A SAT-Based Counterexample Guided Method for Unbounded Synthesis

Alexander Legg[1], Nina Narodytska[2], and Leonid Ryzhyk[2]

[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. 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.

The reactive synthesis problem is EXPTIME-complete so naive algorithms are infeasible on anything but simple systems. There are several techniques that aim to mitigate this complexity by representing states and transitions of the system symbolically [18, 4, 16]. These techniques incrementally construct a symbolic representation of the set of discovered winning states of the game. The downside of this approach is that keeping track of all discovered winning states can lead to a space explosion even when using efficient symbolic representation such as BDDs or CNFs.

* 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.

† Work completed at Carnegie Mellon University

An alternative approach, proposed by Narodytska et al. [17], is to eschew states and focus on *runs* of the game. The 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 other existing synthesis methods, it does not store, in either symbolic or explicit form, the set of winning states. Instead, it uses counterexample guided backtracking search to identify a small subset of runs that are sufficient to solve the game.

This method has been shown to outperform BDDs on certain classes of games; however it suffers from an important limitation: it is only able to solve games with a bounded number of rounds. In case of safety games, this means proving that the controller can keep the game within the safe region for a bounded number of steps. This is insufficient in most practical situations that require unbounded realisability.

In this paper, we extend the method by Narodytska et al. [17] to unbounded safety games. To this end we enhance the method with computational learning: every time the search algorithm discovers a new counterexample, the learning procedure analyzes the counterexample, extracting a subset of states winning for one of the players from it. The learning procedure ensures that reaching a fixed point in these sets is sufficient to establish unbounded realisability. Our method can be seen as a hybrid between the counterexample guided algorithm by Narodytska et al. and methods based on losing set computation: we use the former to guide the search, while we rely on the latter to ensure convergence.

We evaluate our method on the benchmarks of the 2015 synthesis competition (SYNTCOMP'15). While our solver solves fewer total instances than competitors, it solves the largest number of unique instances, i.e., instances that could not be solved by any other solver. These results confirm that there exist classes of problems that are hard for traditional synthesis techniques, but can be efficiently solved by our method. While further performance improvements are clearly needed, in its current state the method can be used in a portfolio solver to increase the number of tractable instances.

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 evaluate our methodology, and compare our approach to other synthesis techniques.

2 Background

A *safety game*, $G = \langle X, U, C, \delta, I, E \rangle$, is defined over boolean state variables X, uncontrollable action variables U, and controllable action variables 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 $\delta(X, U, C, X')$ of the game is a boolean formula that relates current state and action 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 action, the *controller* responds by choosing a controllable action 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 action 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 maximum bound κ all runs are restricted to length κ, 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 U and C.

A *controller strategy* $\pi^c : (2^X, 2^U) \to 2^C$ is a mapping of states and uncontrollable inputs to controllable actions. A controller strategy is winning in a bounded game of maximum bound κ 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 *realisability* is the problem of determining the existence of such a strategy for a bounded game.

An *environment strategy* $\pi^e : 2^X \to 2^U$ is a mapping of states to uncontrollable actions. 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 $(x_0, \pi^e(x_0), c_0), (x_1, \pi^e(x_1), c_1) \dots (x_n, \pi^e(x_n), c_n)$ 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. [17], which is the main building block for our unbounded algorithm.

Example. *We introduce a running example to assist the explanation. We consider a simple arbiter system in which the environment requests for either 1 or 2 resources, and the controller may grant access to two resources. The total number of requests grows each round by the number of environment requests and shrinks by the number of resources granted by the controller in the previous round. The controller must ensure that the number of unhandled requests does not accumulate to more than 2. Figure 1 shows the variables (1a) and the formulas for computing next-state variables (1c) for this example. We use primed identifiers to denote next-state variables and curly braces to define the domain of a variable.*

This example is the $n = 2$ instance of the more general problem of an arbiter of n resources. The set of winning states grows exponentially with the number of resources the controller could use to grant requests, which makes the problem hard for BDD solvers. In Section 3 we will outline how this example can be solved without enumerating all winning states.

The bounded synthesis algorithm consists of two competing solvers, one for the controller and one for the environment, that solve abstractions of the safety game. Each solver builds a candidate strategy for its corresponding player consisting of assignments to 2^C and 2^U respectively.

