

A SAT-Based Counterexample Guided Method for Unbounded Synthesis

Alexander Legg¹, Nina Narodytska², and Leonid Ryzhyk²

¹ NICTA* and UNSW

`alexander.legg@nicta.com.au`

² Samsung Research America

Abstract. Reactive synthesis techniques based on constructing the winning region of the system have been shown to work well in many cases but suffer from state explosion in others. A different approach, proposed recently, applies SAT solvers in a counterexample guided framework to solve the synthesis problem. However, this method is limited to synthesising systems that execute for a bounded number of steps and is incomplete for synthesis with unbounded safety and reachability objectives. We present an extension of this technique to unbounded synthesis. Our method applies Craig interpolation to abstract game trees produced by counterexample-guided search in order to construct a monotonic sequence of may-losing regions. Experimental results based on SYNTCOMP 2015 competition benchmarks show this to be a promising alternative that solves some previously intractable instances.

1 Introduction

Reactive systems are ubiquitous in real-world problems such as circuit design, industrial automation, or device drivers. Automatic synthesis can provide a *correct by construction* controller for a reactive system from a specification. However, the reactive synthesis problem is 2EXPTIME-complete so naive algorithms are infeasible on even simple systems.

Reactive synthesis is formalised as a game between the *controller* and its *environment*. In this work we focus on safety games, in which the controller must prevent the environment from forcing the game into an error state. Much of the complexity of reactive synthesis stems from tracking the set of states in which a player is winning.

There are several techniques that aim to mitigate this complexity by representing states symbolically. Historically the most successful technique has been to use *Binary Decision Diagrams* (BDDs). BDDs efficiently represent a relation on a set of game variables but in the worst case the representation may be exponential. This means that BDDs are not a one-size-fits-all solution for all reactive synthesis specifications.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Advances in SAT solving technology has prompted research into its applicability to synthesis as an alternative to BDDs. One approach is to find sets of states in CNF [?]. Another approach is to eschew states and focus on *runs* of the game. Previous work has applied this idea to realizability of bounded games [3] by forming abstract representations of the game. In this paper, we extend this idea to unbounded games by constructing approximate sets of winning states from abstract trees.

2 Reactive Synthesis

A *safety game*, $G = \langle X, L_u, L_c, \delta, I, E, \rangle$, consists of a set of boolean state variables, sets of uncontrollable and controllable label variables, a transition relationship $\delta : (X, L_u, L_c) \rightarrow X$, an initial state, and an error state. The *controller* and *environment* players choose controllable and uncontrollable labels respectively and the game proceeds according to δ .

An *run* of a game $(x_0, u_0, c_0), (x_1, u_1, c_1) \dots (x_n, u_n, c_n)$ is a chain of state and label pairs of length n s.t. $x_{k+1} \leftarrow \delta(x_k, u_k, c_k)$. A run is winning for the controller if $x_0 = I \wedge \forall i \in \{1..n\} (x_i \neq E)$. In a bounded game of rank n all runs are restricted to length n , whereas unbounded games consider runs of infinite length. Since we consider only deterministic games a run is equivalent to a list of assignments to L_c and L_u .

A *controller strategy* $\pi^c : (X, L_u) \rightarrow L_c$ is a mapping of states and uncontrollable inputs to controllable labels. A controller strategy is winning in a bounded game of rank n if all runs $(x_0, u_0, \pi^c(x_0, u_0)), (x_1, u_1, \pi^c(x_1, u_1)) \dots (x_n, u_n, \pi^c(x_n, u_n))$ are winning. Bounded *realizability* is the problem of determining the existence of such a strategy for a bounded game.

An *environment strategy* $\pi^e : X \rightarrow L_u$ is a mapping of states to uncontrollable labels. A bounded run is winning for the environment if $x_0 = I \wedge \exists i \in \{1..n\} (x_i = E)$ and an environment strategy is winning for a bounded game if there exists a run $(x_0, \pi^e(x_1), c_1), (x_1, \pi^e(x_1), c_1) \dots (x_n, \pi^e(x_n), c_n)$ that wins for the environment. Safety games are zero sum, therefore the existence of a controller strategy implies the nonexistence of an environment strategy and vice versa.

2.1 Abstract Game Trees

A set of runs can be represented symbolically by a tree of valuations to controllable and uncontrollable label variables where valuations are allowed to be *unfixed*. An unfixed value symbolically represents all possible valuations. Formally, we define the tree as a set of lists of label valuations and a node in the tree may be identified by the list made by following the path from the root.

