Counterexample-guided Control*

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar

EECS Department, University of California, Berkeley
{tah,jhala,rupak}@eecs.berkeley.edu

Abstract. A major hurdle in the algorithmic verification and control of systems is the need to find suitable abstract models, which omit enough details to overcome the state-explosion problem, but retain enough details to exhibit satisfaction or controllability with respect to the specification. The paradigm of counterexample-guided abstraction refinement suggests a fully automatic way of finding suitable abstract models: one starts with a coarse abstraction, attempts to verify or control the abstract model, and if this attempt fails and the abstract counterexample does not correspond to a concrete counterexample, then one uses the spurious counterexample to guide the refinement of the abstract model. We present a counterexample-guided refinement algorithm for solving ω-regular control objectives. The main difficulty is that in control, unlike in verification, counterexamples are strategies in a game between system and controller. In the case that the controller has no choices, our scheme subsumes known counterexample-guided refinement algorithms for the verification of ω-regular specifications. Our algorithm is useful in all situations where ω-regular games need to be solved, such as supervisory control, sequential and program synthesis, and modular verification. The algorithm is fully symbolic, and therefore applicable also to infinite-state systems.

1 Introduction

The key to the success of algorithmic methods for the verification (analysis) and control (synthesis) of complex systems is *abstraction*. Useful abstractions have two desirable properties. First, the abstraction should be *sound*, meaning that if a property (e.g., safety, controllability) is proved for the abstract model of a system, then the property holds also for the concrete system. Second, the abstraction should be *effective*, meaning that the abstract model is not too fine and can be handled by the tools at hand; for example, in order to use conventional model checkers, the abstraction must be both finite-state and of manageable size. Recent research has focused on a third desirable property of abstractions. A sound and effective abstraction (provided it exists) should be found *automatically*; otherwise, the labor-intensive process of constructing suitable abstract models often negates the benefits of automatic methods for verification and control. The most successful paradigm in automatic abstraction is the method of *counterexample-guided abstraction refinement* [5, 6, 9]. According to that paradigm, one starts with a very coarse abstract model, which is effective but may not be informative, meaning that it may not exhibit the desired property even if the concrete system does. Then the abstract model is refined iteratively as follows: first, if the abstract model does not exhibit the desired property, then an abstract counterexample is constructed automatically; second, it can be checked automatically if the abstract counterexample corresponds to a concrete counterexample; if this is not the case, then, third, the abstract model is refined automatically in order to eliminate the spurious counterexample.

The method of counterexample-guided abstraction refinement has been developed for the *verification* of linear-time properties [9], and universal branching-time properties [10]. It has been

* This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the ONR grant N00014-02-1-0671, and the NSF grants CCR-9988172, CCR-0085949, and CCR-0225610.

applied successfully in both hardware [9] and software verification [6, 18]. We develop the method of counterexample-guided abstraction refinement for the *control* of linear-time objectives. In verification, a counterexample to the satisfaction of a linear-time property is a *trace* that violates the property: for safety properties, a finite trace; for general ω-regular properties, an infinite, periodic (lasso-shaped) trace. In control, counterexamples are considerably more complicated: a counterexample to the controllability of a system with respect to a linear-time objective is a *tree* that represents a strategy of the system for violating the property no matter what the controller does. For safety objectives, finite trees are sufficient as counterexamples; for general ω-regular objectives on finite abstract models, infinite trees are necessary, but they can be finitely represented as graphs with cycles, because finite-state strategies are as powerful as infinite-state strategies [17].

In somewhat more detail, our method proceeds as follows. Given a two-player game structure (player 1 "controller" vs. player 2 "system"), we wish to check if player 1 has a strategy to achieve a given ω-regular winning condition. Solutions to this problem have applications in supervisory control [22], sequential hardware synthesis and program synthesis [8, 7, 21], modular verification [2, 4, 14], receptiveness checking [3, 15], interface compatibility checking [12], and schedulability analysis [1]. We automatically construct an abstraction of the given game structure that is as coarse as possible and as fine as necessary in order for player 1 to have a winning strategy. We start with a very coarse abstract game structure and refine it iteratively. First, we check if player 1 has a winning strategy in the abstract game; if so, then the concrete system can be controlled; otherwise, we construct an abstract player-2 strategy that spoils against all abstract player-1 strategies. Second, we check if the abstract player-2 strategy corresponds to a spoiling strategy for player 2 in the concrete game; if so, then the concrete system cannot be controlled; otherwise, we refine the abstract game in order to eliminate the abstract player-2 strategy. In this way, we automatically synthesize "maximally abstract" controllers, which distinguish two states of the controlled system only if they need to be distinguished in order to achieve the control objective. It should be noted that ω-regular verification problems are but special cases of ω-regular control problems, where player 1 (the controller) has no choice of moves. Our method, therefore, includes as a special case counterexample-guided abstraction refinement for linear-time *verification*.

Furthermore, our method is fully *symbolic*: while traditional symbolic verification computes fixpoints on the iteration of a transition-precondition operator on regions (symbolic state sets), and traditional symbolic control computes fixpoints on the iteration of a more general, game-precondition operator *Cpre* (controllable *Pre*) [4, 20], our counterexample-guided abstraction refinement also computes fixpoints on the iteration of *Cpre* and two additional region operators, called *Focus* and *Shatter*. The *Focus* operator is used to check if an abstract counterexample is genuine or spurious. The *Shatter* operator, which is used to refine an abstract model guided by a spurious counterexample, splits an abstract state into several states. Our top-level algorithm calls only these three system-specific operators: *Cpre*, *Focus*, and *Shatter*. It is therefore applicable not only to finite-state systems but also to infinite-state systems, such as hybrid systems, on which these three operators are computable (termination can be studied as an orthogonal issue along the lines of [13]; clearly, our abstraction-based algorithms terminate in all cases in which the standard, *Cpre*-based algorithms terminate, such as in the control of timed automata [20], and they may terminate in more cases).

In a previous paper, we improved the naive iteration of the "abstract-verify-refine" loop by integrating the construction of the abstract model and the verification process [18]. The improvement is called *lazy abstraction*, because the abstract model is constructed on demand during verification, which results in nonuniform abstractions, where some areas of the state space are abstracted more coarsely than others, and thus guarantees an abstract model that is as small as possible. The lazy-abstraction paradigm can be applied also to the algorithm presented here, which subsumes both verification and control. The details of this, however, need to be omitted for space reasons.

