Solving Games without Controllable Predecessor

Nina Narodytska[1 2], Alexander Legg[1], Fahiem Bacchus[2],
Leonid Ryzhyk[1 2], and Adam Walker[1]

[1] NICTA** and UNSW, Sydney, Australia
[2] University of Toronto, Canada

Abstract. Two-player games are a useful formalism for the synthesis of reactive systems. The traditional approach to solving such games iteratively computes the set of winning states for one of the players. This requires keeping track of all discovered winning states and can lead to space explosion even when using efficient symbolic representations. We propose a new method for solving reachability games. Our method works by exploring a subset of the possible concrete runs of the game and proving that these runs can be generalised into a winning strategy on behalf of one of the players. We use counterexample-guided backtracking search to identify a subset of runs that are sufficient to consider to solve the game. We evaluate our algorithm on several families of benchmarks derived from real-world device driver synthesis problems.

1 Introduction

Two-player games are a useful formalism for the synthesis of reactive systems, with applications in software [15] and hardware design [4], industrial automation [7], etc. We consider finite-state *reachability games*, where player 1 (the controller) must force the game into a *goal region* given any valid behaviour of player 2 (the environment).

The most successful method for solving two-player games is based on the *controllable predecessor* ($Cpre$) operator [14], which, given a target set of states, computes the set from which the controller can force the game into the target set in one round. $Cpre$ is applied iteratively, until a fixed point is reached. The downside of this method is that it keeps track of all discovered winning states, which can lead to a space explosion even when using efficient symbolic representation such as BDDs or DNFs.

We propose a new method for solving reachability games. Our method works by exploring a subset of the concrete runs of the game and proving that these runs can be generalised into a winning strategy on behalf of one of the players. In contrast to the $Cpre$-based approach, as well as other existing synthesis methods, it does not represent, in either symbolic or explicit form, the set of states visited by the winning strategy. Instead, it uses counterexample-guided backtracking search to identify a small subset of runs that are sufficient to solve the game.

We evaluate our algorithm on several benchmarks derived from driver synthesis problems. We find that it outperforms a highly optimised BDD-based solver on the subset of benchmarks that do not admit a compact representaion of the winning set, thus demonstrating the potential of the new approach.

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

2 Related work

Our algorithm is inspired by the RAReQS QBF solver [10]. RAReQS treats a QBF formula in the prenex normal form as a game between the universal and the existential player. It uses counterexample-guided backtracking search to efficiently expand quantifier blocks. We build on the ideas of RAReQS, to construct a domain-specific solver for reachability games that takes advantage of the structure of such games.

One alternative to the $Cpre$-based method encodes the game as a quantified boolean formula (QBF), where controller and environment moves are encoded as alternating existential and universal quantifiers [2]. More recently several SAT-based synthesis methods have been proposed [13, 5]. Similarly to $Cpre$-based techniques, they incrementally compute the set of winning (or losing) states, in the DNF form, and refine it using a SAT solver. Sabharwal et al. [16] explore the duality of games and QBF formulas and propose a hybrid CNF/DNF-based encoding of games that helps speed up QBF solving. The bounded synthesis method [11] aims to synthesise a controller implementation with a bounded number of states. In the present work, we impose a bound on the number of rounds in the game, which is necessary to encode it into SAT.

Our method uses counterexample-guided abstraction refinement to identify potentially winning moves of the game. Several abstraction refinement algorithms for games have been proposed in the literature [9, 1]. Our algorithm is complementary to these techniques and can be combined with them.

The idea of solving games by generalising a winning run into a complete strategy has been explored in explicit-state synthesis [6]. In contrast to these methods, we use a SAT solver to compute and generalise winning runs symbolically. This enables us to solve games with very large state spaces, which is not possible using explicit search, even when performing it on the fly.

