A SAT-Based Counterexample Guided Method for Unbounded Synthesis

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 anything but 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. In this way the environment is equivalent to an existential search of the game and the controller is universal. Much of the complexity of reactive synthesis stems from managing the interactions of the alternating quantifiers on the state space of the game.

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 efficiently represent a relation on a set of game variables but in the worst case the representation may be exponential in the number of variables. As a result BDDs are not a one-size-fits-all solution for all reactive synthesis specifications.

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 [3, 15]. Another approach is to eschew states and focus on *runs* of the game. Previous work has applied this idea to realizability of bounded games [16] by forming abstractions of the game and refining in a counterexample-guided

framework. This has been shown to outperform BDDs on certain classes of game specifications but is only able to find counterexamples to the safety property of a certain length. In this paper, we extend this idea to unbounded games by approximating sets of unsafe states from abstract games. Careful construction ensures that a fixed point in these sets guarantees completeness.

Section 2 outlines the original bounded synthesis algorithm. In Section 3 we describe and prove the correctness of our extension of the algorithm to unbounded games. In the following sections we discuss optimisations to the algorithm, evaluate our methodology, and compare our approach to other synthesis techniques.

2 Background

A *safety game*, $G = \langle X, L_u, L_c, \delta, I, E \rangle$, is defined over boolean state variables X, uncontrollable label variables L_u, and controllable label variables L_c. I is the initial state of the game given as a valuation of state variables. $E(X)$ is the set of error states represented by its characteristic formula. The transition relation $\delta(X, L_u, L_c, X')$ of the game is a boolean formula that relates current state and label to the set of possible next states of the game. We assume deterministic games, where $\delta(x, u, c, x'_1) \wedge \delta(x, u, c, x'_2) \implies (x'_1 = x'_2)$.

At every round of the game, the *environment* picks an uncontrollable label, the *controller* responds by choosing a controllable label and the game transitions into a new state according to δ. A *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 s.t. $\delta(x_k, u_k, c_k, x_{k+1})$. A run is winning for the controller if $x_0 = I \wedge \forall i \in \{1..n\}(\neg E(x_i))$. In a *bounded game* with bound 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 uniquely described by a list of assignments to L_u and L_c.

A *controller strategy* $\pi^c : (X, L_u) \to L_c$ 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.

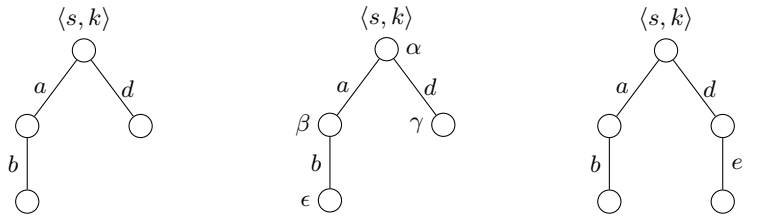
An *environment strategy* $\pi^e : X \to L_u$ is a mapping of states to uncontrollable labels. A bounded run is winning for the environment if $x_0 = I \wedge \exists i \in \{1..n\}(E(x_i))$ and an environment strategy is winning for a bounded game if all runs $(x_0, \pi^e(x_1), c_1), (x_1, \pi^e(x_1),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

2.1 Counterexample-guided bounded synthesis

We review the bounded synthesis algorithm by Narodytska et al. [16], which is the main building block for our unbounded algorithm.

Realisability for a bounded safety game can be solved by checking a quantified formula consisting of the game's transition relation *unrolled* to the length of the bounded game. The quantifiers represent the two players of the game: universal for the controller and existential for the environment. Just as the players alternate in choosing actions every step of the game, the formula is prefixed by a quantifier alternation over L_u and L_c for each step.

Instead of solving both quantifiers at once, two competing solvers are used to solve abstractions of the game. Each solver builds a candidate strategy for its corresponding player and uses its opponent's candidate to restrict its own search. The solvers communicate to refine one another's candidate strategies by finding counterexamples. Eventually one solver will be unable to find a counterexample, indicating that its opponent's candidate strategy is a winning strategy for the game.