Controllable	Uncontrollable	State
request : {1, 2}	grant0 = {0, 1}	resource0 = {0, 1}
	grant1 : {0, 1}	resource1 = {0, 1}
		nrequests : {0, 1, 2, 3}

(a) Variables

```
resource0 = 0; resource1 = 0; nrequests = 0;
```

(b) Initial State

```
resource0' = grant0;
resource1' = grant1;
nrequests' = (nrequests + request >= resource0 + resource1)
           ? (nrequests + request - resource0 - resource1) : 0;
```

(c) Transition Relation

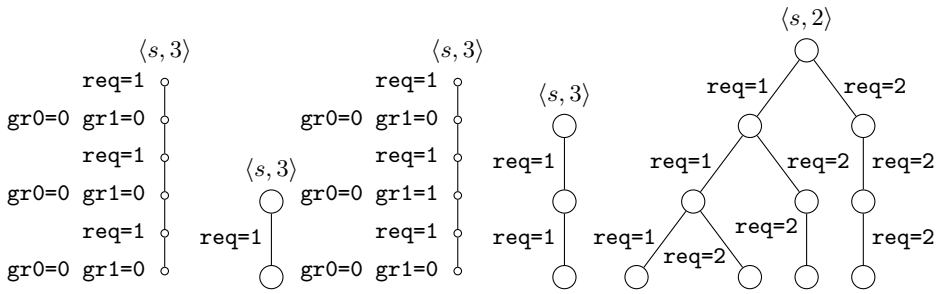
Fig. 1: Example

The game abstraction is initially constructed from an opponent candidate strategy. It is refined by counterexamples to the solver's own candidate strategy that are formed when the opponent solves the abstract game corresponding to that strategy. By abstracting the game in this way we conceptually direct the solvers' search to consider opponent actions that may lead to winning strategies.

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* (AGTs). Figure 2b shows an example abstract game tree restricting the environment (abstract game trees restricting the controller are similar). In the abstract game, the controller can freely choose actions whilst the environment is required to pick actions from the tree. After reaching a leaf, the enivironment continues playing unrestricted. The tree in Figure 2b restricts the first environment action to `request=1`. At the leaf of the tree the game continues 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, $\textsc{leaves}(T)$ the subset of leaf nodes. For edge e, $\textsc{action}(e)$ is the action that labels the edge, and for node n, $\textsc{height}(k, n)$ is the distance from n to the last round of a game bounded to k rounds. $\textsc{height}(k, T)$ is the height of the root node of the tree. For node n of the tree, $\textsc{succ}(n)$ is the set of pairs $\langle e, n' \rangle$ where n' is a child node of n and e is the edge connecting n and n'.

Given an environment (controller) abstract game tree T a *partial strategy* $Strat : \textsc{nodes}(T) \to C$ ($Strat : \textsc{nodes}(T) \to U$) labels each node of the tree with the controller's (environment's) action to be played in that node. Given a partial strategy $Strat$, we can map each leaf l of the abstract game tree to $\langle s', i' \rangle = \textsc{outcome}(\langle s, i \rangle, Strat, l)$ obtained by playing all controllable and uncontrollable actions on the path from the root to the leaf.



(a) Environment-winning trace

(b) AGT

(c) Controller-winning trace

(d) AGT

(e) Refined AGT

Fig. 2: Abstract game trees.

Example. *To solve our example, the algorithm begins by invoking a SAT solver to find a trace of length k (here we consider $k = 3$) from the initial state to the error state (see Figure 2a). This trace is used to create an abstract game tree by taking the first action in the trace and fixing it as the first action in the game tree (Figure 2b). This abstract game tree represents the conjecture that* `req=1` *may be a good first action. Next the solver attempts to find a counterexample to this conjecture by finding a trace that avoids the error state with the first environment action fixed to* `req=1` *(Figure 2c), i.e. by playing the abstract game on behalf of the controller.*

Observe that the controller chooses to grant only one resource, just enough to ensure that an error state will not be reached. This example will show the worst case when the SAT solver returns actions that will lead to large abstract game trees. However, in practice we employ heuristics to guide the SAT solver away from the worst case.