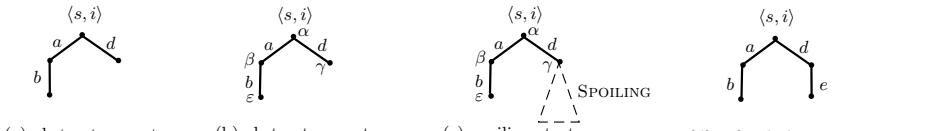
3 Background

Games and strategies A reachability game $G = \langle S, L_c, L_u, I, O, \delta\rangle$ consists of a set of states S, controllable actions L_c, uncontrollable actions L_u, initial state $I \in S$, 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 an initial state. In each round, the controller picks an action $c \in L_c$. The environment responds by picking an action $u \in L_u$, and the game transitions to a new state $\delta(s, c, u)$.

A *controller strategy* $\pi : S \to L_c$ associates with every state a controllable action to play in this state. Given a bound n on the number of rounds, π is a *winning* strategy in state s at round $i \leq n$ if any sequence $(s_i, u_i, s_{i+1}, u_{i+1}, ..., s_n)$, such that $s_i = s$ and $s_{k+1} = \delta(s_k, \pi(s_k), u)$, visits the goal set: $\exists j \in [i, n].s_j \in O$. A state-round pair $\langle s, i\rangle$ is winning if there exists a winning strategy in s at round i. A state-round-action tuple $\langle s, i, c\rangle$ is winning if there does *not* exist a spoiling strategy for s and c at round i.

In this paper we are concerned with the problem of *solving the game*, i.e., checking whether the initial state I is winning at round 0 for the given bound n. Note that bounding the number of rounds to reach the goal is a conservative restriction: any winning strategy in the bounded game is winning in the unbounded game. If, on the other hand, a winning strategy for a bound n cannot be found, n can be relaxed.

Symbolic games In this paper we deal with *symbolic games* defined over three sets of boolean variables X, Y_c, and Y_u. Each state $s \in S$ represents a valuation of variables X,



(a) abstract game tree (b) abstract game tree with partial strategy (c) spoiling strategy in leaf (d) (d) refined abstract game tree

Fig. 1: Abstract game tree.

each action $c \in L_c$ ($u \in L_u$) represents a valuation of variables Y_c (Y_u). The transition relation δ of the game is given as a boolean formula $\Delta(X, Y_c, Y_u, X')$ over state, action, and next-state variables.

4 Abstract game trees

Our algorithm constructs a series of abstractions of the input game. An abstraction restricts actions available to one of the players. Specifically, we consider abstractions represented as trees of actions, referred to as *abstract game trees*. Together with a state-round pair $\langle s, i\rangle$, an abstract game tree defines an *abstract game* played from this state. Figure 1a shows an example abstract game. In the abstract game, the environment player is required to pick actions from the tree, starting from the root node. After reaching a leaf, it continues playing unrestricted. The tree in Figure 1a restricts the initial environment action to the set $\{a, d\}$. After choosing action d, the environment reaches a leaf of the tree and continues playing unrestricted. Alternatively, after choosing a, the environment is required to play action b in the next round.

Nodes of an abstract game tree are uniquely identified by the list of edge labels along the path from the root to the node. We identify an abstract game tree with the set of its nodes. For example, the tree in Figure 1a can be written as $\{(), (d), (a), (a, b)\}$. We denote $leaves(T)$ the subset of leaf nodes of a tree T.

A *partial strategy* $Strat : T \to L_c$ assigns a controllable action to be played in each node of the abstract game tree. Figure 1b shows an example partial strategy. The controller starts by choosing action α. If the environment plays a, the controller responds with β in the next round, and so on. Given a partial strategy $Strat$, we can map each leaf l of the abstract game tree to $\langle s', i'\rangle = outcome(\langle s, i\rangle, Strat, l)$ obtained by playing all controllable and uncontrollable actions on the path from the root to the leaf.

5 The algorithm

Figure 2 and Algorithm 1 illustrate our algorithm, called EvaSolver. The algorithm takes a concrete game G as an implicit argument. In addition, it takes a state-round pair $\langle s, i\rangle$ and an abstract game tree AbsGT and returns a winning partial strategy for it, if one exists. The initial invocation of the algorithm takes the initial state $\langle I, 0\rangle$ and an empty abstract game tree $\emptyset$. The empty game tree does not constrain opponent moves, hence solving such an abstraction is equivalent to solving the original concrete game. The algorithm is organised as a counterexample-guided abstraction refinement (CEGAR) loop. The first step of the algorithm uses the FindCand function, described in detail below, to come up with a candidate partial strategy for AbsGT. If it fails to find a strategy, this means that no winning partial strategy exists for AbsGT. If, on the other hand, a candidate partial strategy is found, we need to verify if it is indeed winning for AbsGT.

