks model the data path of four I/O devices in the abstracted form. Each specification was parameterised to scale in the size of the state space and the winning strategy length. The four families are UART, SPI, IDE and Ethernet, which model respectively the send queue of a UART serial device, the command queue of an SPI device, the DMA descriptor list of an IDE controller, and the transmit buffer of an Ethernet device respectively. On two of these families, namely SPI and Ethernet, EvaSolver outperforms a state-of-the-art BDD-based game solver.

Figure 5 summarises our results. For each family, we show strategy generation time as a function of the number of state variables in the specification for 5 hardest instances of the family solved by EvaSolver. Specifically, we show the time it took the base solver to determine the winner (the WD line), as well as the total time taken to solve the game and compute the winning strategy using our algorithm (WD+Strategy).

Profiling showed that non-negligible overhead was introduced by transferring CNFs from EvaSolver's internal representation to the representation used by the interpolation library. This overhead can be almost completely eliminated with additional engineering effort. Hence, we also show the effective time spent solving the game and computing the strategy, excluding the formula translation time (the Effective Time line).

Table 1 shows a detailed breakdown of experimental results, including the number of state variables for each instance (**Vars**) and the total time taken by the solver (**Total(s)**), split between the time used to determine the winner (and generate certificate trees) (**Base(s)**) and the strategy generation time (**Strategy(s)**). The **OH** and **EffOH** columns show total and effective overheads introduced by the strategy generation step. The **Size** column shows the size of the strategy, i.e., the number of (W,a,k) tuples returned by the GenLocalStrats function. The last three columns report on our use of the PeRIPLO interpolation library in terms of



(a) IDE (b) UART

(c) SPI Flash (d) Ethernet

Figure 5: Performance of strategy extraction algorithm on four parameterised benchmarks. The X-axis shows the number of state vars in the game.

Vars	Total(s)	Base(s)	Strategy(s)	OH	EffOH	Size	INum	IAvg	IMax
IDE benchmark									
26	32.62	25.42	7.20	1.28	1.16	50	48	1193 (23)	24155 (118)
28	42.20	35.49	6.72	1.19	1.10	59	52	2489 (27)	39384 (119)
30	60.04	51.93	8.11	1.16	1.08	92	43	72 (17)	1583 (148)
32	115.11	107.35	7.77	1.07	1.04	60	36	18 (14)	85 (27)
34	283.08	227.67	55.40	1.24	1.21	159	49	2150 (15)	103941 (38)
SPI benchmark									
15	0.35	0.26	0.09	1.36	1.26	8	5	11 (9)	21 (9)
22	0.94	0.72	0.22	1.31	1.19	15	12	10 (10)	22 (18)
29	2.46	1.90	0.56	1.29	1.16	22	17	12 (10)	25 (18)
36	3.56	2.91	0.65	1.22	1.11	107	22	11 (10)	22 (18)
43	9.11	7.09	2.03	1.29	1.13	166	27	11 (10)	21 (14)
50	16.20	12.85	3.35	1.26	1.12	233	32	11 (11)	23 (18)
57	25.00	19.86	5.14	1.26	1.12	322	37	12 (11)	21 (18)
64	38.48	31.48	7.00	1.22	1.10	416	42	12 (11)	24 (18)
71	57.88	47.94	9.94	1.21	1.09	70	47	12 (12)	21 (18)
78	91.51	75.02	16.49	1.22	1.10	636	52	12 (12)	22 (19)
85	141.10	116.71	24.39	1.21	1.09	773	57	13 (12)	23 (20)
92	193.96	162.05	31.91	1.20	1.09	917	62	13 (13)	24 (21)
99	309.44	256.88	52.55	1.20	1.09	1059	67	13 (13)	22 (22)
106	449.49	377.48	72.00	1.19	1.09	1223	72	13 (13)	23 (23)
113	1645.44	1543.84	101.60	1.07	1.03	117	77	14 (13)	24 (24)
120	901.95	830.17	71.77	1.09	1.04	1637	82	14 (14)	25 (25)
127	2259.65	2143.40	116.25	1.05	1.02	139	87	14 (14)	26 (26)
134	2385.74	2193.65	192.09	1.09	1.04	152	92	14 (14)	27 (27)
Ethernet benchmarks									
14	0.06	0.03	0.02	1.60	1.52	2	1	24 (13)	24 (13)
17	0.49	0.29	0.20	1.70	1.45	21	7	33 (16)	87 (30)
20	1.97	1.14	0.82	1.72	1.45	176	15	41 (16)	110 (26)
23	5.39	3.23	2.16	1.67	1.39	185	25	64 (23)	180 (42)
26	14.61	7.94	6.66	1.84	1.48	266	36	100 (24)	347 (45)
29	27.41	15.71	11.70	1.74	1.43	677	44	155 (24)	779 (48)
32	58.02	35.38	22.64	1.64	1.36	208	61	136 (28)	676 (55)
35	111.69	69.26	42.43	1.61	1.35	351	80	141 (31)	933 (75)
38	238.09	151.21	86.89	1.57	1.33	545	116	184 (32)	1081 (63)
41	513.61	321.78	191.82	1.60	1.34	1525	154	184 (35)	1123 (72)
44	845.51	530.68	314.83	1.59	1.34	2159	191	276 (37)	2253 (64)
47	903.79	590.19	313.60	1.53	1.30	1547	228	261 (38)	1780 (71)
50	1368.23	875.90	492.33	1.56	1.33	1670	236	372 (38)	2292 (85)
UART benchmarks									
15	2.86	2.19	0.67	1.31	1.19	12	40	28 (6)	90 (14)
20	3.16	2.33	0.83	1.36	1.20	20	14	35 (12)	155 (23)
21	10.06	6.96	3.09	1.44	1.24	35	34	48 (9)	306 (26)
26	27.89	18.55	9.34	1.50	1.27	65	60	92 (13)	730 (41)
27	63.68	41.49	22.20	1.53	1.29	93	94	96 (13)	825 (44)
28	137.24	90.68	46.56	1.51	1.27	103	136	138 (13)	1356 (42)
29	270.66	178.75	91.92	1.51	1.27	134	184	212 (15)	2806 (47)
34	553.29	360.76	192.53	1.53	1.28	191	246	299 (16)	6360 (54)
35	938.68	612.63	326.05	1.53	1.28	285	307	258 (16)	7949 (69)
36	1525.99	995.25	530.74	1.53	1.28	410	382	348 (17)	6408 (62)
37	2464.13	1611.45	852.68	1.53	1.28	950	456	414 (18)	10592 (75)
38	3927.64	2577.39	1350.25	1.52	1.28	1223	546	504 (18)	34431 (74)
39	6030.77	4031.98	1998.79	1.50	1.26	674	633	608 (18)	29996 (72)