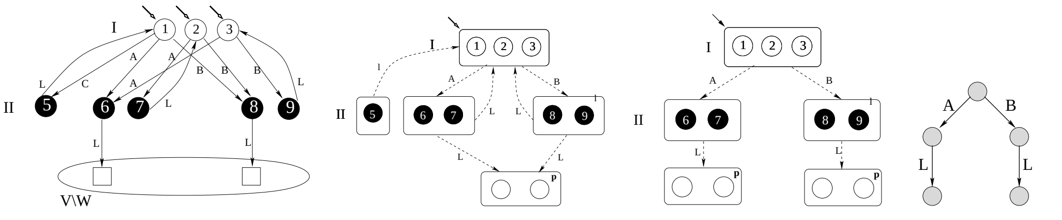
2 Games and Abstraction

Two-player games Let Λ be a set of labels, and Φ a set of propositions. A (*two-player*) *game structure* $\mathcal{G} = (V_1, V_2, \delta, P)$ consists of two (possibly infinite) disjoint sets V_1 and V_2 of player-1 and player-2 states (let $V = V_1 \cup V_2$ denote the set of all states), a labeled transition relation $\delta \subseteq V \times \Lambda \times V$, and a function P: $V \to 2^{\Phi}$ that maps every state to a set of propositions. For every state $v \in V$, we call $L(v) = \{l \in \Lambda \mid \exists w.\, (v, l, w) \in \delta\}$ the set of *available moves*. In the sequel, i ranges over the set $\{1, 2\}$ of players. Intuitively, at state $v \in V_i$, player i chooses a move $l \in L(v)$, and the game proceeds nondeterministically to some state w satisfying $\delta(v, l, w)$.[1] We require that every player-2 state $v \in V_2$ has an available move, that is, $L(v) \neq \emptyset$. For a move $l \in \Lambda$, let $Avl(l) = \{v \in V \mid l \in L(v)\}$ be the set of states in which move l is available. We extend the transition relation to sets via the operators $Apre, Epre$: $2^V \times \Lambda \to 2^V$ by defining $Apre(X, l) = \{v \in V \mid \forall w.\, \delta(v, l, w) \Rightarrow w \in X\}$ and $Epre(X, l) = \{v \in V \mid \exists w.\, \delta(v, l, w) \wedge w \in X\}$. For a proposition $p \in \Phi$, let $[p] = \{v \in V \mid p \in P(v)\}$ and $[\overline{p}] = V \setminus [p]$ be the sets of states in which p is true and false, respectively. We assume that Φ contains a special proposition *init*, which specifies a set $[init] \subseteq V$ containing the initial states. A *run* of the game structure $\mathcal{G}$ is a finite or infinite sequence $v_0 v_1 v_2 \ldots$ of states $v_j \in V$ such that for all $j \geq 0$, if v_j is not the last state of the run, then there is a move $l_j \in \Lambda$ with $\delta(v_j, l_j, v_{j+1})$. A *strategy of player i* is a partial function f_i: $V^* \cdot V_i \to \Lambda$ such that for every state sequence $u \in V^*$ and every state $v \in V_i$, if $L(v) \neq \emptyset$, then $f_i(u \cdot v)$ is defined and $f_i(u \cdot v) \in L(v)$. Intuitively, a player-i strategy suggests, when possible, a move for player i given a sequence of states that end in a player-i state. Given two strategies f_1 and f_2 of players 1 and 2, the *possible outcomes* $\Omega_{f_1,f_2}(v)$ from a state $v \in V$ are runs: a run $v_0 v_1 v_2 \ldots$ belongs to $\Omega_{f_1,f_2}(v)$ iff $v = v_0$ and for all $j \geq 0$, either $L(v_j) = \emptyset$ and v_j is the last state of the run, or $v_j \in V_i$ and $\delta(v_j, f_i(v_0 \ldots v_j), v_{j+1})$. Note that the last state of a finite outcome is always a player-1 state.

Winning conditions A *game* $(\mathcal{G}, \Gamma)$ consists of a game structure $\mathcal{G}$ and an objective Γ for player 1. We focus on safety games, and briefly discuss games with more general ω-regular objectives at the very end of the paper. A *safety game* has an objective of the form $\Box\overline{err}$, where $err \in \Phi$ is a proposition which specifies a set $[err] \subseteq V$ of error states. Intuitively, the goal of player 1 is to keep the game in states in which *err* is false, and the goal of player 2 is to drive the game into a state in which *err* is true. Moreover, in all games we consider, whenever a dead-end state is encountered, player 1 loses. Formally, a run $v_0 v_1 v_2 \ldots$ is *winning for player 1* if it is infinite and for all $j \geq 0$, we have $v_j \notin [err]$. Let Π_1 denote the set of runs that are winning for player 1. In general, an *objective* for player 1 is a set $\Gamma \subseteq (2^{\Phi})^{\omega}$ of infinite words over the alphabet 2^{Φ}, and Π_1 contains all infinite runs $v_0 v_1 \ldots$ such that $P(v_0), P(v_1), \ldots \in \Gamma$. The game starts from any initial state. A strategy f_1 is *winning for player 1* if for all strategies f_2 of player 2 and all states $v \in [init]$, we have $\Omega_{f_1,f_2}(v) \subseteq \Pi_1$; that is, all possible outcomes are winning for player 1. Dually, a strategy f_2 is *spoiling for player 2* if for all strategies f_1 of player 1, there is a state $v \in [init]$ such that $\Omega_{f_1,f_2}(v) \not\subseteq \Pi_1$. Note that in our setting, nondeterminism is always on the side of player 2. If the objective Γ is ω-regular, then either player 1 has a winning strategy or player 2 has a spoiling strategy [17]. We say that player 1 *wins* the game if there is a player-1 winning strategy.

Example 1 [ExSafety] Figure 1(a) shows an example of a safety game. The white states are player 1 states, and the black ones are player 2 states. The labels on the edges denote moves. The objective is $\Box\overline{p}$, that is, player 1 seeks to avoid the error states $[p]$. The player 1 states 1, 2, and 3 are the initial states, i.e., we wish player 1 to win from all three states. Note that in fact player 1 does win from the states 1, 2, and 3: at state 1, she plays the move C; at 2, she plays A; and

[1] Even if the transition relation is deterministic, abstractions of the game may be nondeterministic.