The entire set of runs of a bounded game of rank n is symbolically represented by a tree of depth $2n$ and width 1 populated by unfixed edges. Reducing the set of runs in a game forms an abstract game, which can be represented symbolically by an *abstract game tree*.

A strategy is equivalent to the set of all runs with the player's labels obeying the strategy mapping π . Therefore, a strategy can also be represented by a tree. Relaxing the restriction of the strategy mapping allows for a *partial strategy* in which multiple labels are now possible for a single state.

A strategy or partial strategy can also be thought of as an abstract game. A partial strategy for the controller is a restriction only on the controllable labels in the game. So if the environment can not win in the abstract game equivalent to the controller's partial strategy, then all strategies allowable by that partial strategy must be winning.

An abstract game tree can be checked for the existence of a winning run by a SAT solver[3]. The tree must be encoded into CNF by making copies of the transition relation for each game step in tree. The formula must also check whether the error state has been reached in no branch/in all branches for the controller and environment respectively. The functions that produce these formulas are shown in Algorithm 4.

Algorithm 1 Tree formulas for Controller and Environment respectively

```

1: function TREEFORMULA(gt)
2:   if RANK(gt) == 0 then
3:     return  $\neg E(x^{gt})$ 
4:   else
5:     return  $\neg E(x^{gt}) \wedge \bigwedge_{n \in \text{SUCC}(gt)} (\delta(n) \wedge \text{LABEL}(n) \wedge \text{TREEFORMULA}(n))$ 
6:   end if
7: end function
8: function TREEFORMULA(gt)
9:   if RANK(gt) == 0 then
10:    return  $E(x^{gt})$ 
11:   else
12:    return  $E(x^{gt}) \vee \bigvee_{n \in \text{SUCC}(gt)} (\delta(n) \wedge \text{LABEL}(n) \wedge \overline{\text{TREEFORMULA}(n)})$ 
13:   end if
14: end function

```

When we produce a formula using TREEFORMULA the SAT solver is playing on behalf on the controller. Any unfixed labels in the tree, controllable or uncontrollable, will be existentially quantified. This means that if there exists a way for the game to be solved when both players cooperate the SAT solver will find it. If no winning run exists in an abstract game even when the players are cooperating then there is definitely no winning run when the opponent is playing adversarily.

2.2 Counterexample Guided Bounded Synthesis

By checking for the existence of a winning run for the environment in an abstract game tree constructed from a partial controller strategy we are checking if the

partial strategy is winning. If no spoiling run can be found then the partial strategy must always win. If a spoiling run is found then we then we have a counterexample that proves that the controller strategy is not winning. This forms the basis of a counterexample guided abstraction refinement framework that operates on candidate strategies.

This technique was proposed by Narodytska et al [3]. In Section 3 we propose an extension to synthesis of unbounded games.

The bounded synthesis algorithm begins with an empty game as seen in Figure 1a. Initially we are playing on behalf of the environment because it chooses the first move in each step. The empty game is passed to the SAT solver, which searches for a candidate environment strategy. If a candidate is found (Figure 1b) then it is checked for a spoiling strategy by solving for the controller. If no spoiling strategy exists, that means our candidate is a winning strategy and the algorithm terminates. Otherwise we find a counterexample (Figure 1c), which is used to refine the empty game tree to include the first move from the controller's spoiling strategy (Figure 1d). The algorithm continues by finding a new candidate for the environment (Figure 1e), and a new counterexample to refine the game abstraction once again (Figure 1f). The full algorithm is listed in Algorithm 2.

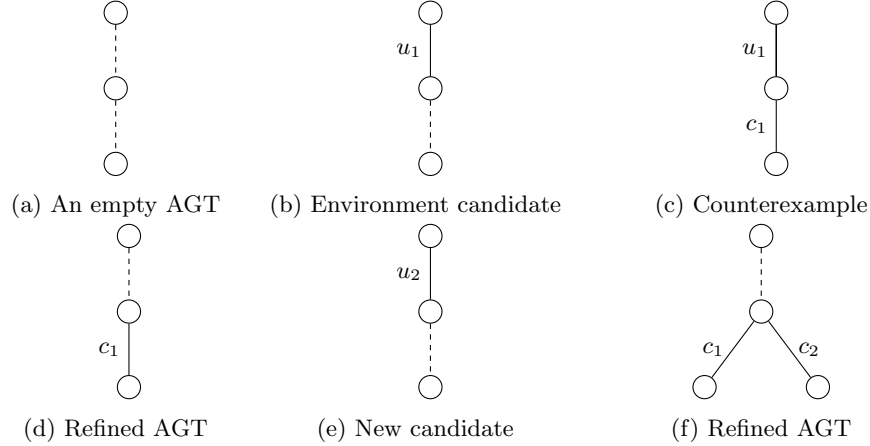


Fig. 1: Abstract game trees

3 Unbounded Synthesis

Bounded synthesis can be used to prove the existence of a winning strategy for the environment by providing a witness. For the controller, the strongest claim that can be made is that the strategy is winning as long as the game does not extend longer than the bound.