The environment's candidate is refined using the controller's counterexample traces. In Figure 2d, the game tree tree is refined based on the first action of the controller's trace. The `req=1` *action in the second round is still winning under the assumption that the controller will not grant any resources. The environment is free to make that assumption because the controller's counterexample does not yet contain any grants. Eventually, the refinements to the environment's candidate will force the controller to grant more resources and result in a game tree such as in Figure 2e.*

The bounded synthesis algorithm solves abstract game trees by invoking a SAT solver to search for candidate partial strategies. Candidate strategies are used as game abstractions for the opponent and can be verified by a call to the opponent solver. The solver recursively refines the abstraction by finding counterexample strategies. 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 = NULL then return NULL ▷ 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 T′ with counterexample
10:        cand ← solveAbstract(p, s, k, T′)              ▷ Solve refined game tree
11:    end loop
12: end function
13: function findCandidate(p, s, k, T)
14:    T̂ ← extend(T)                          ▷ Extend the tree with unfixed actions
15:    f ← if p = cont then treeFormula(k, T̂) else ¬treeFormula(k, T̂)
16:    sol ← SAT(s(X_T̂) ∧ f)
17:    if sol = unsat then
18:        if unbounded then                  ▷ Active only in the unbounded solver
19:            if p = cont then learn(s, T̂) else ¬learn(s, T̂)
20:        end if
21:        return NULL                                      ▷ No candidate exists
22:    else
23:        return {⟨n, c⟩ | n ∈ nodes(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 bound and state at leaf
29:        u ← solveAbstract(opponent(p), s′, k′, ∅)      ▷ Solve for the opponent
30:        if u ≠ NULL then return ⟨true, l, u⟩           ▷ Return counterexample
31:    end for
32:    return ⟨false, ∅, ∅⟩
33: end function
```

The algorithm takes a concrete game G with maximum bound κ 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 p, if one exists. The initial invocation of the algorithm takes the initial state I, bound κ 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 game round. 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 that is winning when the opponent is restricted to T. If it fails to find a strategy, this

means that no winning strategy exists against the opponent playing its partial strategy represented by T. If, on the other hand, a candidate partial strategy is found, we need to verify if it is indeed winning for the abstract game 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).

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.

The CEGAR loop depends on the ability to guess candidate partial strategies in FINDCANDIDATE. For this purpose we use the heuristic that a partial strategy may be winning if each OUTCOME of the strategy can be extended to a run of the game that is winning for the current player. Clearly, if such a partial strategy does not exist then no winning partial strategy can exist for the abstract game tree. We can formulate this heuristic as a SAT query, which is constructed recursively by TREEFORMULA (for the controller) or $\overline{\text{TREEFORMULA}}$ (for the environment) in Algorithm 2.

The tree is first extended to the maximum bound with edges that are labeled with arbitrary opponent actions (Algorithm 1, line 14). For each node in the tree, new SAT variables are introduced corresponding to the state (X_T) and action (U_T or C_T) variables of that node. Additional variables for the opponent actions in the edges of T are introduced (U_e or C_e) and set to ACTION(e). The state and action variables of node n are connected to successor nodes SUCC(n) by an encoding of the transition relation and constrained to the winning condition of the player.

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 maximum bound.

It is possible to set a maximum 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 [3], which is the smallest

Algorithm 2 Tree formulas for Controller and Environment respectively

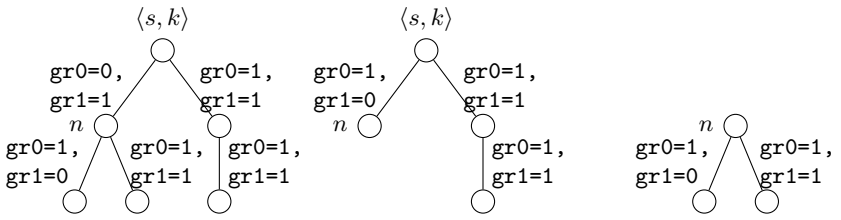
1: **function** TREEFORMULA(k, T)
2: **if** HEIGHT$(k, T) = 0$ **then**
3: **return** $\neg \mathrm{E}(X_T)$
4: **else**
5: **return** $\neg \mathrm{E}(X_T) \wedge$
6: $$\bigwedge_{\langle e,n\rangle \in \mathrm{SUCC}(T)} (\delta(X_T, U_e, C_T, X_n) \wedge U_e = \mathrm{ACTION}(e) \wedge \mathrm{TREEFORMULA}(k, n))$$
7: **end if**
8: **end function**
9: **function** $\overline{\mathrm{TREEFORMULA}}(k, T)$
10: **if** HEIGHT$(k, T) = 0$ **then**
11: **return** $\mathrm{E}(X_T)$
12: **else**
13: **return** $\mathrm{E}(X_T) \vee$