(a) Game (b) Abstraction (c) ACT T^α (d) *type*(T^α)

Fig. 1. Example ExSafety

at 3, she plays B. In each case, the only move L available to player 2 brings the game back to the original state. This ensures that the game never reaches a state in $[p]$. ∎

The (*player-1*) *controllable predecessor* operator $Cpre_1\colon 2^V \to 2^V$ denotes, for a set $X \subseteq V$ of states, the states from which player 1 can force the game into X in one step. Player 1 can force the game into X from a state $v \in V_1$ iff there is *some* available move l such that all l-successors of v are in X, and player 1 can force the game into X from a state $v \in V_2$ iff for *all* available moves l, all l-successors of v are in X. Formally:

$$Cpre_1(X) \;=\; (V_1 \;\cap\; \bigcup_{l \in \Lambda} (Avl(l) \cap Apre(X,l))) \;\cup\; (V_2 \;\cap\; \bigcap_{l \in \Lambda} Apre(X,l))$$

In particular, the set of states from which player 1 can keep the game away from *err* states is the greatest fixpoint $\nu X.\, [\overline{err}] \cap Cpre_1(X)$. Hence player 1 wins the safety game with objective $\Box\overline{err}$ iff $[init] \subseteq (\nu X.\, [\overline{err}] \cap Cpre_1(X))$.

Abstractions of games Since solving a game may be expensive, we wish to construct sound abstractions of the game with smaller state spaces. Soundness means that if player 1 wins the abstract game, then she wins also the original, concrete game. To ensure soundness, we restrict the power of player 1 and increase the power of player 2 [19]. Therefore, we abstract the player-1 states so that *fewer* moves are available, and the player-2 states so that *more* moves are available. An *abstraction* $\mathcal{G}^\alpha$ for the game structure $\mathcal{G}$ is a game structure $(V_1^\alpha, V_2^\alpha, \delta^\alpha, P^\alpha)$ and a concretization function $[\![\cdot]\!]\colon V^\alpha \to 2^V$ (where $V^\alpha = V_1^\alpha \cup V_2^\alpha$ is the abstract state space) such that conditions (1)–(3) hold. (1) The abstraction preserves the player structure and propositions: for $i \in \{1,2\}$ and all $v^\alpha \in V_i^\alpha$, we have $[\![v^\alpha]\!] \subseteq V_i$; for all $v^\alpha \in V^\alpha$, if $v, v' \in [\![v^\alpha]\!]$, then $P(v) = P(v')$ and $P^\alpha(v^\alpha) = P(v)$. (2) The abstract states cover the concrete state space: $\bigcup_{v^\alpha \in V^\alpha} [\![v^\alpha]\!] = V$. (3) For each player-1 abstract state $v^\alpha \in V_1^\alpha$, define $L^\alpha(v^\alpha) = \bigcap_{v \in [\![v^\alpha]\!]} L(v)$, and for each player-2 abstract state $v^\alpha \in V_2^\alpha$, define $L^\alpha(v^\alpha) = \bigcup_{v \in [\![v^\alpha]\!]} L(v)$. Then, for all $v^\alpha, w^\alpha \in V^\alpha$ and all $l \in \Lambda$, we have $\delta^\alpha(v^\alpha, l, w^\alpha)$ iff $l \in L^\alpha(v^\alpha)$ and there are states $v \in [\![v^\alpha]\!]$ and $w \in [\![w^\alpha]\!]$ with $\delta(v, l, w)$. Note that the abstract state space V^α and the concretization function $[\![\cdot]\!]$ uniquely determine the abstraction $\mathcal{G}^\alpha$. Intuitively, each abstract state $v^\alpha \in V^\alpha$ represents a set $[\![v^\alpha]\!] \subseteq V$ of concrete states. We will use only abstractions with finite state spaces. The controllable predecessor operator on the abstract game structure $\mathcal{G}^\alpha$ is denoted $Cpre_1^\alpha$.

Proposition 1 [Soundness of abstraction] *Let $\mathcal{G}^\alpha$ be an abstraction for a game structure $\mathcal{G}$, and let Γ be an objective for player 1. If player 1 wins the abstract game $(\mathcal{G}^\alpha, \Gamma)$, then player 1 also wins the concrete game $(\mathcal{G}, \Gamma)$.*

Example 2 [ExSafety] Figure 1(b) shows one particular abstraction for the game structure from Figure 1(a). The boxes denote abstract states with the states they represent drawn inside them. The dashed arrows are the abstract transitions. Note that from the starting player 1 box, the move C is not available, because it is not available at states 2 and 3. as not all the states in the box can do it. In the abstract game, player 2 has a spoiling strategy: after player 1 plays either move A or move B, player 2 can play move L and take the game to the error set $[p]$. ∎

3 Counterexample-guided Abstraction Refinement

A *counterexample* to the claim that player 1 can win a game is a spoiling strategy for player 2. A counterexample for an abstract game $(\mathcal{G}^\alpha, \Gamma)$ may be either *genuine*, meaning that it corresponds to a counterexample for the concrete game $(\mathcal{G}, \Gamma)$, or *spurious*, meaning that it arises due to the coarseness of the abstraction. In the sequel, we check whether or not an abstract counterexample is genuine for a a fixed safety game $(\mathcal{G}, \Box\overline{err})$ and abstraction $\mathcal{G}^\alpha$. Moreover, if the counterexample is spurious, then we refine the abstraction in order to rule out that particular counterexample.

