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 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.

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 [2]. Another approach is to eschew states and focus on *runs* of the game. Previous work has applied this idea to realizability of bounded games [14] 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 $\delta : (X, L_u, L_c) \to X$, 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* $\pi^c : (X, L_u) \to L_c$ 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 $x_0 = I \wedge \exists i \in \{1..n\}(x_i = E)$ 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 $2n$ (a step is two moves - environment then controller) and width 1 populated by unfixed edges. Fixing some moves and thereby reducing the set of runs forms an *abstract game tree* that symbolically represents a smaller version of the game.

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[14]. 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 (*respectively*: in all branches) for the controller (*resp*: environment.) The functions that produce these formulas are shown in Algorithm 1.

Algorithm 1 Tree formulas for Controller and Environment respectively

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

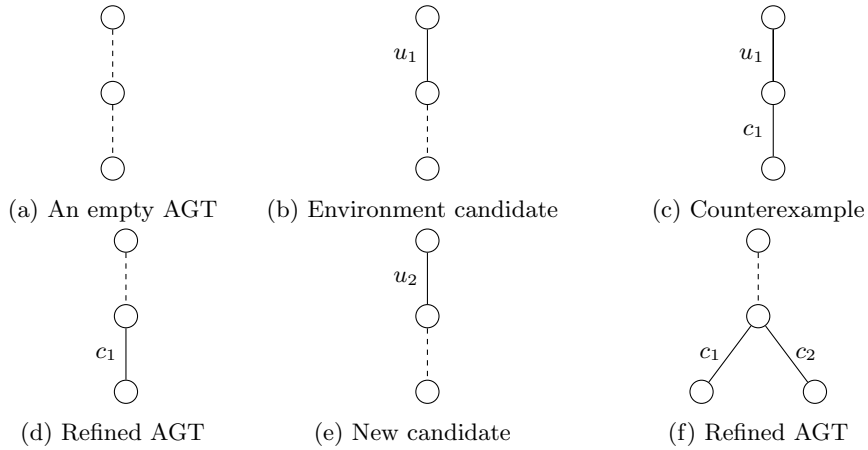
When we produce a formula using treeFormula the SAT solver is playing on behalf on the controller. Any unfixed labels in the tree, controllable or uncontrollable, will be existentially quantified. This means that if there exists a way for the game to be solved when both players cooperate on unfixed labels the SAT solver will find it. 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.

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 environment-winning run is exists within the partial strategy then it is a winning partial strategy, otherwise the spoiling run is a counterexample that proves that the partial strategy is not winning. This forms the basis of a counterexample guided abstraction refinement framework [6] that operates on candidate strategies.

This technique was proposed by Narodytska et al [14]. In Section 3 we propose an extension to synthesis of unbounded games but the original algorithm is outlined here for completeness.

The bounded synthesis algorithm (Algorithm 2) 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 procedure is listed in Algorithm 2.



(a) An empty AGT (b) Environment candidate (c) Counterexample

(d) Refined AGT (e) New candidate (f) Refined AGT

Fig. 1: Abstract game trees

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 [1], 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.

Algorithm 2 Bounded Synthesis

function SOLVEABSTRACT($p, \langle s, k\rangle, T$)
  $cand \leftarrow$ FINDCANDIDATE($p, \langle s, k\rangle, T$) ▷ Look for a candidate
  if $k = n - 1$ **then return** $cand$ ▷ Reached the bound
  $T' \leftarrow T$
  loop
    if $cand = \emptyset$ **then return** $\emptyset$ ▷ No candidate: return with no solution
    $\langle cex, l, u\rangle \leftarrow$ VERIFY($p, \langle s, k\rangle, T, cand$) ▷ Verify candidate
    if $cex =$ `false` **then return** $cand$ ▷ No counterexample: return candidate
    $T' \leftarrow$ GTAPPEND(T', l, u) ▷ Refine AGT with counterexample
    $cand \leftarrow$ SOLVEABSTRACT($p, \langle s, k\rangle, T'$) ▷ Solve refined AGT
  end loop
end function
function FINDCANDIDATE($p, \langle s, k\rangle, T$)
  $f \leftarrow$ **if** $p =$ `cont` **then** TREEFORMULA(T) **else** $\overline{\text{TREEFORMULA}}(T)$
  $sol \leftarrow \text{SAT}(s(X_T) \wedge f)$
  if $sol =$ `unsat` **then**
    if `unbounded` **then** ▷ Active only in the unbounded solver
      $\sigma \leftarrow$ GENERALISE(s) ▷ Expand s to a set of states
      if $p =$ `cont` **then** LEARN(σ, T) **else** $\overline{\text{LEARN}}(\sigma, T)$
    end if
    return $\emptyset$ ▷ No candidate exists
  else
    return $\{\langle n, c\rangle | n \in$ GTNODES(T) $, c =$ SOL(n)$\}$ ▷ Fix candidate moves in T
  end if
end function
function VERIFY($p, \langle s, k\rangle, gt, cand$)
  for $l \in leaves(gt)$ **do**
    $\langle k', s'\rangle \leftarrow$ OUTCOME($\langle s, k\rangle, gt, l$) ▷ Get rank and state at leaf
    $u \leftarrow$ SOLVEABSTRACT(OPPONENT(p), $\langle k', s'\rangle, \emptyset$) ▷ Solve for the opponent
    if $u \neq \emptyset$ **then return** $\langle$`false`$, l, u\rangle$ ▷ Return counterexample
  end for
  return $\langle$`true`$, \emptyset, \emptyset\rangle$
end function