(a) AGT (b) AGT with partial strategy (c) Refined AGT

Fig. 1: Abstract game trees

An abstraction of the game restricts actions available to one of the players. Specifically, we consider abstractions represented as trees of actions, referred to as *abstract game trees*. Figure 1a shows an example abstract game tree for the controller (abstract game trees for the environment are similar). In the abstract game, the controller 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 controller action to the set $\{a, d\}$. After choosing action a, the controller is required to play action b in the next round. It then reaches a leaf of the tree and continues playing unrestricted. The root of the tree is annotated by the initial state s of the abstract game and the bound k on the number of rounds. We denote $\textsc{nodes}(T)$ the set of all nodes of a tree T and $\textsc{leaves}(T)$ the subset of leaf nodes.

Given a controller (environment) abstract game tree T a *partial strategy* $Strat : \textsc{nodes}(T) \to L_u$ ($Strat : \textsc{nodes}(T) \to L_c$) labels each node of the tree with the opponent's action to be played in that node. Figure 1b shows an example partial strategy. The environment starts by choosing action α. If the controller plays a, the environment 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 $\langle s', i' \rangle = \text{OUTCOME}(\langle s, i \rangle, Strat, l)$ obtained by playing all controllable and uncontrollable actions on the path from the root to the leaf.

The bounded synthesis algorithm solves abstract game trees by invoking a SAT solver to search for candidate partial strategies. Candidate strategies are checked and by a call to the opponent solver, which also refines the abstract game tree. The refinements are then solved recursively. The full procedure is illustrated in Algorithm 1.

Algorithm 1 Bounded Synthesis

```
1: function solveAbstract(p, s, k, T)
2:     cand ← findCandidate(p, s, k, T)                    ▷ Look for a candidate
3:     if k = 1 then return cand                            ▷ Reached the bound
4:     T' ← T
5:     loop
6:         if cand = ∅ then return ∅          ▷ No candidate: return with no solution
7:         ⟨cex, l, u⟩ ← verify(p, s, k, T, cand)                ▷ Verify candidate
8:         if cex = false then return cand  ▷ No counterexample: return candidate
9:         T' ← append(T', l, u)                ▷ Refine AGT with counterexample
10:        cand ← solveAbstract(p, s, k, T')                    ▷ Solve refined AGT
11:    end loop
12: end function
13: function findCandidate(p, s, k, T)
14:    f ← if p = cont then treeFormula(T) else ¬treeFormula(T)
15:    sol ← SAT(s(X_T) ∧ f)
16:    if sol = unsat then
17:        if unbounded then                ▷ Active only in the unbounded solver
18:            σ ← generalise(s)                          ▷ Expand s to a set of states
19:            if p = cont then learn(σ, T) else ¬learn(σ, T)
20:        end if
21:        return ∅                                              ▷ No candidate exists
22:    else
23:        return {⟨n, c⟩ | n ∈ gtNodes(T) , c = sol(n)}  ▷ Fix candidate moves in T
24:    end if
25: end function
26: function verify(p, s, k, T, cand)
27:    for l ∈ leaves(gt) do
28:        ⟨k', s'⟩ ← outcome(s, k, cand, l)                ▷ Get rank and state at leaf
29:        u ← solveAbstract(opponent(p), k', s', ∅)          ▷ Solve for the opponent
30:        if u ≠ ∅ then return ⟨false, l, u⟩                 ▷ Return counterexample
31:    end for
32:    return ⟨true, ∅, ∅⟩
33: end function
```

The algorithm takes a concrete game G with maximum bound n as an implicit argument. In addition, it takes a player p (controller or environment), state s, bound k and an abstract game tree T and returns a winning partial strategy

for OPPONENT(p), if one exists. The initial invocation of the algorithm takes the initial state $\langle I, n\rangle$ and an empty abstract game tree $\emptyset$. Initially the solver is playing on behalf of the environment since that player takes the first move in every step. 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 FINDCANDIDATE function, described below, to come up with a candidate partial strategy for T. If it fails to find a strategy, this means that no winning partial strategy exists for T. If, on the other hand, a candidate partial strategy is found, we need to verify if it is indeed winning for T.

The VERIFY procedure searches for a *spoiling* counterexample strategy in each leaf of the candidate partial strategy by calling SOLVEABSTRACT for the opponent. The dual solver solves games on behalf of the controller player.