Algorithm 2 Bounded Synthesis

```

1: function SOLVEABSTRACT( $\langle s, k \rangle, T$ )
2:    $cand \leftarrow \text{FINDCANDIDATE}(\langle s, k \rangle, T)$  ▷ Look for a candidate
3:   if  $k = n - 1$  then return  $cand$  ▷ Reached the bound
4:    $T' \leftarrow T$ 
5:   loop
6:     if  $cand = \emptyset$  then return  $\emptyset$  ▷ No candidate: return with no solution
7:      $\langle cex, l, u \rangle \leftarrow \text{VERIFY}(\langle s, k \rangle, T, cand)$  ▷ Verify candidate
8:     if  $cex = \text{false}$  then return  $cand$  ▷ No counterexample: return candidate
9:      $T' \leftarrow \text{GTAPPEND}(T', l, u)$  ▷ Refine AGT with counterexample
10:     $cand \leftarrow \text{SOLVEABSTRACT}(\langle s, k \rangle, T')$  ▷ Solve refined AGT
11:  end loop
12: end function
13: function FINDCANDIDATE( $\langle s, k \rangle, T$ )
14:    $sol \leftarrow \text{SAT}(s(X_T) \wedge \text{TREEFORMULA}(T))$ 
15:   if  $sol = \text{unsat}$  then return  $\emptyset$  ▷ No candidate exists
16:   return  $\{\langle n, c \rangle \mid n \in \text{GTNODES}(T), c = \text{SOL}(n)\}$  ▷ Fix candidate moves in  $T$ 
17: end function
18: function VERIFY( $\langle s, k \rangle, gt, cand$ )
19:   for  $l \in \text{leaves}(gt)$  do
20:      $\langle k', s' \rangle \leftarrow \text{OUTCOME}(\langle s, k \rangle, gt, l)$  ▷ Get rank and state at leaf
21:      $u \leftarrow \text{SOLVEABSTRACT}(\langle k', s' \rangle, \emptyset)$  ▷ Solve for the opponent
22:     if  $u \neq \emptyset$  then return  $\langle \text{false}, l, u \rangle$  ▷ Return counterexample
23:   end for
24:   return  $\langle \text{true}, \emptyset, \emptyset \rangle$ 
25: end function

```

In model checking the maximum bound is decided based on the states of the game [1]. The naïve approach is to use size of the state space as the bound ($2^{|X|}$). With a bound of this size all states may be explored. One optimisation is to use the diameter of the game, which is the smallest number d such that for any state x there is a path of length $\leq d$ to all other reachable states. However, for large games these bounds are intractable.

When performing model checking or synthesis with BDDs [2] the set of states that are winning for the environment is iteratively constructed by computing the states from which the environment can force the game into the previous winning set. Eventually this process reaches a fixed point and the total set of environment winning states is known.

A similar concept can be applied to the bounded synthesis algorithm to iteratively increase the bound of the game and terminate when a fixed point is reached. When a strategy is found to be winning on an abstract game tree, we record as winning the states from which the opponent could find no counterexample. To find these states we use interpolation of subformulas of the game tree.

3.1 Learning States with Interpolants

Given two formulas A and B such that $A \wedge B$ is unsatisfiable, it is possible to construct an interpolant \mathcal{I} such that $A \rightarrow \mathcal{I}$, $B \wedge \mathcal{I}$ is unsatisfiable, and \mathcal{I} refers only to the intersection of variables in A and B . An interpolant can be constructed efficiently from a resolution proof of the unsatisfiability of $A \wedge B$ [4].