Abstract counterexample trees Our abstract games are finite-state, and for safety games, memoryless spoiling strategies suffice for player 2. Finite trees are therefore a natural representation of counterexamples. We work with rooted, directed, finite trees with labels on both nodes and edges. Each node is labeled by an abstract state $v^\alpha \in V^\alpha$ or a concrete state $v \in V$, and possibly a set $r \subseteq V$ of concrete states. We write $\mathtt{n} : v^\alpha$ for node $\mathtt{n}$ labeled with v^α, and $\mathtt{n} : v^\alpha : r$ if $\mathtt{n}$ is labeled with both v^α and r. Each edge is labeled with a move $l \in \Lambda$. If $\mathtt{n} \xrightarrow{l} \mathtt{n}'$ is an edge labeled by l, then $\mathtt{n}'$ is called an l-child of $\mathtt{n}$. A leaf is a node without children. For two trees S and T, we write $S \preceq T$ iff S is a connected subgraph of T which contains the root of T. The *type* of a labeled tree T results from T by removing all node labels (but keeping all edge labels). Furthermore, $\mathit{Subtypes}(T) = \{\mathit{type}(S) \mid S \preceq T\}$. An *abstract counterexample tree* (ACT) T^α is a finite tree whose nodes are labeled by abstract states such that conditions (1)–(4) hold. (1) If the root is labeled by v^α, then $[\![v^\alpha]\!] \subseteq [\mathit{init}]$. (2) If $\mathtt{n}' : w^\alpha$ is an l-child of $\mathtt{n} : v^\alpha$, then $(v^\alpha, l, w^\alpha) \in \delta^\alpha$. (3) If node $\mathtt{n} : v^\alpha$ is a nonleaf player-1 node (that is, $v^\alpha \in V_1^\alpha$), then for *each* move $l \in L^\alpha(v^\alpha)$, the node $\mathtt{n}$ has at least one l-child. Note that if node $\mathtt{n} : v^\alpha$ is a nonleaf player-2 node ($v^\alpha \in V_2^\alpha$), then for *some* move $l \in L^\alpha(v^\alpha)$, the node $\mathtt{n}$ has at least one l-child. (4) If a leaf is labeled by v^α, then either $v^\alpha \in V_1^\alpha$ and $L^\alpha(v^\alpha) = \emptyset$, or $[\![v^\alpha]\!] \subseteq [\mathit{err}]$. Intuitively, T^α corresponds to a *set* of spoiling strategies for player 2 in the abstract safety game.

Example 3 [ExSafety] Figure 1(c) shows an ACT T^α for the abstract game of Figure 1(b), and Figure 1(d) shows the type of T^α. After player 1 plays either move A or move B, player 2 plays L to take the game to the error set. ∎

Concretizing abstract counterexamples A *concrete counterexample tree* (CCT) S is a finite tree whose nodes are labeled by concrete states such that conditions (1)–(4) hold. (1) If the root is labeled by v, then $v \in [\mathit{init}]$. (2) If $\mathtt{n}' : w$ is an l-child of $\mathtt{n} : v$, then $(v, l, w) \in \delta$. (3) If node $\mathtt{n} : v$ is a nonleaf player-1 node ($v \in V_1$), then for each move $l \in L(v)$, the node $\mathtt{n}$ has at least one l-child. (4) If a leaf is labeled by v, then either $v \in V_1$ and $L(v) = \emptyset$, or $v \in [\mathit{err}]$. The CCT S *realizes* the ACT T^α if $\mathit{type}(S) \in \mathit{Subtypes}(T^\alpha)$ and for each node $\mathtt{n} : w$ of S and corresponding node $\mathtt{n} : v^\alpha$ of T^α, we have $w \in [\![v^\alpha]\!]$. The ACT T^α is *genuine* if there is a CCT that realizes T^α, and otherwise T^α is *spurious*. To determine if the ACT T^α is genuine, we annotate every node $\mathtt{n} : v^\alpha$ of T^α, in addition, with a set $r \subseteq [\![v^\alpha]\!]$ of concrete states; that is, $\mathtt{n} : v^\alpha : r$. The result is called an *annotated* ACT. The set r represents an overapproximation for the set of states that

Algorithm 1 AnalyzeCounterex(T^α)

Input: an abstract counterexample tree T^α with root $\mathtt{n}_0$.
Output: if T^α is spurious, then SPURIOUS and an annotation of T^α; otherwise GENUINE.
for each node $\mathtt{n}\!:\!v^\alpha$ of T^α **do** annotate $\mathtt{n}\!:\!v^\alpha$ by $[\![v^\alpha]\!]$
while there is some node $\mathtt{n}\!:\!v^\alpha\!:\!r$ with $r \neq \mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$ **do**
  replace the annotation r of $\mathtt{n}\!:\!v^\alpha\!:\!r$ by $\mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$
  if $r_0 = \emptyset$ for the annotated root $\mathtt{n}_0\!:\!\_\!:\!r_0$ **then return** (SPURIOUS, T^α with annotations)
  end while
return GENUINE

can be part of a CCT with a type in $\mathit{Subtypes}(T^\alpha)$. Initially, $r = [\![v^\alpha]\!]$. The overapproximation r is sharpened repeatedly by application of a symbolic operator called *Focus*. For a node $\mathtt{n}$ of T^α, let $C(\mathtt{n}) = \{l \in \Lambda \mid \mathtt{n} \text{ has an } l\text{-child}\}$ be the set of moves that label the outgoing edges of $\mathtt{n}$. For each move $l \in C(\mathtt{n})$, let $\{\mathtt{n}_{l,j}\!:\!v^\alpha_{l,j}\!:\!r_{l,j}\}$ be the set of l-children of $\mathtt{n}$ (indexed by j). The operator $\mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$ returns a subset of r:

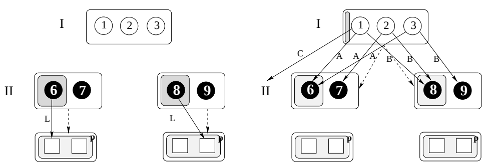
$$\mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r) = \begin{cases} r & \text{if } \mathtt{n} \text{ leaf and } L^\alpha(v^\alpha) \neq \emptyset \\ r \cap \left(\bigcap_{l \in C(\mathtt{n})} \mathit{Epre}(\cup_j r_{l,j}, l)\right) \cap \left(\bigcap_{l \notin C(\mathtt{n})} \overline{\mathit{Avl}(l)}\right) & \text{if } \mathtt{n} \text{ other player-1 node} \\ r \cap \left(\bigcup_{l \in C(\mathtt{n})} \mathit{Epre}(\cup_j r_{l,j}, l)\right) & \text{if } \mathtt{n} \text{ player-2 node} \end{cases}$$

An application of $\mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$ sharpens the set r by determining which of the states in r actually have successors that can be part of a spoiling strategy for player 2 in the concrete game. For leaves $\mathtt{n}\!:\!v^\alpha\!:\!r$ with $L^\alpha(v^\alpha) \neq \emptyset$, it must be that every state in r is an error state, and so can be part of a CCT. For all other player-1 nodes $\mathtt{n}\!:\!v^\alpha\!:\!r$, a state $v \in r$ can be part of a CCT only if (i) all moves available at v are contained in $C(\mathtt{n})$ and (ii) for every available move l, there is an l-child from which player 2 has a spoiling strategy; that is, for every available move l, the state v must have a successor in the union of all l-children's overapproximations. For player-2 nodes $\mathtt{n}\!:\!v^\alpha\!:\!r$, a state $v \in r$ can be part of a CCT only if there is some child from which player 2 has a spoiling strategy; that is, the state v must have a successor in the union of all children's overapproximations.