14: $$\bigvee_{\langle e,n\rangle \in \mathrm{SUCC}(T)} (\delta(X_T, U_T, C_e, X_n) \wedge C_e = \mathrm{ACTION}(e) \wedge \overline{\mathrm{TREEFORMULA}}(k, n))$$
15: **end if**
16: **end function**

number d such that for any state x there is a path of length $\leq d$ to all other reachable states. However, the diameter is difficult to estimate and can still lead to infeasibly long games.

We instead present an approach that iteratively solves games of increasing bound while learning winning states from abstract games using interpolation. We show that reaching a fixed point in learned states is sufficient for completeness.

3.1 Learning States with Interpolants

We extend the bounded synthesis algorithm to learn states losing for one of the players from failed attempts to find candidate strategies. The learning procedure kicks in whenever FINDCANDIDATE cannot find a candidate strategy for an abstract game tree T with initial state s and bound k. For the controller, this means the environment partial strategy represented by T will always reach E from s no matter the assignments to C variables (the bound is irrelevant). For the environment, the controller can avoid E for k rounds no matter the assignments to U variables. Hence, we have established that s is a losing state either always (for the controller) or for bound k (for the environment. We can learn additional losing states from the tree via interpolation. This is achieved in lines 18–20 in Algorithm 1, enabled in the unbounded version of the algorithm, which invoke LEARN or $\overline{\mathrm{LEARN}}$ to learn controller or environment losing states respectively (Algorithm 3).



(a) A losing AGT T (b) Tree slice T_1 (c) Tree slice T_2

Fig. 3: Splitting of an abstract game tree by the learning procedure.

Example. *Consider the game tree T in Figure 3a from our running example. This tree is labelled with controller actions that prevent the environment from forcing the game into the error state. No matter how many requests the environment makes, the controller will grant a surplus. So the* findCandidate *function can not find a candidate strategy for the environment. We now learn the states for which these controller actions can force the game to stay in the controller winning set.*

We choose a non-leaf node n of T with maximal depth, i.e., a node whose children are leafs (Algorithm 3, line 3). We then split the tree at n such that both slices T_1 and T_2 contain a copy of n (line 4). Figure 3b shows T_1, which contains all of T except n's children, and T_2 (Figure 3c), which contains only n and its children. There is no candidate strategy for T so $s \wedge \overline{\text{treeFormula}}(k, T)$ is unsatisfiable. By construction, $\overline{\text{treeFormula}}(k, T) \equiv \overline{\text{treeFormula}}(k, T_1) \wedge \overline{\text{treeFormula}}(k, T_2)$ and hence $s \wedge \overline{\text{treeFormula}}(k, T_1) \wedge \overline{\text{treeFormula}}(k, T_2)$ is also unsatisfiable. This enables the construction of an *interpolant* that captures losing states at node n.

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 [8] $\mathcal{I}$ 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 $F_1 \wedge F_2$ [19].

We construct an interpolant with $F_1 = s(X_T) \wedge \text{treeFormula}(k, T_1)$ and $F_2 = \text{treeFormula}(k, T_2)$ (line 5). The only variables shared between F_1 and F_2 are the state variable copies belonging to node n. By the properties of the interpolant, $F_2 \wedge \mathcal{I}$ is unsatisfiable, therefore all states in $\mathcal{I}$ are losing against abstract game tree T_2 in Figure 3c. We also know that $F_1 \rightarrow \mathcal{I}$, thus $\mathcal{I}$ contains all states reachable at n by following T_1 and avoiding error states.

Example. *In the pictured trees the interpolant* $\texttt{nrequests} = 0$ *is reachable at n and no move the environment chooses will reach an error state in one round from this set. A BDD solver using an uncontrollable predecessor operation on the error set would be forced to find a much more complicated set:* $\texttt{nrequests} = 3 \vee (\texttt{nrequests} = 2 \wedge (\texttt{resource0} = 0 \vee \texttt{resource1} = 0)) \vee (\texttt{nrequests} = 1 \wedge (\texttt{resource0} = 0 \wedge \texttt{resource1} = 0))$*. This set gets more complicate with*

Algorithm 3 Learning algorithms

Require: $s(X_T) \wedge \textsc{treeFormula}(k, T) \equiv \bot$
Require: *Must-invariant* holds
Ensure: *Must-invariant* holds
Ensure: $s(X_T) \wedge B^M \not\equiv \bot$ ▷ s will be added to B^M
1: **function** $\textsc{learn}(s, T)$
2: **if** $\textsc{succ}(T) = \emptyset$ **then return**
3: $n \leftarrow$ non-leaf node with min height
4: $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
5: $\mathcal{I} \leftarrow \textsc{interpolate}(s(X_T) \wedge \textsc{treeFormula}(k, T_1), \textsc{treeFormula}(k, T_2))$