Algorithm 1 CEGAR-based algorithm for solving games

```
function EvaSolver(⟨s, i⟩, AbsGT)
output a winning partial strategy if there is one; ∅ otherwise
    Cand ← FindCand(⟨s, i⟩, AbsGT)
    // FindCand returns a precise solution for i = n − 1
    if i = n − 1 return Cand
    AbsGT′ ← AbsGT
    loop
        if Cand = ∅ return ∅
        Counterex ← Verify(⟨s, i⟩, AbsGT, Cand)
        if Counterex = NULL return Cand
        else
            AbsGT′ ← Refine(AbsGT′, Counterex)
            Cand ← EvaSolver(⟨s, i⟩, AbsGT′)
    end loop
end function

function Refine(AbsGT, ⟨l, Spoiling⟩)
    let l = (e_i, ..., e_r)
    return AbsGT ∪ {(e_i, ..., e_r), Spoiling(())}
end function

function FindCand(⟨s, i⟩, AbsGT)
    φ ← ⋀_{j=i...n−1} δ(s_j, c_j, u_j, s_{j+1}) ∧
        ⋁_{j=i...n} O(s_j)
    for l ∈ leaves(AbsGT) do
        // e_j are environment actions along the path from
        // the root to l in AbsGT
        let l = (e_i, ..., e_r)
        p ← ⋀_{m=i...r} u_m = e_m
        φ_l ← rename(φ, l) ∧ (s_i = s) ∧ p
    sol ← SAT(⋀_{l∈leaves(AbsGT)} φ_l)
    if sol = unsat return ∅
    return {⟨v, c⟩ | v ∈ nodes(AbsGT), c = sol |_{c_v}}
end function

function Verify(⟨s, i⟩, AbsGT, Cand)
    for l ∈ leaves(⟨s, i⟩, AbsGT) do
        ⟨s′, i′⟩ = outcome(⟨s, i⟩, AbsGT, l)
        Spoiling ← EvaSolver(⟨s′, i′⟩, ∅)
        if Spoiling ≠ ∅ return ⟨l, Spoiling⟩
    return NULL // no spoiling strategy found
end function
```

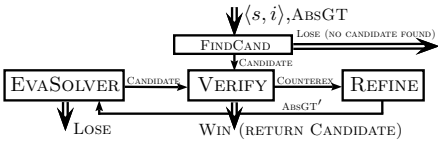


Fig. 2: CEGAR loop of Algorithm 1

The Verify procedure searches for a *spoiling* counterexample strategy in each leaf of the candidate partial strategy by calling the *dual solver* $\overline{\text{EvaSolver}}$. The dual solver solves a *safety* game on behalf of the environment player, where the environment must stay away from the goal for a bounded number of steps. Figure 1c shows a spoiling strategy discovered in one of the leaves of the abstract game tree. The dual algorithm is analogous to the primary solver. We do not present its pseudocode due to limited space. If the dual solver can find no spoiling strategy at any of the leaves, then the candidate partial strategy is a winning one. Otherwise, the Refine function extracts the first move of the spoiling strategy (i.e., the move that the strategy plays in the root node $()$ of the abstract game tree constructed by the dual solver) and uses it to refine the abstract game by adding a new edge labelled with this move to the leaf (Figure 1d).

We solve the refined game by recursively invoking EvaSolver on it. If no partial winning strategy is found for the refined game then there is also no partial winning strategy for the original abstract game, and the algorithm returns a failure. Otherwise, the partial strategy for the refined game is *projected* on the original abstract game by removing the leaves introduced by refinements. The resulting partial strategy becomes a candidate strategy to be verified at the next iteration of the loop.