If the dual solver can find no spoiling strategy at any of the leaves, then the candidate partial strategy is a winning one. Otherwise, VERIFY returns the move used by the opponent to defeat a leaf of the partial strategy, which is appended to the corresponding node in T in order to refine it in line (9) as shown in Figure (1c).

We solve the refined game by recursively invoking SOLVEABSTRACT 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. In the worst case the loop terminates after all actions in the game are refined into the abstract game.

Candidates are discovered by passing a formulation of the abstract game tree to a SAT solver in FINDCANDIDATE. This formula contains CNF encodings of all of the unrolled runs represented by the tree and the winning condition of the current player. Runs are encoded by copying the transition relation for every step in the abstract game. When playing for the controller, the SAT solver searches for a satisfying assignment to the unfixed label variables in tree so that none of the runs reaches the error state. The environment formulation is satisfiable if any run does reach the error state. The formula is constructed recursively from the root of a tree by TREEFORMULA (see Algorithm 2).

Since the game tree formulation is passed to a SAT solver, both controllable and uncontrollable unfixed labels will be existentially quantified. This means that the SAT solver will find any way to win the game while both players are cooperating. If no winning run exists in an abstract game even when the players are cooperating then there is no winning run when the opponent is playing adversarily. When a winning run is found, the actions chosen by the SAT solver are used to refine the game tree. This is advantageous for many synthesis problems where the game must be formalised as adversarial for correctness but the final implementation will cooperate with its environment in the real world. An

example of such a system is a device driver that cooperates with the device and OS to provide the interface between the two.

Algorithm 2 Tree formulas for Controller and Environment respectively

1: **function** TREEFORMULA(T)
2: **if** RANK(T) $= 0$ **then**
3: **return** $\neg \mathrm{E}(X_T)$
4: **else**
5: **return** $\neg \mathrm{E}(X_T) \wedge$
6: $$\bigwedge_{n \in \mathrm{SUCC}(T)} (\delta(X_n, U_n, C_n, X'_n) \wedge \mathrm{LABEL}(n) \wedge \mathrm{TREEFORMULA}(n))$$
7: **end if**
8: **end function**
9: **function** $\overline{\mathrm{TREEFORMULA}}(T)$
10: **if** RANK(T) $= 0$ **then**
11: **return** $\mathrm{E}(X_T)$
12: **else**
13: **return** $\mathrm{E}(X_T) \vee$
14: $$\bigvee_{n \in \mathrm{SUCC}(T)} (\delta(X_n, U_n, C_n, X'_n) \wedge \mathrm{LABEL}(n) \wedge \overline{\mathrm{TREEFORMULA}}(n))$$
15: **end if**
16: **end function**

3 Unbounded Synthesis

Bounded synthesis can be used to prove the existence of a winning strategy for the environment on the unbounded game 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 beyond the bound.

It is possible to set a bound such that all runs in the unbounded game will be considered. The naïve approach is to use size of the state space as the bound ($2^{|X|}$) so that all states may be explored by the algorithm. A more nuanced approach is to use the diameter of the game [2], 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. This approach is common in bounded model checking but for synthesis it quickly becomes intractable to consider such long runs.

When performing model checking or synthesis with BDDs [6] 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.

We apply a similar fixed point computation on game states to the bounded synthesis algorithm in order to promote it to unbounded games. Our unbounded procedure iteratively executes the bounded algorithm while recording states from abstract game trees. By carefully managing the way states are recorded we guarantee that a fixed point proves the controller's strategy correct.

3.1 Extending Bounded Synthesis to Unbounded Games

Algorithm 3 Unbounded Synthesis

```
1: function solveUnbounded(T)
2:     B^M ← E
3:     B^m[0] ← E
4:     for k = 1 ... do
5:         if SAT(I ∧ B^M) then return unrealisable   ▷ Losing in the initial state
6:         if ∃i B^m[i] ≡ B^m[i + 1] then return realisable   ▷ Reached fixed point
7:         B^m[k] ← ⊤
8:         checkRank(I, k)
9:     end for
10: 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