When performing model checking or synthesis with BDDs [5] 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 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 (Algorithm 3) effectively consists of two communicating solvers: the counterexample-guided bounded reachability solver and the unbounded solver based on incremental induction. The unbounded solver calls the bounded solver with an increasing bound (line 8) while learning states from abstract game trees that either must or may lose for the controller. When the initial state is contained within the must-losing states the game is known to be unrealisable (line 5).

May-losing states are learned by removing controller-winning states. We maintain a set of states for every rank up the current bound. The states we learn while executing the bounded algorithm can only said to be controller-winning for a particular rank or below. However, we construct each set carefully

such that it is monotonically larger than the set of the rank below (see Figure 2) and so that the environment is unable to force play from one set to the next. Due to this careful construction, 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).

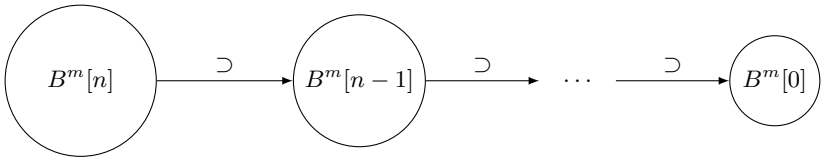


Fig. 2: May-losing states

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.

3.2 Learning States with Interpolants

Given two formulas A and B such that $A \wedge B$ is unsatisfiable, it is possible to construct a Craig interpolant[7] $\mathcal{I}$ 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 $A \wedge B$ [15].

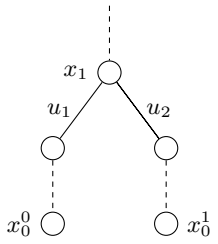


Fig. 3: A controller-losing game tree

Consider the snippet of a game tree in Figure 3. 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, 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 TREEFORMULA($step$) $\wedge$ TREEFORMULA($parent$) must too be unsatisfiable.

We can construct an interpolant with $A =$ TREEFORMULA($parent$) and $B =$ 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 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

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 (TREEFORMULA($parent$) $\wedge \mathcal{I}$) $\wedge$ TREEFORMULA($step$) 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 5). 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 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

3.3 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 [14].

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 $f^m[i]$:

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

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.

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 $\sigma(X_T) \wedge |treeFormula|(T_1)$ also satisfies $\mathcal{I}$. Hence, removing $\mathcal{I}$ from $B^m[\text{RANK}(n)]$ makes $\sigma(X_T) \wedge |treeFormula|(T_1)$ 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 sensible opponent moves

One aspect of the algorithm proposed by Narodytska et al[14] 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).

6 Related Work

Synthesis of safety games is a thoroughly explored area of research with most efforts directed toward solving games with BDDs [5] and abstract interpretation [16, 4]. Satisfiability solving has been used previously for synthesis in a suite of methods proposed by Bloem et al [2]. 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.

There are different approaches to bounded synthesis than the one described here. The authors of [10] suggest a methodology directly inspired by bounded model checking and it has been adapted to symbolic synthesis [8]. Lazy synthesis [9] is a counterexample-guided approach to bounded synthesis that refines an implementation for the game instead of an abstraction of it.

SAT-based bounded model checking approaches that unroll the transition relation have been extended to unbounded by using conflicts in the solver [11], or by interpolation [12]. However, there are no corresponding adaptations to synthesis. Incremental induction [3] is another technique for unbounded model checking with an equivalent synthesis method [13], which computes the set of safe states incrementally.

7 Conclusion

References

1. 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)
2. 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)
3. Bradley, A.R.: Sat-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 70–87 (2011)
4. 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)
5. 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)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification, CAV. pp. 154–169 (2000)
7. Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
8. Ehlers, R.: Symbolic bounded synthesis. In: Computer Aided Verification, CAV. pp. 365–379 (2010)
9. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 219–234 (2012)
10. Finkbeiner, B., Schewe, S.: Bounded synthesis. Software Tools for Technology Transfer, STTT 15(5-6), 519–539 (2013)
11. 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)
12. McMillan, K.L.: Interpolation and sat-based model checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2725, pp. 1–13 (2003)
13. Morgenstern, A., Gesell, M., Schneider, K.: Solving games using incremental induction. In: Integrated Formal Methods, IFM. pp. 177–191 (2013)
14. Narodytska, N., Legg, A., Bacchus, F., Ryzhyk, L., Walker, A.: Solving games without controllable predecessor. In: Computer Aided Verification. pp. 533–540 (2014)
15. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
16. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: Formal Methods in Computer-Aided Design FMCAD. pp. 219–226 (2014)