The procedure AnalyzeCounterex (Algorithm 1) iterates the *Focus* operator on the nodes of a given ACT T^α until there is no change. Let $\mathit{Focus}^*(\mathtt{n})$ denote the fixpoint value of the annotation for node $\mathtt{n}$ of T^α. For the root $\mathtt{n}_0$ of T^α, if $\mathit{Focus}^*(\mathtt{n}_0)$ is empty, then T^α is spurious. Otherwise, consider the annotated ACT that results from T^α by annotating each node $\mathtt{n}$ with $\mathit{Focus}^*(\mathtt{n})$, and removing all nodes $\mathtt{n}$ for which $\mathit{Focus}^*(\mathtt{n})$ is empty. This annotated ACT has a type in $\mathit{Subtypes}(T^\alpha)$, and moreover, its annotations contain exactly the states that can be part of a CCT that realizes T^α. Consequently, if $\mathit{Focus}^*(\mathtt{n}_0)$ is nonempty, then T^α is genuine, and the result of the procedure AnalyzeCounterex is a representation of the CCTs that realize T^α. The nondeterminism in the while loop of AnalyzeCounterex can be efficiently resolved by focusing each node after focusing all of its children. Since T^α is a finite tree, in this bottom-up way, each node is focused exactly once. Indeed, for finite-state game structures and nonsymbolic representations of ACTs, where all node annotations are stored as lists of concrete states, algorithm AnalyzeCounterex can be implemented in linear time.

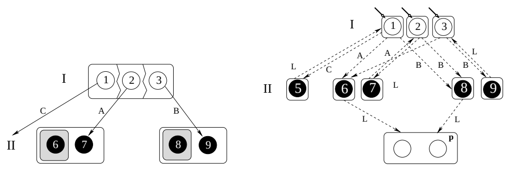
Proposition 2 [Counterexample checking] *An ACT T^α for a safety game is spurious iff the procedure* AnalyzeCounterex(T^α) *returns* SPURIOUS. *Checking if an ACT for a safety game is spurious can be done in time linear in the size of the tree.*

Example 4 [EXSAFETY] Figure 2 shows the result of running AnalyzeCounterex on the ACT T^α from Figure 1(c). The shaded parts of the boxes denote the states that may be a part of a



(a) (b)

Fig. 2. Focusing T^α



(a) Shattering (b) Refined abstraction

Fig. 3. Abstraction refinement

CCT. The dashed arrows indicate abstract transitions, and the solid arrows concrete transitions. Figure 2(a) shows the result of focusing the player 2 nodes. All states in the leaves are error states, and therefore in the shaded boxes. Only states 6 and 8 can go to the error region from the two abstract player-2 states; hence only they are in the focused regions indicated by shaded boxes. Figure 2(b) shows the result of a subsequent application of *Focus* to the root. No state in the root can play only moves A and B and subsequently go to states from which player 2 can spoil. Hence none of these states can serve as the root of a CCT whose type is in $\mathit{Subtypes}(T^\alpha)$. Since the focused region of the root is empty, we conclude that the ACT T^α is spurious. ∎

Abstraction refinement If we find an ACT T^α to be spurious, then we must refine the abstraction $\mathcal{G}^\alpha$ in order to rule out T^α. Consider a state $\mathtt{n}:v^\alpha$ of T^α. Abstraction refinement may split the abstract state v^α into several states $v_1^\alpha, \ldots, v_m^\alpha$, with $[\![v_k^\alpha]\!] = r_k$ for $1 \le k \le m$, such that $r_1 \cup \ldots \cup r_m = [\![v^\alpha]\!]$. For this purpose, we define a symbolic *Shatter* operator, which takes, for a node $\mathtt{n}:v^\alpha:r$ of the annotated version of T^α generated by the procedure AnalyzeCounterex, the triple $(\mathtt{n}, [\![v^\alpha]\!], r)$, and returns the set $\{r_1, \ldots, r_m\}$. The set r_1 is the "good" set r (the annotation), from which player 2 does indeed have a spoiling strategy of a type in $\mathit{Subtypes}(T^\alpha)$. The sets $r_2, \ldots, r_m$ are "bad" subsets of $[\![v^\alpha]\!] \setminus r$, from which no such spoiling strategy exists. Each "bad" set r_k, for $2 \le k \le m$, is small enough that there is a simple single reason for the absence of a spoiling strategy. For player-1 nodes $\mathtt{n}$, a set r_k may be "bad" because every state $v \in r_k$ either (i) has a move available which is not in $C(\mathtt{n})$, or (ii) has a move l available such that none of the l-successors of v is in a "good" set, from which player 2 can spoil. For player-2 nodes $\mathtt{n}$, there is a single "bad" set, which contains the states that have no successor in a "good" set. Formally, the operator $\mathit{Shatter}(\mathtt{n}, q, r)$ is defined to take a node $\mathtt{n}$ of the ACT T^α, and two sets $q, r \subseteq V$ of concrete states such that $r \subseteq q$, and it returns a collection $R \subseteq 2^V$ of state sets $r_k \subseteq q$. For each move $l \in C(\mathtt{n})$, let $\{\mathtt{n}_{l,j}:v_{l,j}^\alpha:r_{l,j}\}$ again be the set of l-children of $\mathtt{n}$. Then:

$$\mathit{Shatter}(\mathtt{n}, q, r) \;=\; \begin{cases} \{r\} \cup \{(q \backslash r) \cap \mathit{Avl}(l) \mid l \notin C(\mathtt{n})\} & \\ \qquad \cup \{(q \backslash r) \cap \overline{\mathit{Epre}(\cup_j r_{l,j}, l)} \mid l \in C(\mathtt{n})\} & \text{if } \mathtt{n} \text{ is a player-1 node} \\ \{r,\ (q \backslash r)\} & \text{if } \mathtt{n} \text{ is a player-2 node} \end{cases}$$