6: $B^M \leftarrow B^M \vee \mathcal{I}$
7: $\textsc{learn}(s, T_1)$
8: **end function**
Require: $s(X_T) \wedge \overline{\textsc{treeFormula}}(k, T) \equiv \bot$
Require: *May-invariant* holds
Ensure: *May-invariant* holds
Ensure: $s(X_T) \wedge B^m[\textsc{height}(k, T)] \equiv \bot$ ▷ s will be removed from B^m
9: **function** $\overline{\textsc{learn}}(s, T)$
10: **if** $\textsc{succ}(T) = \emptyset$ **then return**
11: $n \leftarrow$ non-leaf node with min height
12: $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
13: $\mathcal{I} \leftarrow \textsc{interpolate}(s(X_T) \wedge \overline{\textsc{treeFormula}}(k, T_1), \overline{\textsc{treeFormula}}(k, T_2))$
14: **for** $i = 1$ to $\textsc{height}(k, n)$ **do**
15: $B^m[i] \leftarrow B^m[i] \setminus \mathcal{I}$
16: **end for**
17: $\overline{\textsc{learn}}(s, T_1)$
18: **end function**

more resources in the system. The advantage of interpolation is that we may overapproximate the losing states.

We have discovered a set $\mathcal{I}$ of states losing for the environment. Environment-losing states are only losing for a particular bound: given that there does not exist an environment strategy that forces the game into an error state in k rounds or less; there may still exist a longer environment-winning strategy. We therefore record learned environment-losing states along with associated bounds. To this end, we maintain a conceptually infinite array of sets $B^m[k]$ that are may-losing (*bad*) for the controller, indexed by bound k. $B^m[k]$ are initialised to $\top$ for all k. Whenever an environment-losing set $\mathcal{I}$ is discovered for a node n with bound $\textsc{height}(k, n)$ in line 13 of Algorithm 3, this set is subtracted from $B^m[i]$, for all i less than or equal to the bound (lines 14–16).

The $\overline{\textsc{treeFormula}}$ function is modified for the unbounded solver (Algorithm 4) to constrain the environment to the appropriate B^m. This enables further interpolants to be constructed by the learning procedure recursively splitting more nodes from T_1 (Algorithm 3, line 7) since the states that are losing to T_2 are no longer contained in B^m.

Algorithm 4 Amended tree formulas for Controller and Environment

1: **function** treeFormula(k, T)
2: **if** height$(k, T) = 0$ **then**
3: **return** $\neg B^M(X_T)$
4: **else**
5: **return** $\neg B^M(X_T) \wedge$
6:
$$\bigwedge_{\langle e,n\rangle \in \text{succ}(T)} (\delta(X_T, U_e, C_n, X_n) \wedge U_e = \text{action}(e) \wedge \text{treeFormula}(k, n))$$
7: **end if**
8: **end function**
9: **function** $\overline{\text{treeFormula}}(k, T)$
10: **if** height$(k, T) = 0$ **then**
11: **return** $\mathrm{E}(X_T)$
12: **else**
13: **return** $B^m[\text{height}(k, T)](X_T) \wedge$
14:
$$\left(\mathrm{E}(X_T) \vee \bigvee_{\langle e,n\rangle \in \text{succ}(T)} (\delta(X_T, U_n, C_e, X_n) \wedge C_e = \text{action}(e) \wedge \overline{\text{treeFormula}}(k, n))\right)$$
15: **end if**
16: **end function**

Learning of states losing from the controller is similar (learn in Algorithm 3). The main difference is that environment-losing states are losing for all bounds. Therefore we record these states in a single set B^M of must-losing states (Algorithm 3, line 6). This set is initialised to the error set E and grows as new losing states are discovered. The modified $\overline{\text{treeFormula}}$ function (Algorithm 4) blocks must-losing states, which also allows for recursive learning over the entire tree.

3.2 Main synthesis loop

Figure 5 shows the main loop of the unbounded synthesis algorithm. The algorithm invokes the modified bounded synthesis procedure with increasing bound k until the initial state is in B^M (environment wins) or B^m reaches a fixed point (controller wins). We prove correctness in the next section.

3.3 Correctness

We define two global invariants of the algorithm. The *may-invariant* states that sets $B^m[i]$ grow monotonically with i and that each $B^m[i+1]$ overapproximates the states from which the environment can force the game into $B^m[i]$. We call this operation $Upre$, the uncontrollable predecessor. So the *may-invariant* is:

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

Algorithm 5 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 < k. B^m[i] ≡ B^m[i + 1] then        ▷ Reached fixed point
 7:             return realisable
 8:         B^m[k] ← E
 9:         checkBound(k)
10:     end for
11: end function
Require: May and must invariants hold
Ensure: May and must invariants hold
Ensure: I ∉ B^m[k] if there exists a winning controller strategy with bound k
Ensure: I ∈ B^M if there exists a winning environment strategy with bound k
12: function checkBound(k)
13:     return solveAbstract(env, I, k, ∅)
14: end function
```