```
11: function checkRank(s, k)
12:     return solveAbstract(env, ⟨s, k⟩, ∅)
13: end function
```

The unbounded synthesis algorithm enhances the bounded method with computational learning. States are carefully recorded from unsatisfiable abstract games while maintaining an invariant ensuring the monotonicity of learning.

During the CEGAR loop of the bounded algorithm, each of the competing solvers searches for satisfying assignments of labels to abstract game trees. When the abstract game is unsatisfiable this indicates that the states represented by nodes in the abstract game tree are losing for the current player. We extract these states from the game tree using interpolation.

When executing the unbounded solver, lines 18 and 19 become active in the bounded solver. These lines call the learning procedures when the solver fails to find a candidate for an abstract game tree. The states symbolically represented by nodes in the tree are losing for whichever player could not find a winning candidate and can be extracted from the tree using interpolation.

The states in an abstract game with no controller candidate are *must-losing*. The environment can always force the game into the error states from these states.

From abstractions with no environment candidate we record the complement of states in the tree as *may-losing*. The environment cannot reach the error state in a number of steps equal to the distance to the bottom of the tree.

We maintain a set of states for each rank up to the current bound. We maintain an invariant over these sets via careful construction so that they are monotonically increasing by rank. We also ensure that the environment is unable to force play from one set to the next. Due to these invariants, when two adjacent sets become equivalent we know that the algorithm has reached a fixed point and the controller is winning in the unbounded game (line 6).

3.2 Learning States with Interpolants

Given two formulas F_1 and F_2 such that $F_1 \wedge F_2$ is unsatisfiable, it is possible to construct a Craig interpolant[7] $\mathcal{I}$ such that $F_1 \rightarrow \mathcal{I}$, $F_2 \wedge \mathcal{I}$ is unsatisfiable, and $\mathcal{I}$ refers only to the intersection of variables in F_1 and F_2. An interpolant can be constructed efficiently from a resolution proof of the unsatisfiability of $F_1 \wedge F_2$ [17].

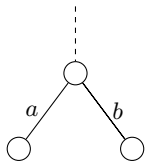


Fig. 2: A controller-losing game tree

Consider the snippet of a game tree in Figure 2. The tree is losing for the controller, the topmost node is at rank 1, and the a and b nodes are at rank 0. Since the tree is controller-losing we know that at least one state in the tree will uncontrollably lead to the error state. As a result, $\textsc{treeFormula}(T)$ is unsatisfiable. If we take a step of the tree, such as the step where the environment plays a, and cut it from the rest of the tree then $\textsc{treeFormula}(\mathit{step}) \wedge \textsc{treeFormula}(\mathit{parent})$ must too be unsatisfiable.

We can construct an interpolant with $F_1 = \textsc{treeFormula}(\mathit{parent})$ and $F_2 = \textsc{treeFormula}(\mathit{step})$. The only variables shared between F_1 and F_2 are the state variables at rank 1. We know that $F_2 \wedge \mathcal{I}$ is unsatisfiable, therefore all states in $\mathcal{I}$ must lose to the uncontrollable action a. We also know that $F_1 \rightarrow \mathcal{I}$, thus $\mathcal{I}$ contains all states reachable by the parent tree (on runs that avoid the error state.)

Now we can consider the step where the environment plays b, and the parent now without either step in the snippet. The formula $(\textsc{treeFormula}(\mathit{parent}) \wedge \mathcal{I}) \wedge \textsc{treeFormula}(\mathit{step})$ must be unsatisfiable. $\mathcal{I}$ contains all states that lose to a so any other state reachable at rank 1 in this subtree must lose to b. Therefore we can compute another interpolant that contains states that lose to b.

Algorithm 4 Amended tree formulas for Controller and Environment respectively

function treeFormula(gt)
  if rank$(gt) = 0$ **then**
    return $\neg B^M(x^{gt})$
  else
    return $\neg B^M(x^{gt}) \wedge \bigwedge_{n \in \text{succ}(gt)} (\delta(n) \wedge \text{label}(n) \wedge \text{treeFormula}(n))$