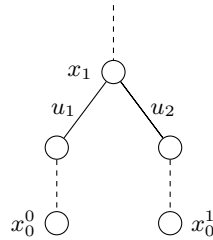


Fig. 2: A controller-losing game tree

Consider the snippet of a game tree in Figure 2. The tree is losing for the controller, the node labelled x_1 is at rank 1, and x_0^0 and x_0^1 are at rank 0. Since the tree is controller-losing we know that at least one run represented by the tree contains the error state. As a result, $\text{TREEFORMULA}(gt)$ is unsatisfiable. If we take the step x_1 to x_0^0 and cut it from the rest of the tree then $\text{TREEFORMULA}(step) \wedge \text{TREEFORMULA}(parent)$ must still be unsatisfiable.

We can construct an interpolant with $A = \text{TREEFORMULA}(parent)$ and $B = \text{TREEFORMULA}(step)$. The only variables shared between A and B are the state

variables at x_1 . We know that $B \wedge \mathcal{I}$ is unsatisfiable, therefore all states in \mathcal{I} must lose to the uncontrollable label u_1 . We also know that $A \rightarrow \mathcal{I}$, thus \mathcal{I} contains all states reachable by the parent tree (on runs that avoid the error state.)

Algorithm 3 Learning algorithms

Require: $\sigma(X_T) \wedge \text{TREEFORMULA}(T) \equiv \perp$

Require: *Must-invariant* holds

Ensure: *Must-invariant* holds

Ensure: $\sigma(X_T) \wedge B^M$

1: **function** LEARN(σ, T)

2: **if** $T = \emptyset$ **then return**

3: $n \leftarrow$ non-leaf node with max rank

4: $\langle T_1, T_2 \rangle \leftarrow \text{GTSPLIT}(T, n)$

5: $\mathcal{I} \leftarrow \text{INTERPOLATE}(\sigma(X_T) \wedge \text{TREEFORMULA}(T_1), \text{TREEFORMULA}(T_2))$

6: $B^M \leftarrow B^M \vee \mathcal{I}$

7: LEARN(σ, T_1)

8: **end function**

Require: $\sigma(X_T) \wedge \overline{\text{TREEFORMULA}}(T) \equiv \perp$

Require: *May-invariant* holds

Ensure: *May-invariant* holds

Ensure: $\sigma(X_T) \wedge B^m[\text{RANK}(T)] \equiv \perp$

9: **function** LEARN(σ, T)

10: **if** $T = \emptyset$ **then return**

11: $n \leftarrow$ non-leaf node with max rank

12: $\langle T_1, T_2 \rangle \leftarrow \text{GTSPLIT}(T, n)$

13: $\mathcal{I} \leftarrow \text{INTERPOLATE}(\sigma(X_T) \wedge \overline{\text{TREEFORMULA}}(T_1), \overline{\text{TREEFORMULA}}(T_2))$

14: **for** $i = 1$ to $\text{RANK}(n)$ **do**

15: $B^m[i] \leftarrow B^m[i] \wedge \neg \mathcal{I}$

16: **end for**

17: $\overline{\text{LEARN}}(\sigma, T_1)$

18: **end function**

Now we can consider the step x_1 to x_0^1 , and the parent - now without either $x_1 \rightarrow x_0^0$ or $x_1 \rightarrow x_0^1$. The formula $(\text{TREEFORMULA}(\text{parent}) \wedge \mathcal{I}) \wedge \text{TREEFORMULA}(\text{step})$ must be unsatisfiable. \mathcal{I} contains all states that lose to u_1 so any other state reachable at x_1 must lose to u_2 . Therefore we can compute another interpolant that contains states that lose to u_2 .

This is the foundation for a recursive algorithm that consumes an entire tree by removing a single step on each iteration (Algorithm 3). All learned states from which the controller *must* lose are recorded in a set of *bad* states B^M . This algorithm can also be performed on environment-losing trees with the caveat that any state learnt at a node of rank n is only known to lose at ranks less than or equal to n . We negate the interpolants we learn this way and record them in a mapping of ranks to sets of states B^m which *may* be bad states for the controller.

Algorithm 4 Amended tree formulas for Controller and Environment respectively

```

1: function TREEFORMULA(gt)
2:   if RANK(gt) == 0 then
3:     return  $\neg B^M(x^{gt})$ 
4:   else
5:     return  $\neg B^M(x^{gt}) \wedge \bigwedge_{n \in \text{SUCC}(gt)} (\delta(n) \wedge \text{LABEL}(n) \wedge \text{TREEFORMULA}(n))$ 
6:   end if
7: end function

8: function  $\overline{\text{TREEFORMULA}}$ (gt)
9:   if RANK(gt) == 0 then
10:    return  $E(x^{gt})$ 
11:   else
12:    return  $B^m[\text{rank}(gt)](x^{gt}) \wedge$ 
13:     $(E(x^{gt}) \vee \bigvee_{n \in \text{SUCC}(gt)} (\delta(n) \wedge \text{LABEL}(n) \wedge \overline{\text{treeFormula}}(n)))$ 
14:   end if
15: end function

```

3.2 Correctness

The unbounded synthesis algorithm (Algorithm 5) effectively consists of two communicating solvers: the counterexample-guided bounded reachability solver and the unbounded solver based on incremental induction. Fortunately, correctness of both solvers can be established largely independently, which simplifies the proof of correctness of the overall algorithm.