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

$$B^M \subseteq B.$$

Both invariants trivially hold after B^m and B^M have been initialised in the beginning of the algorithm. The sets B^m and B^M are only modified by the functions LEARN and $\overline{\text{LEARN}}$. Below we prove that $\overline{\text{LEARN}}$ maintains the invariants. The proof of LEARN is similar.

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

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

Line (11–12) splits the tree T into T_1 and T_2, such that T_2 has depth 1. Consider formulas $F_1 = s(X_T) \wedge \overline{\text{treeFormula}}(k, T_1)$ and $F_2 = \overline{\text{treeFormula}}(k, T_2)$. 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 $s(X_T) \wedge \overline{\text{treeFormula}}(k, T)$, 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 s 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 $B^m[\text{height}(k, n) - 1]$ playing

against the counterexample moves in T_2. Hence, $\mathcal{I} \cap Upre(B^m[\text{HEIGHT}(k,n) - 1]) = \emptyset$. Furthermore, since the may-invariant holds, $\mathcal{I} \cap Upre(B^m[i]) = \emptyset$, for all $i < \text{HEIGHT}(k,n)$. Hence, removing $\mathcal{I}$ from all $B^m[i], i \leq \text{HEIGHT}(k,n)$ 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 $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k,T_1)$ also satisfies $\mathcal{I}$. Hence, removing $\mathcal{I}$ from $B^m[\text{HEIGHT}(k,n)]$ makes $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k,T_1)$ 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, $\overline{\text{TREEFORMULA}}(k,T_1) \equiv B^m[\text{HEIGHT}(k,T_1)](X^T)$; hence $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k,T_1) \equiv s(X_T) \wedge B^m[\text{HEIGHT}(k,T_1)](X^T) \equiv \bot$. Thus the second postcondition of $\overline{\text{LEARN}}$ holds.

The proof of LEARN is similar to the above proof of LEARN. An interpolant constructed from $F_1 = s(X_T) \wedge \text{TREEFORMULA}(k,T_1)$ and $F_2 = \text{TREEFORMULA}(k,T_2)$ has the property $\mathcal{I} \wedge F_2 \equiv \bot$ and the precondition ensures that the controller is unable to force the game into B^M playing against the counterexample moves in T_2. Thus adding $\mathcal{I}$ to B^M maintains the must-invariant satisfying the first postcondition.

Likewise, in the second last recursive call of LEARN with the empty tree T_1 and root node n: $\text{TREEFORMULA}(k,T_1) \equiv \neg B^M(X_T)$. Hence $s(X_T) \wedge \text{TREEFORMULA}(k,T_1) \equiv s(X_T) \wedge \neg B^M(X_T) \equiv \bot$. Therefore $s(X_T) \wedge B^M(X_T) \not\equiv \bot$, the second postcondition, is true.

3.5 Proof of Termination

We must prove that CHECKBOUND terminates and that upon termination its postcondition holds, i.e., state I is removed from $B^m[\kappa]$ if there is a winning controller strategy on the bounded safety game of maximum bound κ 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 bound κ. This means that at some point the algorithm discovers a counterexample tree of bound κ for which the environment cannot force into E. The algorithm then invokes the $\overline{\text{LEARN}}$ method, which removes I from $B^m[\kappa]$. Alternatively, if there is a winning strategy for the environment at bound κ then a counterexample losing for the controller will be found. Subsequently LEARN will be called and I added to B^M.