The loop terminates, in the worst case, after refining the game with all possible environment actions. However, to achieve good performance, the algorithm must be able to solve the game using a small number of refinements. The FindCand procedure plays the key role in achieving this. We use the following criterion to find potentially winning candidates efficiently: we search for a partial strategy such that after playing the strategy from the root to any of the leaves of the abstract game tree, we can choose a sequence of follow-up moves for both players taking the game into the goal region.

Effectively, we try to win the game under the assumption that the players cooperate to reach the goal rather than competing with each other. If such an optimistic partial strategy does not exist, then we cannot win the abstract game. On the other hand, if we do find such a strategy, it is likely to either be a winning one or to produce a useful counterexample that will speed up the search for a winning strategy. This is based on the observation that in industrial synthesis problems the environment typically represents a hardware or software system designed to allow efficient control. Environment actions model responses to control signals, which require appropriate reaction from the controller, but are not aimed to deliberately counteract the controller. Unlike in truly competitive games like chess, a straightforward path to the goal is likely to be a good first approximation of a correct winning strategy.

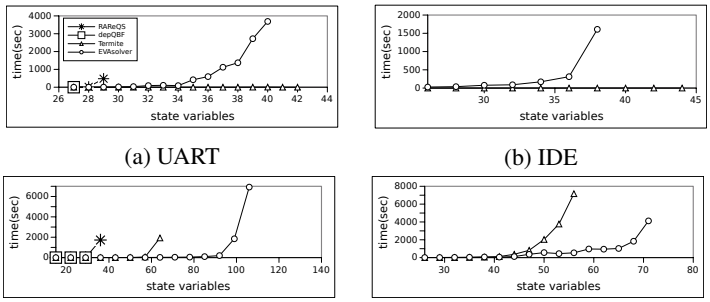
We find a candidate partial strategy that satisfies the above criterion using a SAT solver, as shown by the FINDCAND function. We unroll the transition relation δ into a formula ϕ that encodes a winning run of the game starting from the ith round. For each leaf l of the abstract game tree with the path from the root to the leaf labelled with environment actions $(e_i, \ldots, e_r)$, we construct a formula ϕ_l describing a winning run through the leaf. The formula consists of three conjuncts. The first conjunct $rename(\phi, l)$ renames variables in ϕ so that the resulting formulas for leaves sharing a common edge of the abstract game tree share the corresponding action and state variables, while using separate copies of all other variables. The second and third conjuncts fix initial state and environment actions along the path from the root to the leaf. We invoke a SAT solver to find assignments to state and action variables simultaneously satisfying all leaf formulas ϕ_l. If this formula is unsatisfiable, then state $\langle s, i\rangle$ is losing and the algorithm returns $\emptyset$; otherwise, it constructs a spoiling strategy by extracting values of controllable moves in nodes of the abstract game tree from the solution returned by the SAT solver.

Correctness of EVASOLVER follows from the following properties of the algorithm: (1) the counterexample-guided search strategy is complete, i.e., it is guaranteed to find a winning strategy, if one exists, possibly after exploring all possible runs of the game, and (2) our SAT encoding of the game is sound, i.e., if the SAT formula generated by FINDCAND is unsatisfiable then there does not exist a winning strategy from state $\langle s, i\rangle$.

Memoising losing states Our implementation of EVASOLVER uses an important optimisation. Whenever the SAT solver invocation in FINDCAND returns $unsat$, we obtain a proof that s is a losing state for the controller. We generalise this fact by extracting a minimal unsatisfiable core from the SAT solver and projecting it on state variables x. This gives us a cube of states losing for the controller. We modify the winning run formula ϕ to exclude this cube from a winning run. This guarantees that candidate partial strategies generated by the algorithm avoid previously discovered losing states.

6 Evaluation

We evaluate our algorithm on four families of benchmarks derived from driver synthesis problems. These benchmarks model the data path of four I/O devices in the abstracted form. In particular, we model the transmit buffer of an Ethernet adapter, the send queue of a UART serial controller, the command queue of an SPI Flash controller, and the IDE hard disk DMA descriptor list. Models are parameterised by the size of the corresponding data structure. Specifications are written in a simple input language based



(a) UART (b) IDE

(c) SPI Flash (d) Ethernet

Fig. 3: Performance of different solvers on four parameterised benchmarks. The X-axis shows the number of state vars in the game (determined by the benchmark parameter).

on the NuSMV syntax [8]. The transition relation of the game is given in the form of variable update functions $x := f(X, Y_c, Y_u)$, one for each state variable $x \in X$.