We define two global invariants of the algorithm. The *may-invariant* states that sets $B^m[i]$ grows monotonically with i and that each $B^m[i+1]$ overapproximates the uncontrollable predecessor of $f^m[i]$:

$$\forall i < k. B^m[i] \subseteq B^m[i+1], \text{Upred}(B^m[i]) \subseteq B^m[i+1].$$

The *must-invariant* guarantees that the must-losing set B^M is an underapproximation of the actual losing set B :

$$B^M \subseteq B$$

The sets B^m and B^M are only modified by the inductive solver, implemented by `LEARN` and `$\overline{\text{LEARN}}$` functions. Below we prove that these functions indeed maintain the invariants.

3.3 Proof of $\overline{\text{learn}}$

We prove that postconditions of `$\overline{\text{LEARN}}$` are satisfied assuming that its preconditions hold.

Line (11) splits the tree T into T_1 and T_2 , such that T_2 has depth 1. Consider formulas $f_1 = \sigma(X_T) \wedge \text{TREEFORMULA}(T_1)$ and $F_2 = \text{TREEFORMULA}(T_2)$. These

Algorithm 5 Unbounded Synthesis

```

1: function SOLVEUNBOUNDED( $T$ )
2:    $B^M \leftarrow E$ 
3:    $B^m[0] \leftarrow E$ 
4:   for  $k = 1 \dots$  do
5:     if  $\text{SAT}(I \wedge B^M)$  then return unrealisable  $\triangleright$  Losing in the initial state
6:     if  $\exists i \ B^m[i] \equiv B^m[i+1]$  then return realisable  $\triangleright$  Reached fixed point
7:      $B^m[k] \leftarrow \top$ 
8:   end for
9: end function

Require: May and must invariants hold
Ensure: May and must invariants hold
Ensure:  $\neg s \in B^m[k]$  if there is a winning controller strategy of length  $k$  starting at  $s$ 
Ensure:  $s \in B^M$  if there is a winning environment strategy of length  $k$  starting at  $s$ 
10: function CHECKRANK( $s, k$ )
11:   return SOLVEABSTRACT( $\langle s, k \rangle, \emptyset$ )
12: end function
13: function SOLVEABSTRACT( $\langle s, k \rangle, T$ )
14:    $cand \leftarrow \text{FINDCANDIDATE}(\langle s, k \rangle, T)$   $\triangleright$  Look for a candidate
15:   if  $k = n - 1$  then return  $cand$   $\triangleright$  Reached the bound
16:    $T' \leftarrow T$ 
17:   loop
18:     if  $cand = \emptyset$  then return  $\emptyset$   $\triangleright$  No candidate: return with no solution
19:      $\langle cex, l, u \rangle \leftarrow \text{VERIFY}(\langle s, k \rangle, T, cand)$   $\triangleright$  Verify candidate
20:     if  $cex = \text{false}$  then return  $cand$   $\triangleright$  No counterexample: return candidate
21:      $T' \leftarrow \text{GTAPPEND}(T', l, u)$   $\triangleright$  Refine AGT with counterexample
22:      $cand \leftarrow \text{SOLVEABSTRACT}(\langle s, k \rangle, T')$   $\triangleright$  Solve refined AGT
23:   end loop
24: end function
25: function FINDCANDIDATE( $\langle s, k \rangle, T$ )
26:    $sol \leftarrow \text{SAT}(s(X_T) \wedge \text{TREEFORMULA}(T))$ 
27:   if  $sol = \text{unsat}$  then
28:      $\sigma \leftarrow \text{GENERALISE}(s)$   $\triangleright$  Expand  $s$  to a set of states
29:      $\text{LEARN}(\sigma, T)$ 
30:     return  $\emptyset$   $\triangleright$  No candidate exists
31:   else
32:     return  $\{\langle n, c \rangle \mid n \in \text{GTNODES}(T), c = \text{SOL}(n)\}$   $\triangleright$  Fix candidate moves in  $T$ 
33:   end if
34: end function
35: function VERIFY( $\langle s, k \rangle, gt, cand$ )
36:   for  $l \in \text{leaves}(gt)$  do
37:      $\langle k', s' \rangle \leftarrow \text{OUTCOME}(\langle s, k \rangle, gt, l)$   $\triangleright$  Get rank and state at leaf
38:      $u \leftarrow \text{SOLVEABSTRACT}(\langle k', s' \rangle, \emptyset)$   $\triangleright$  Solve for the opponent
39:     if  $u \neq \emptyset$  then return  $\langle \text{false}, l, u \rangle$   $\triangleright$  Return counterexample
40:   end for
41:   return  $\langle \text{true}, \emptyset, \emptyset \rangle$ 
42: end function