3.6 Optimisation: Generalising the initial state

This optimisation allows us to learn may and must losing states faster. Starting with a larger set of initial states we increase the reachable set and hence increase the number of states learned by interpolation. This optimisation requires a modification to SOLVEABSTRACT to handle sets of states, which is not shown.

The optimisation is relatively simple and is inspired by a common greedy heuristic for minimising `unsat` cores. Initial state I assigns a value to each variable in X. If the environment loses $\langle I, k\rangle$ then we attempt to solve for a generalised version of I by removing one variable assignment at a time. If the environment loses from the larger set of states then we continue generalising. 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 checkBound.

Algorithm 6 Generalise I optimisation

```
function checkBound(k)
    r ← solveAbstract(env, I, k, ∅)
    if r ≠ ∅ then return r
    s' ← I
    for x ∈ X do
        r ← solveAbstract(env, s' \ {x}, k, ∅)
        if r = NULL then s' ← s' \ {x}        ▷ Removes the assignment to x from s'
    end for
    return NULL
end function
```

4 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 [13].

The implementation of our algorithm uses Glucose [2] for SAT solving and Periplo [20] for interpolant generation. The benchmarks were run on a cluster of Intel Quad Core Xeon E5405 2GHz CPUs with 16GB of memory. The solver was allowed exclusive access to a node for one hour to solve an instance.

The results of our benchmarking are shown, along with the synthesis competition results [1], in Table 1. The competition was run on Intel Quad Core 3.2Ghz CPUs with 32GB of memory, also on isolated nodes for one hour per instance.

Our implementation was able to solve 103 out of the 250 specification in the alloted time, including 12 instances that were not solved by any other solver in SYNTCOMP'15. The unique instances we solved are listed in Table 2.

Solver	**Solved**	**Unique**
Simple BDD Solver (2)	195	10
AbsSynthe (seq2)	187	2
Simple BDD Solver (1)	185	
AbsSynthe (seq3)	179	
Realizer (sequential)	179	
AbsSynthe (seq1)	173	1
Demiurge (D1real)	139	5
Aisy	98	
Unbounded Synthesis	*103*	*12*

Table 1: Synthesis Competition 2015 Results

1. `6s216rb0_c0to31`	7. `driver_c10n`
2. `cnt30y`	8. `stay18y`
3. `driver_a10n`	9. `stay20n`
4. `driver_a8n`	10. `stay20y`
5. `driver_b10y`	11. `stay22n`
6. `driver_b8y`	12. `stay22y`

Table 2: Instances uniquely solved by our approach

Five of the instances unique to our solver are device driver instances and another five are from the `stay` family. This supports the hypothesis that different game solving methodologies perform better on certain classes of specifications.

We also present a cactus plot of the number of instances solved over time (Figure 4). For reference we plotted our results along with Demiurge [4], the only SAT-based tool in the competition, and the winner of the sequential realisability track Simple BDD Solver 2 [21]. We were able to run Demiurge on our hardware but show competition results for Simple BDD Solver 2.

Whilst our solver is unable to solve as many instances as other tools, it was able to solve more unique instances than any solver in the competition. This confirms that our methodology is able to fill gaps in a state of the art synthesis toolbox by more efficiently solving instances that are hard for other techniques. For this reason our solver would be a worthwhile addition to a portfolio solver. In the parallel track of the competition, Demiurge uses a suite of 3 separate but communicating solvers. The solvers relay unsafe states to one another, which is compatible with the set B^M in our solver. It remains future work to explore the possibility of combining these solvers.

5 Related Work

Synthesis of safety games is a thoroughly explored area of research with most efforts directed toward solving games with BDDs [7] and abstract interpretation [21, 6]. Satisfiability solving has been used previously for synthesis in a suite of methods proposed by Bloem et al. [4]. The authors propose employing competing