  end if
end function
function $\overline{\text{treeFormula}}$(gt)
  if rank$(gt) = 0$ **then**
    return $\text{E}(x^{gt})$
  else
    return $B^m[rank(gt)](x^{gt}) \wedge$
$(\text{E}(x^{gt}) \vee \bigvee_{n \in \text{succ}(gt)} (\delta(n) \wedge \text{label}(n) \wedge \overline{treeFormula}(n)))$
  end if
end function

This is the foundation for a recursive algorithm that consumes an entire tree by removing a single step on each iteration (Algorithm 5). All learned states from which the controller must lose are recorded in a set of bad states B^M. Environment-losing are learned in the same way but are recorded by removing the state from a set of sets of controller may-losing states B^m. An environment-losing state of rank n is removed from every set at rank n and lower.

3.3 Correctness

The correctness of the unbounded synthesis algorithm can be established independently from that of the bounded algorithm. Fortunately, the correctness of the bounded solver has been established in [16].

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 $B^m[i]$:

$$\forall i < k.\ B^m[i] \subseteq B^m[i+1], Upre(B^m[i]) \subseteq B^m[i+1].$$

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.

Algorithm 5 Learning algorithms

Require: $\sigma(X_T) \wedge \textsc{treeFormula}(T) \equiv \bot$
Require: *Must-invariant* holds
Ensure: *Must-invariant* holds
Ensure: $\sigma(X_T) \wedge B^M$
function $\textsc{learn}(\sigma, T)$
  if $T = \emptyset$ **then return**
  $n \leftarrow$ non-leaf node with max rank
  $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
  $\mathcal{I} \leftarrow \textsc{interpolate}(\sigma(X_T) \wedge \textsc{treeFormula}(T_1), \textsc{treeFormula}(T_2))$
  $B^M \leftarrow B^M \vee \mathcal{I}$
  $\textsc{learn}(\sigma, T_1)$
end function
Require: $\sigma(X_T) \wedge \overline{\textsc{treeFormula}}(T) \equiv \bot$
Require: *May-invariant* holds
Ensure: *May-invariant* holds
Ensure: $\sigma(X_T) \wedge B^m[\textsc{rank}(T)] \equiv \bot$
function $\overline{\textsc{learn}}(\sigma, T)$
  if $T = \emptyset$ **then return**
  $n \leftarrow$ non-leaf node with max rank
  $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
  $\mathcal{I} \leftarrow \textsc{interpolate}(\sigma(X_T) \wedge \overline{\textsc{treeFormula}}(T_1), \overline{\textsc{treeFormula}}(T_2))$
  for $i = 1$ to $\textsc{rank}(n)$ **do**
    $B^m[i] \leftarrow B^m[i] \wedge \neg\mathcal{I}$
  end for
  $\overline{\textsc{learn}}(\sigma, T_1)$
end function

3.4 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 $F_1 = \sigma(X_T) \wedge \overline{\text{treeFormula}}(T_1)$ and $F_2 = \overline{\text{treeFormula}}(T_2)$. These 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 $\mathcal{I} \wedge F_2 \equiv \bot$. 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, $\mathcal{I} \cap Upre(B^m[\text{rank}(n)]) = \emptyset$. Furthermore, since the may-invariant holds, $\mathcal{I} \cap Upre(i) = \emptyset$, 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 $\sigma(X_T) \wedge \overline{\text{treeFormula}}(T_1)$ also satisfies $\mathcal{I}$. Hence, removing $\mathcal{I}$ from $B^m[\text{rank}(n)]$ makes $\sigma(X_T) \wedge \overline{\text{treeFormula}}(T_1)$ 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 $\sigma(X_T) \wedge \overline{\text{treeFormula}}(T_1) \equiv \sigma(X_T) \wedge B^m[\text{rank}(T)](X^T) \equiv \bot$. Thus the second postcondition of $\overline{\text{LEARN}}$ holds.

3.5 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

If more states are removed from B^m on each iteration of checkRank the algorithm will converge and terminate sooner. Additionally, any states removed from B^M will reduce the number of states to be considered by the controller in future iterations.

As it is written, the algorithm only considers an overapproximations of states reachable from I for learning. Assuming that the algorithm does not terminate, then the addition of an extra step into the game on each iteration has the potential to greatly increase the reachable states. Considering some of these states earlier than they become reachable may lead to earlier termination.

Algorithm 6 Generalise I optimisation