Table 1: Detailed experimental results.

the number of interpolation operations performed by the algorithm when solving the instance, the average and the maximal size of interpolants returned by PeRIPLO. Numbers in brackets show the size of the interpolant compiled to a BDD.

Our results show that strategy generation adds a modest overhead to the base solver. Effective overheads are about 12% for IDE and SPI, about 35% for Ethernet and about 30% for UART. Most of this overhead is due to interpolant computation. Moreover, experiments show that our algorithm scales linearly with the time taken by the base solver to determine the winner.

These results show that our algorithm is efficient, scalable and robust. The last property is particularly interesting, as existing strategy extraction algorithms for traditional game solvers, based on winning set compilation, have been reported to introduce significant variance across instances (Bloem et al. 2014b; Bloem, Könighofer, and Seidl 2013). A conclusive comparison can only be performed by evaluating both types of algorithms on a common set of benchmarks. Such a comparison requires first extending EVASOLVER to support unbounded safety and reachability games and is part of the future work.

6 Related work

All existing strategy extraction algorithms for games were developed for use with game solvers based on winning set compilation (Bloem et al. 2007; 2014a). Such a solver generates a sequence of expanding state sets from which the game can be reached in $1, 2, \ldots$ steps. The task of the strategy extraction algorithm is to compute a function that in every winning state chooses a single move that forces the game closer to the goal. In contrast, our strategy generation algorithm does not require the game solver to compile winning regions, but instead uses abstract game trees.

Another line of related work is strategy extraction algorithms for QBFs used in QBF certification. QBF strategy extraction methods are specific to the underlying proof system used by the QBF search algorithm (Lonsing and Biere 2010; Egly, Lonsing, and Widl 2013; Goultiaeva, Gelder, and Bacchus 2011). A strategy in a QBF is an oracle that, given the history of moves played in the game, outputs the next move for the winning player. An additional procedure is required to convert this oracle into a memory-free strategy function that maps a state to a controller move. Our work can been seen as such a procedure for $\forall Exp + Res$ proof system based solvers (Janota and Marques-Silva 2013).