Note that $\bigcup \mathit{Shatter}(\mathtt{n}, q, \mathit{Focus}(\mathtt{n} : v^\alpha : q)) = q$. The refinement of the given abstraction $\mathcal{G}^\alpha$ is achieved by the procedure RefineAbstraction (Algorithm 2). Given a collection $R \subseteq 2^V$ of state sets, define the equivalence relation $\equiv_R \subseteq V \times V$ by $v_1 \equiv_R v_2$ if for all sets $r \in R$, we have $v_1 \in r$ precisely when $v_2 \in r$. Let $\mathit{Closure}(R)$ denote the equivalence classes of $\equiv_R$. Given $V \subseteq \bigcup R$, the set $\mathit{Closure}(R) \subseteq 2^V$ of sets of concrete states uniquely specifies an abstraction for $\mathcal{G}$, denoted $\mathit{Abstraction}(R)$, which contains for each set $r \in \mathit{Closure}(R)$ an abstract state w_r^α with

Algorithm 2 $\mathsf{RefineAbstraction}(\mathcal{G}^\alpha, T^\alpha)$

Input: an abstraction $\mathcal{G}^\alpha$ and an abstract counterexample tree T^α.
Output: if T^α is spurious, then SPURIOUS and a refined abstraction; otherwise GENUINE.
if $\mathsf{AnalyzeCounterex}(T^\alpha) = (\textsc{Spurious}, S^\alpha)$ **then**
  $R := \{[\![v^\alpha]\!] \mid v^\alpha \in V^\alpha\}$
  for each annotated node $\mathtt{n}\!:\!v^\alpha\!:\!r$ of S^α **do** $R := R \cup \mathit{Shatter}(\mathtt{n}, [\![v^\alpha]\!], r)$
  return (SPURIOUS, $\mathit{Abstraction}(R)$)
else return GENUINE

Algorithm 3 $\mathsf{CxSafetyControl}(\mathcal{G}, \Box\overline{err})$

Input: a game structure $\mathcal{G}$ and a safety objective $\Box\overline{err}$.
Output: either CONTROLLABLE and a player-1 winning strategy,
  or UNCONTROLLABLE and a player-2 spoiling strategy represented as ACT.
$\mathcal{G}^\alpha := \mathit{InitialAbstraction}(\mathcal{G}, \Box\overline{err})$
repeat
  $(\mathit{winner}, T^\alpha) := \mathit{ModelCheck}(\mathcal{G}^\alpha, \Box\overline{err})$
  if $\mathit{winner} = 2$ and $\mathsf{RefineAbstraction}(\mathcal{G}^\alpha, T^\alpha) = (\textsc{Spurious}, \mathcal{H}^\alpha)$ **then** $\mathcal{G}^\alpha := \mathcal{H}^\alpha$; $\mathit{winner} := \perp$
until $\mathit{winner} \neq \perp$
if $\mathit{winner} = 1$ **then return** (CONTROLLABLE, T^α)
return (UNCONTROLLABLE, T^α)

$[\![w_r^\alpha]\!] = r$ (from this the other components of the abstraction are determined). In particular, let $R_1 = \bigcup_{(\mathtt{n}v^\alpha) \in T^\alpha} \mathit{Shatter}(\mathtt{n}, [\![v^\alpha]\!], \mathit{Focus}^*(\mathtt{n}))$ and $R_2 = \{[\![v^\alpha]\!] \mid v^\alpha \in V^\alpha\}$. Our refined abstraction is $\mathit{Abstraction}(R_1 \cup R_2)$. The new abstraction returned by the procedure $\mathsf{RefineAbstraction}(\mathcal{G}^\alpha, T^\alpha)$ rules out ACTs that are similar to the spurious ACT T^α. Given two ACTs T^α and S^α, we say that T^α *subsumes* S^α if $\mathit{type}(S^\alpha) \in \mathit{Subtypes}(T^\alpha)$ and for each node $\mathtt{n} : w^\alpha$ of S^α and corresponding node $\mathtt{n}\!:\!v^\alpha$ of T^α, we have $[\![w^\alpha]\!] \subseteq [\![v^\alpha]\!]$.

Proposition 3 [Abstraction refinement] *If T^α is a spurious ACT for the abstraction $\mathcal{G}^\alpha$ of a safety game, then the abstraction returned by the procedure* $\mathsf{RefineAbstraction}(\mathcal{G}^\alpha, T^\alpha)$ *has no ACT that is subsumed by T^α.*

Example 5 [EXSAFETY] Figure 3 shows the effect of the *Shatter* operator on the root of the ACT T^α from Figure 1(c), and the resulting refined abstract game for which T^α is no longer an ACT. For all nonroot nodes, shattering is trivial, namely, into the focused region and its complement. We break up the states in the root into (i) state 1, which can play the move C not available to the abstract state, (ii) state 2, which can proceed by move A to a state from which the abstract player-2 spoiling strategy fails (i.e., a state not inside a shaded box), and (iii) state 3, which can proceed by move B to a state from which the abstract player-2 spoiling strategy fails. ∎

4 Counterexample-Guided Controller Synthesis

Safety control Given a game structure $\mathcal{G}$ and a safety objective $\Box\overline{err}$, we wish to determine if player 1 wins, and if so, construct a winning strategy ("synthesize a controller"). Our algorithm, which generalizes the "abstract-verify-refine" loop of [5, 6, 9], proceeds as follows:

Step 1 ("abstraction") We first construct an initial abstract game $(\mathcal{G}^\alpha, \Box\overline{err})$. This could be the trivial abstraction induced by the two propositions *init* and *err*, which has at most 8 abstract states (at most 4 for each player, depending on which of the two propositions are true).

Step 2 ("model checking") We symbolically model check the abstract game to find if player 1 can win, by iterating the $Cpre_1^{\alpha}$ operator. If so, then the model checker provides a winning player-1 strategy for the abstract game, from which a winning player-1 strategy in the concrete game can be constructed [13]. If not, then the model checker symbolically produces an ACT [11]. As the abstract state space is finite, the model checking is guaranteed to terminate.

Step 3 ("counterexample-guided abstraction refinement") If model checking returns an ACT T^{α}, then we use the procedure $\mathsf{AnalyzeCounterex}(T^{\alpha})$ to check if the ACT is genuine. If so, then player 2 has a spoiling strategy in the concrete game, and the system is not controllable. If the ACT is spurious, then we use the procedure $\mathsf{RefineAbstraction}(\mathcal{G}^{\alpha}, T^{\alpha})$ to refine the abstraction $\mathcal{G}^{\alpha}$, so that T^{α} (and similar counterexamples) cannot arise on subsequent invocations of the model checker. This step uses the operators *Focus* and *Shatter*, which are defined in terms of *Epre* and can therefore be implemented symbolically. Since T^{α} is a finite tree, also this step is guaranteed to terminate.