We compare our solver against two existing approaches to solving games. First, we encode input specifications as QBF instances and solve them using two state-of-the-art QBF solvers: RAReQS [10] and depqbf [12], having first run them through the bloqqer [3] preprocessor. Second, we solve our benchmarks using the Termite [17] BDD-based solver that uses dynamic variable reordering, variable grouping, transition relation partitioning, and other optimisations.

Our experiments, summarised in Figure 3, show that off-the-shelf QBF solvers are not well-suited for solving games. Although our algorithm is inspired by RAReQS, we achieve much better performance, since our solver takes into account the structure of the game, rather than treating it as a generic QBF problem.

All four benchmarks have very large sets of winning states. Nevertheless, in the UART and IDE benchmarks, Termite is able to represent winning states compactly with only a few thousand BDD nodes. It scales well and outperforms EVASOLVER on these benchmarks. However, in the two other benchmarks, Termite does not find a compact BDD-based representation of the winning set. EVASOLVER outperforms Termite on these benchmarks as it does not try to enumerate all winning states.

Detailed performance analysis shows that abstract game trees generated in our benchmarks had average branching factors in the range between 1.03 and 1.2, with the maximal depth of the trees ranging from 3 to 58. This confirms the the key premise behind the design of EVASOLVER, namely, solving real-world synthesis problems requires considering only a small number of opponent moves in every state of the game.

7 Conclusion

We presented a method for solving reachability games without constructing the game's winning set, and demonstrated that this method can be more efficient than conventional approaches. Our ongoing work concentrates on further performance improvements as well as on applying the new technique to a broader class of omega-regular games.

Our ongoing work focuses on further improving the performance of EVASOLVER via optimised CNF encodings of abstract games, stronger memoisation techniques, and additional domain-specific heuristics for computing candidate strategies.

References

1. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. In: CONCUR. pp. 74–89. Lisboa, Portugal (Sep 2007)
2. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. STTT 7(2), 118–128 (2005)
3. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: CADE. pp. 101–115. Wroclaw, Poland (Jul 2011)
4. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. ENTCS 190(4), 3–16 (Nov 2007)
5. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. CoRR abs/1311.3530 (2013)
6. Cassez, F.: Efficient on-the-fly algorithms for partially observable timed games. In: FORMATS. pp. 5–24. Salzburg, Austria (Oct 2007)
7. Cassez, F., Jessen, J.J., Larsen, K.G., Raskin, J.F., Reynier, P.A.: Automatic synthesis of robust and optimal controllers - an industrial case study. In: HSCC. pp. 90–104. San Francisco, CA, USA (Apr 2009)
8. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., River, A.T.: NuSMV 2.5 user manual
9. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: ICALP. pp. 886–902. Eindhoven, The Netherlands (Jul 2003)
10. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: SAT. pp. 114–128. Trento, Italy (Jun 2012)
11. Kupferman, O., Lustig, Y., Vardi, M.Y., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: STACS. pp. 615–626 (Mar 2011)
12. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: SAT. pp. 158–171. Edinburgh, UK (Jul 2010)
13. Morgenstern, A., Gesell, M., Schneider, K.: Solving games using incremental induction. In: IFM. pp. 177–191. Turku, Finland (Jun 2013)
14. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) designs. In: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 364–380. Charleston, SC, USA (Jan 2006)
15. Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., Heiser, G.: Automatic device driver synthesis with Termite. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles. Big Sky, MT, USA (Oct 2009)
16. Sabharwal, A., Ansotegui, C., Gomes, C.P., Hart, J.W., Selman, B.: QBF modeling: Exploiting player symmetry for simplicity and efficiency. In: SAT. pp. 382–395. Seattle, WA, USA (Aug 2006)
17. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. Technical Report