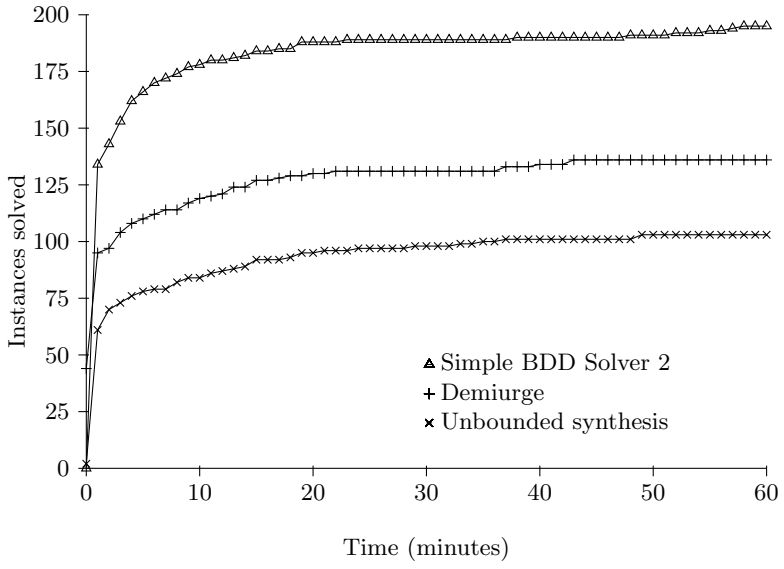


Fig. 4: Number of instances solver over time by the unbounded solver, Demiurge, and Simple BDD Solver.

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 (EPR) approaches.

SAT-based bounded model checking approaches that unroll the transition relation have been extended to unbounded by using conflicts in the solver [14], or by interpolation [15]. However, there are no corresponding adaptations to synthesis.

Incremental induction [5] is another technique for unbounded model checking with an equivalent synthesis method [16], which computes sets of states that overapproximate the losing states (similar to our B^m) and another set of winning states (similar to the negation of B^M). Their algorithm maintains a similar invariant over the sets of losing states as our approach and has the same termination condition. It differs in how the sets are computed, which it does by inductively proving the number of game rounds required by the environment to win from the initial state.

There are different approaches to bounded synthesis than the one described here. The authors of [12] suggest a methodology directly inspired by bounded model checking and it has been adapted to symbolic synthesis [10]. Lazy synthesis [11] is a counterexample guided approach to bounded synthesis that refines an implementation for the game instead of an abstraction of it.

The original bounded synthesis algorithm of Narodytska et al. [17] solves realisability without constructing a strategy. In [9] the realisability algorithm is extended with strategy extraction. The technique relies on interpolation over

abstract game trees to compute the winning strategy. In the present work we use interpolation in a different way in order to learn losing states of the game.

6 Conclusion

We presented an extension to an existing bounded synthesis technique that promotes it to unbounded safety games. The approach taken as whole differs from other synthesis techniques by combining counterexample guided game solving with winning set computation. Intuitively, the abstraction refinement framework of the bounded synthesis algorithm restricts the search to consider only moves that may lead to winning strategies. By constructing sets of bad states during this search we aim to consider only states that are relevant to a winning strategy while solving the unbounded game.

The results show that our approach is able to solve more unique instances than other solvers by performing well on certain classes of games that are hard for other methodologies.

Future work includes incorporating the solver into a parallel suite of communicating solvers [4]. There is evidence that different solvers perform well on different classes of games. Thus we hypothesise that the way forward for synthesis tools is to combine the efforts of many different techniques.

References

1. Syntcomp 2015 results. http://syntcomp.cs.uni-saarland.de/syntcomp2015/experiments/, accessed: 2016-01-29
2. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Theory and Applications of Satisfiability Testing, SAT. pp. 197–205 (2014)
3. 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)
4. 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)
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 70–87 (2011)
6. 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)
7. 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)
8. Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
9. 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)

10. Ehlers, R.: Symbolic bounded synthesis. In: Computer Aided Verification, CAV. pp. 365–379 (2010)
11. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 219–234 (2012)
12. Finkbeiner, B., Schewe, S.: Bounded synthesis. Software Tools for Technology Transfer, STTT 15(5-6), 519–539 (2013)
13. 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
14. 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)
15. McMillan, K.L.: Interpolation and SAT-based model checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2725, pp. 1–13 (2003)
16. Morgenstern, A., Gesell, M., Schneider, K.: Solving games using incremental induction. In: Integrated Formal Methods, IFM. pp. 177–191 (2013)
17. Narodytska, N., Legg, A., Bacchus, F., Ryzhyk, L., Walker, A.: Solving games without controllable predecessor. In: Computer Aided Verification. pp. 533–540 (2014)
18. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) designs. In: Verification, Model Checking, and Abstract Interpretation VMCAI. pp. 364–380 (2006)
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
20. 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)
21. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: Formal Methods in Computer-Aided Design FMCAD. pp. 219–226 (2014)