```

formulas only share variables X_n . Their conjunction $F_1 \wedge F_2$ is unsatisfiable, as by construction any solution of $F_1 \wedge F_2$ also satisfies $\sigma(X_T) \wedge \text{TREEFORMULA}(T)$, which is unsatisfiable (precondition (b)). Hence the interpolation operation is defined for F_1 and F_2 .

Intuitively, the interpolant computed in line (13) overapproximates the set of states reachable from σ by following the tree from the root node to n , and underapproximates the set of states from which the environment loses against tree T_2 .

Formally, \mathcal{I} has the property $\mathcal{I} \wedge F_2 \equiv \perp$. Since T_2 is of depth 1, this means that the environment cannot force the game into $B^m[\text{RANK}(n)]$ playing against the counterexample moves in T_2 . Hence, $\mathcal{I} \cap \text{Upred}(B^m[\text{RANK}(n)]) = \emptyset$. Furthermore, since the may-invariant holds, $\mathcal{I} \cap \text{Upred}(i) = \emptyset$, for all $i < \text{RANK}(n)$. Hence, removing \mathcal{I} from all $B^m[i]$, $i \leq \text{RANK}(n)$ in line (15) preserves the may-invariant, thus satisfying the first post-condition.

Furthermore, the interpolant satisfies $F_1 \rightarrow \mathcal{I}$, i.e., any assignment to X_n that satisfies $\sigma(X_T) \wedge |\text{treeFormula}|(T_1)$ also satisfies \mathcal{I} . Hence, removing \mathcal{I} from $B^m[\text{RANK}(n)]$ makes $\sigma(X_T) \wedge |\text{treeFormula}|(T_1)$ unsatisfiable, and hence all preconditions of the recursive invocation of $\overline{\text{LEARN}}$ in line (17) are satisfied.

At the second last recursive call to $\overline{\text{LEARN}}$, tree T_1 is empty, n is the root node, $\overline{\text{TREEFORMULA}}(T_1) \equiv B^m[\text{RANK}(T)](X^T)$; hence $\sigma(X_T) \wedge \overline{\text{TREEFORMULA}}(T_1) \equiv \sigma(X_T) \wedge B^m[\text{RANK}(T)](X^T) \equiv \perp$. Thus the second postcondition of $\overline{\text{LEARN}}$ holds.

3.4 Proof of Termination

We must prove that CHECKRANK terminates and that upon termination its postcondition holds.

We must prove that CHECKRANK terminates and that upon termination its postcondition holds, i.e., state s is removed from $B^m[k]$ if there is a winning controller strategy on the bounded safety game of rank k or it is added to B^M otherwise. Termination follows from completeness of counterexample-guided search, which terminates after enumerating all possible opponent moves in the worst case.

Assume that there is a winning strategy for the controller at rank k . This means that at some point the algorithm discovers a counterexample tree of rank k for which the environment cannot force into E . The algorithm then invokes the $\overline{\text{LEARN}}$ method, adds s to B^M . Alternatively, if there is a winning strategy for the environment at rank k then a counterexample losing for the controller will be found. Subsequently LEARN will be called and s eliminated from $B^m[k]$.

4 Optimisations

4.1 Generalising the initial state

4.2 Choosing sensible opponent moves

5 Evaluation

6 Related Work

7 Conclusion

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W. (ed.) *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer Berlin Heidelberg (1999)
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: *Symposium on Logic in Computer Science*. pp. 428–439 (1990)
3. Narodytska, N., Legg, A., Bacchus, F., Ryzhyk, L., Walker, A.: Solving games without controllable predecessor. In: *Computer Aided Verification*. pp. 533–540 (2014)
4. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic* 62(3), 981–998 (1997)