```
function checkRank(s, k)
    r ← solveAbstract(env, ⟨s, k⟩, ∅)
    if r ≠ ∅ then return r
    s′ ← s
    for x ∈ s do
        r ← solveAbstract(env, ⟨s′ \ {x}, k⟩, ∅)
        if r = ∅ then s′ ← s′ \ {x}
    end for
    return ∅
end function
```

The optimisation that allows this is relatively simple and is inspired by a common greedy heuristic for minimising `unsat` cores. I is a value assignment to each variable in X. If the environment does not win for $\langle I, k\rangle$ then we attempt to solve for a generalised version of I by removing one assignment at a time. If the controller can win from the larger set of states then we continue generalising without it. In this way we learn more states by increasing the reachable set. In our benchmarks we have observed that this optimisation is beneficial on the first few iterations of checkRank.

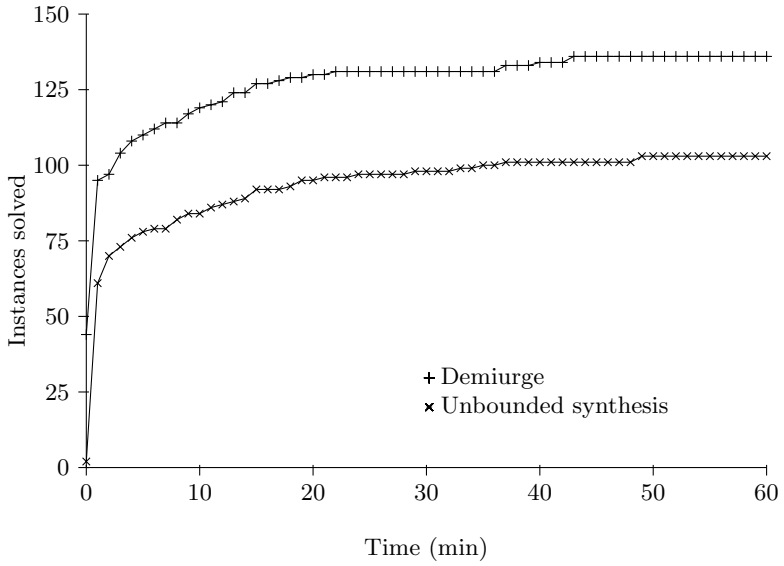
4.2 Choosing opponent moves

One aspect of the algorithm proposed by Narodytska et al. [16] is that it mimics real-world uses of synthesis by allowing cooperation between system and environment. Unfortunately, allowing the opponents to pick moves for one another can frequently backfire. It is common when modelling real-world systems to abstract over out of scope failures with transitions that immediately determine the outcome of the game. For example, in a network driver a request to send a packet may fail due to a disconnected wire, which might be modelled as an environment controlled transition to a system-winning state.

This slows down the synthesis process as all erroneous transition are accumulated into the abstract game tree. To mitigate this effect we have a heuristic to guess non-erroneous labels as *default moves* that are used instead of allowing the opponent to choose. This does not effect the correctness of the algorithm. If line (28) previously returned `sat` and is `unsat` with default moves then those moves would eventually have been added to the abstraction in a refinement step. If the formula is `unsat` even with the opponent choosing moves then it must be `unsat` with default moves.

5 Evaluation

We evaluate our approach on the benchmarks of the 2015 synthesis competition (SYNTCOMP'15). Each benchmark comprises of controllable and uncontrollable inputs to a circuit that assigns values to latches. One latch is configured as the error bit that determines the winner of the safety game. The benchmark suite is a collection of both real-world and toy specifications including generalised buffers, AMBA bus controllers, device drivers, and converted LTL formulas. Descriptions of many of the benchmark families used can be found in the 2014 competition report [12].



The implementation of our algorithm uses Glucose [1] for SAT solving and Periplo [18] for interpolant generation. We compared our implementation to two solvers that competed in the sequential realisability track of SYNTCOMP'15. Simple BDD Solver [19] because it had the best results in the track, and Demiurge [3] because it was the only SAT-based tool to compete.

The benchmarks were run on a cluster of Intel Quad Core Xeon E5405 CPUs with 16GB of memory. Each solver was allowed exclusive access to a node for one hour to solve an instance.

Our implementation was able to solver 103 out of the 250 specification in the alloted time, including 10 instances that were unable to be solved by any other solver including all other solvers in SYNTCOMP'15. Simple BDD Solver and Demiurge were able to solve X and Y respectively. Something about us vs BDDs and us vs demiurge with numbers of specs.

Four of the previously unsolved instances are device driver instances and another five are toy examples. This supports the hypothesis that different game solving methodologies perform better on certain classes of specifications.

6 Related Work