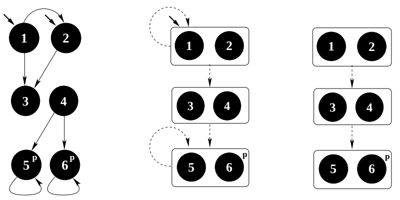
Goto step 2. The process is iterated until we find either a player-1 winning strategy in step 2, or a genuine counterexample in step 3.

The procedure is summarized in Algorithm 3. The function $\mathit{InitialAbstraction}(\mathcal{G}, \Box\overline{err})$ returns a trivial abstraction for $\mathcal{G}$, which preserves *init* and *err*. The function $\mathit{ModelCheck}(\mathcal{G}^{\alpha}, \Box\overline{err})$ returns a pair $(1, T^{\alpha})$ if player 1 can win the abstract game, where T^{α} is a (memoryless) winning strategy for player 1, and otherwise it returns $(2, T^{\alpha})$, where T^{α} is an ACT. From the soundness of abstraction, we get the soundness of the algorithm.

Proposition 4 [Partial correctness of $\mathsf{CxSafetyControl}$] *If the procedure* $\mathsf{CxSafetyControl}(\mathcal{G}, \Box\overline{err})$ *returns* CONTROLLABLE, *then player 1 wins the safety game* $(\mathcal{G}, \Box\overline{err})$. *If the procedure returns* UNCONTROLLABLE, *then player 1 does not win the game.*

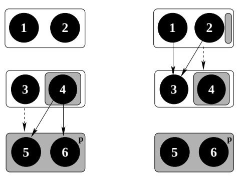
In general, the procedure $\mathsf{CxSafetyControl}$ may not terminate for infinite-state games (it does terminate for finite-state games). However, one can prove sufficient conditions for termination provided certain state equivalences on the game structure have finite index [13]. For example, for timed games [3, 20], where in the course of the procedure $\mathsf{CxSafetyControl}$, the abstract state space always consists of blocks of clock regions, termination is guaranteed. Verification is the special case of control where all states are player-2 states. Hence our algorithm works also for verification, which is illustrated by the following example.

Example 6 [Safety verification] Consider the transition system EXVERIF shown in Figure 4(a). All states are player-2 states. The initial states are 1 and 2, and we wish to check the safety property $\Box\overline{p}$, that the states 5 and 6 are not visited by any run. It is easy to see that the system satisfies this property. Figure 4(b) shows an abstraction for EXVERIF. This is a standard existential abstraction for transition systems. In verification, counterexaples are traces (trees without branches). Figure 4(c) shows a trace τ^{α}, which is an ACT for the abstraction (b). Figure 5 shows the result of running the algorithm $\mathsf{AnalyzeCounterex}$ on τ^{α}. Figure 5(a) shows the effect of applying *Focus* to the second abstract state in τ^{α}. All concrete states in the third abstract state are error states; hence they are all shaded. Only state 4 can go to one of the error states; hence it is the only state in the focused region of the second abstract state. Figure 5(b) shows the second application of *Focus*, to the root of the trace. As neither 1 nor 2 have 4 as a successor, the focused region of the root is empty. This implies that the counterexample is spurious. Figure 6(a) shows the effect of *Shatter* on the abstract trace τ^{α}. Since the shaded box of the second abstract state is $\{4\}$, this abstract state gets shattered into $\{3\}$ and $\{4\}$. No other abstract state is shattered. Figure 6(b) shows the refined abstraction, which is free of counterexamples. ∎



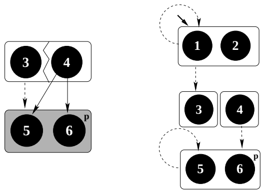
(a) (b) (c) τ^α

Fig. 4. Example ExVerif



(a) (b)

Fig. 5. Focusing τ^α



(a) (b)

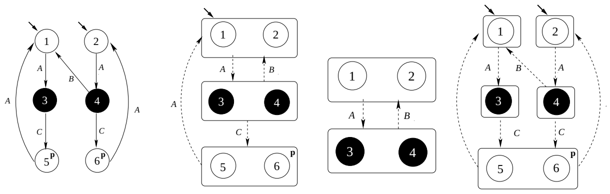
Fig. 6. Refinement

Algorithm 4 CombinedAnalyzeRefine($\mathcal{G}^\alpha, K^\alpha$)

Input: an abstraction $\mathcal{G}^\alpha$, and an abstract counterexample graph K^α with root $\mathtt{n}_0$.
Output: if K^α is spurious, then Spurious and a refined abstraction; otherwise genuine.
for each node $\mathtt{n}\!:\!v^\alpha$ of K^α **do** annotate $\mathtt{n}\!:\!v^\alpha$ by $[\![v^\alpha]\!]$
$R := \{[\![v^\alpha]\!] \mid v^\alpha \in V^\alpha\}$
while there is some node $\mathtt{n}\!:\!v^\alpha\!:\!r$ with $r \neq \mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$ **do**
  $r' := \mathit{Focus}(\mathtt{n}\!:\!v^\alpha\!:\!r)$
  $R := R \cup \mathit{Shatter}(\mathtt{n}, r, r')$
  replace the annotation r of $\mathtt{n}\!:\!v^\alpha\!:\!r$ by r'
  if $r_0 = \emptyset$ for the annotated root $\mathtt{n}_0\!:\!\_\!:\!r_0$ **then return** (Spurious, $\mathit{Abstraction}(R)$)
  end while
return Genuine