7 Conclusion and future work

Our results demonstrate that abstract game tree-based game solvers admit an efficient strategy extraction procedure. As such, they provide a new compelling argument for further exploration of this promising approach to two-player games. In particular, our ongoing work is focusing on extending this approach to support a wider class of games, including unbounded reachability and safety games. This will enable us to directly compare the performance of our strategy extraction algorithm against existing BDD and SAT-based techniques (Bloem et al. 2014a).

References

Ansótegui, C.; Gomes, C. P.; and Selman, B. 2005. The Achilles' heel of QBF. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 275–281.

Beyersdorff, O.; Chew, L.; and Janota, M. 2014. On unification of QBF resolution-based calculi. In *Mathematical Foundations of Computer Science 2014*, volume 8635 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 81–93.

Bloem, R.; Galler, S.; Jobstmann, B.; Piterman, N.; Pnueli, A.; and Weiglhofer, M. 2007. Specify, compile, run: Hardware from PSL. *ENTCS* 190(4):3–16.

Bloem, R.; Egly, U.; Klampfl, P.; Könighofer, R.; and Lonsing, F. 2014a. SAT-Based Methods for Circuit Synthesis. In *FMCAD*.

Bloem, R.; Egly, U.; Klampfl, P.; Könighofer, R.; and Lonsing, F. 2014b. SAT-Based Methods for Circuit Synthesis. *CoRR* http://arxiv.org/abs/1408.2333.

Bloem, R.; Könighofer, R.; and Seidl, M. 2013. SAT-based synthesis methods for safety specs. *CoRR* http://arxiv.org/abs/1311.3530.

Cassez, F.; Jessen, J. J.; Larsen, K. G.; Raskin, J.-F.; and Reynier, P.-A. 2009. Automatic synthesis of robust and optimal controllers - an industrial case study. In *HSCC*, 90–104.

Egly, U.; Lonsing, F.; and Widl, M. 2013. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In McMillan, K. L.; Middeldorp, A.; and Voronkov, A., eds., *LPAR*, volume 8312 of *Lecture Notes in Computer Science*, 291–308. Springer.

Ehlers, R.; Könighofer, R.; and Hofferek, G. 2012. Symbolically synthesizing small circuits. In Cabodi, G., and Singh, S., eds., *FMCAD*, 91–100. IEEE.

Goultiaeva, A.; Gelder, A. V.; and Bacchus, F. 2011. A uniform approach for generating proofs and strategies for both True and False QBF formulas. In *IJCAI*, 546–553.

Janota, M., and Marques-Silva, J. 2013. On propositional QBF expansions and q-resolution. In *SAT*, 67–82.

Janota, M.; Klieber, W.; Marques-Silva, J.; and Clarke, E. M. 2012. Solving QBF with counterexample guided refinement. In *SAT*, 114–128.

Jiang, J.-H. R.; Lin, H.-P.; and Hung, W.-L. 2009. Interpolating functions from large boolean relations. In *ICCAD*, 779–784. IEEE.

Lonsing, F., and Biere, A. 2010. Integrating dependency schemes in search-based QBF solvers. In *SAT*, 158–171.

Narodytska, N.; Legg, A.; Bacchus, F.; Ryzhyk, L.; and Walker, A. 2014. Solving games without controllable predecessor. In *CAV*, 533–540.

Pudlak, P. 1997. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.* 62(3):981–998.

Rollini, S.; Bruttomesso, R.; Sharygina, N.; and Tsitovich, A. 2014. Resolution proof transformation for compression and interpolation. *Formal Methods in System Design* 45(1):1–41.

Ryzhyk, L.; Walker, A. C.; Keys, J.; Legg, A.; Raghunath, A.; Stumm, M.; and Vij, M. 2014. User-guided device driver synthesis. In *OSDI*.

Sabharwal, A.; Ansotegui, C.; Gomes, C. P.; Hart, J. W.; and Selman, B. 2006. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *SAT*, 382–395.

Somenzi, F. 2014. CUDD: CU decision diagram package. http://vlsi.colorado.edu/ fabio/CUDD/.