Synthesis of safety games is a thoroughly explored area of research with most efforts directed toward solving games with BDDs [6] and abstract interpretation [19, 5]. Satisfiability solving has been used previously for synthesis in a suite of methods proposed by Bloem et al [3]. The authors propose a propose employing competing SAT solvers to learn clauses, which is similar to our approach but does not unroll the game. They also suggest QBF solver, template-based, and Effectively Propositional Logic (EPL) approaches.

SAT-based bounded model checking approaches that unroll the transition relation have been extended to unbounded by using conflicts in the solver [13], or by interpolation [14]. However, there are no corresponding adaptations to synthesis. Incremental induction [4] is another technique for unbounded model checking with an equivalent synthesis method [15], which computes the set of safe states incrementally.

There are different approaches to bounded synthesis than the one described here. The authors of [11] suggest a methodology directly inspired by bounded model checking and it has been adapted to symbolic synthesis [9]. Lazy synthesis [10] is a counterexample-guided approach to bounded synthesis that refines an implementation for the game instead of an abstraction of it.

The algorithm presented in this paper solves realisability for safety games. There is a method for extracting strategies from abstract game trees that is compatible with our method [8]. It involves a similar interpolation approach for discovering states in game tree nodes.

7 Conclusion

References

1. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Theory and Applications of Satisfiability Testing, SAT. pp. 197–205 (2014)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 1579, pp. 193–207 (1999)

3. Bloem, R., Knighofer, R., Seidl, M.: Sat-based synthesis methods for safety specs. In: Verification, Model Checking, and Abstract Interpretation VMCAI. vol. 8318, pp. 1–20 (2014)
4. Bradley, A.R.: Sat-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 70–87 (2011)
5. Brenguier, R., Pérez, G.A., Raskin, J., Sankur, O.: Abssynthe: abstract synthesis from succinct safety specifications. In: Workshop on Synthesis, SYNT. pp. 100–116 (2014)
6. 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)
7. Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
8. Eèn, N., Legg, A., Narodytska, N., Ryzhyk, L.: Sat-based strategy extraction in reachability games. In: AAAI Conference on Artificial Intelligences, AAAI. pp. 3738–3745 (2015)
9. Ehlers, R.: Symbolic bounded synthesis. In: Computer Aided Verification, CAV. pp. 365–379 (2010)
10. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 219–234 (2012)
11. Finkbeiner, B., Schewe, S.: Bounded synthesis. Software Tools for Technology Transfer, STTT 15(5-6), 519–539 (2013)
12. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Könighofer, R., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The first reactive synthesis competition (SYNTCOMP 2014). Computing Research Repository, CoRR abs/1506.08726 (2015), http://arxiv.org/abs/1506.08726
13. McMillan, K.L.: Applying sat methods in unbounded symbolic model checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2404, pp. 250–264 (2002)
14. McMillan, K.L.: Interpolation and sat-based model checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2725, pp. 1–13 (2003)
15. Morgenstern, A., Gesell, M., Schneider, K.: Solving games using incremental induction. In: Integrated Formal Methods, IFM. pp. 177–191 (2013)
16. Narodytska, N., Legg, A., Bacchus, F., Ryzhyk, L., Walker, A.: Solving games without controllable predecessor. In: Computer Aided Verification. pp. 533–540 (2014)
17. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
18. Rollini, S., L., A., G., F., A., H., N., S.: PeRIPLO: A framework for producing efficient interpolants for sat-based software verification. In: Logic for Programming Artificial Intelligence and Reasoning, LPAR. pp. 683–693 (2013)
19. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: Formal Methods in Computer-Aided Design FMCAD. pp. 219–226 (2014)