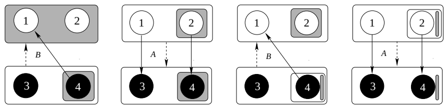
Omega-regular objectives Counterexample-guided abstraction refinement can be generalized to games with arbitrary ω-regular objectives. To begin with, we must implement a symbolic model checker for solving ω-regular games: given a finite-state game structure $\mathcal{G}^\alpha$ and an ω-regular objective $\mathit{\Gamma}$, one can construct a fixpoint formula over the Cpre_1^α operator which characterizes the set of states from which player 1 can win [13]. Moreover, from the fixpoint computation, one can symbolically construct either a winning strategy for player 1 or a spoiling strategy for player 2 [13, 20]. Counterexamples for finite-state ω-regular games are spoiling strategies with finite memory [17], which can be represented as finite graphs. Hence we generalize ACTs from trees to graphs as follows: an *abstract counterexample graph* (ACG) K^α is a rooted, directed, finite graph whose nodes are labeled by abstract states such that conditions (1)–(3) from the definition of ACT hold, and (4) if a leaf (a node with outdegree 0) is labeled by v^α, then $v^\alpha \in V_1^\alpha$ and $L^\alpha(v^\alpha) = \emptyset$. The definition of concrete counterexamples and of the operator *Subtypes* are generalized from trees to graphs in a similar, straight-forward way, giving rise to the notion of whether an ACG is *genuine* or *spurious*. So suppose that the function $\mathit{ModelCheck}(\mathcal{G}^\alpha, \mathit{\Gamma})$ returns a pair $(1, K^\alpha)$ if player 1 can win the abstract game, where K^α is a (finite-memory) winning strategy for player 1, and otherwise returns $(2, K^\alpha)$, where K^α is an ACG. In the latter case we must now check whether or not K^α is spurious, and if so, then refine the abstraction $\mathcal{G}^\alpha$.

While in the safety case, we analyzed counterexamples (Algorithm 1) before we refined the abstraction (Algorithm 2), for general ω-regular objectives, we combine both procedures (Algorithm 4). The algorithm CombinedAnalyzeRefine computes the fixpoint of the *Focus* operator on a given ACG K^α, and simultaneously refines the given abstraction $\mathcal{G}^\alpha$ by shattering an abstract state with each application of *Focus*. In contrast to the case of trees, for general graphs we cannot apply



(a) Game (b) Abstraction (c) K^α (d) Refinement

Fig. 7. Example ExBüchi



(a) (b) (c) (d)

Fig. 8. CombinedAnalyzeRefine on K^α

a bottom-up strategy for focusing. Indeed, in the presence of cycles, the computation of *Focus** may require focusing a node several times before a fixpoint is reached, and CombinedAnalyzeRefine is not guaranteed to terminate (it does terminate for finite-state games). It is easy to see that the procedures AnalyzeCounterex and RefineAbstraction are a special case of CombinedAnalyzeRefine for the case that each node needs to be focused only once. In this case, all shattering can be delayed until focusing is complete, and thus repeated shattering while refocusing the same abstract state can be avoided. Suppose that the procedure CxControl is obtained from CxSafetyControl (Algorithm 3) by replacing the safety objective $\Box\overline{err}$ with an arbitrary ω-regular objective Γ, and by calling the function CombinedAnalyzeRefine in place of RefineAbstraction. Then we have the following result.

Theorem 1. [Partial correctness of CxControl] *Let $\mathcal{G}$ be a game structure, and let Γ be an ω-regular objective. If the procedure* CxControl($\mathcal{G}, \Gamma$) *returns* Controllable, *then player 1 wins the game; if the procedure returns* Uncontrollable, *then player 1 does not win.*

Example 7 [Büchi game] Figure 7(a) shows an example of a Büchi game. We wish to check if player 1 can force the game into a p-state infinitely often, i.e., the objective is $\Box\Diamond p$. Figure 7(b) shows an abstraction for the game. Figure 7(c) shows the result of solving the abstract game, namely, an ACG K^α that has player 2 force a loop not containing a p-state. Figure 8 shows how the ACG is analyzed and discovered to be spurious. Figure 8(a) shows the effect of *Focus* on the lower node of K^α. As only the state 4 has a move into $\{1, 2\}$, the shaded box for the lower node is $\{4\}$. Consequently, the abstract state $\{3, 4\}$ is shattered into $\{3\}$ and $\{4\}$. Figure 8(b) shows the effect of *Focus* on the upper node of K^α. Only state 2 has an A-successor in the shaded box of the lower node; hence the focused region for the upper node becomes $\{2\}$, and the upper node gets shattered into $\{1\}$ and $\{2\}$. In Figure 8(c) we again apply *Focus* to the lower node. Since no state has a B-move to the focused region of the upper node, the focused region of the lower node becomes empty. Figure 8(d) illustrates that after another *Focus* on the upper node, its focused region becomes empty as well. Figure 7(d) shows the resulting refined abstraction; it is easy to see that player 2 has no spoiling strategy. ∎

In [10], the authors consider counterexample-guided abstraction refinement for model checking universal CTL formulas. In this case (and for some more expressive logics considered in [10]), counterexamples are tree-like, and our algorithms for analyzing counterexamples and refining abstractions apply also (indeed, since in this case counterexamples are models of existential CTL formulas, abstract counterexample trees contain only player-2 nodes). More generally, the model-checking problem for the μ-calculus can be reduced to the problem of solving parity games [16].

Via this reduction, our method provides also a counterexample-guided abstraction refinement procedure for model checking the μ-calculus.

References

1. K. Altisen, G. Gössler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *RTSS: Real-Time Systems Symposium*, pages 154–163. IEEE, 1999.
2. R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In *CONCUR: Concurrency Theory*, LNCS 1664, pages 82–97. Springer, 1999.
3. R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR: Concurrency Theory*, pages 74–88. LNCS 1243, Springer, 2001.
4. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.
5. R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118:142–157, 1995.
6. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL: Principles of Programming Languages*, pages 1–3. ACM, 2002.
7. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the AMS*, 138:295–311, 1969.
8. A. Church. Logic, arithmetic, and automata. In *International Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1962.
9. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.
10. E.M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *LICS: Logic in Computer Science*, pages 19–29. IEEE, 2002.
11. E.M. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC: Design Automation Conference*, pages 427–432. ACM/IEEE, 1995.
12. L. de Alfaro and T.A. Henzinger. Interface automata. In *FSE: Foundations of Software Engineering*, pages 109–120. ACM, 2001.
13. L. de Alfaro, T.A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR: Concurrency Theory*, pages 536–550. LNCS 2154, Springer, 2001.
14. L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Detecting errors before reaching them. In *CAV: Computer-Aided Verification*, LNCS 1855, pages 186–201. Springer, 2000.
15. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.
16. E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model checking fragments of μ-calculus. In *CAV: Computer-Aided Verification*, LNCS 697, pages 385–396. Springer, 1993.
17. Y. Gurevich and L. Harrington. Trees, automata, and games. In *STOC: Symposium on Theory of Computing*, pages 60–65. ACM, 1982.
18. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL: Principles of Programming Languages*, pages 58–70. ACM, 2002.
19. T.A. Henzinger, R. Majumdar, F.Y.C. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In *SAS: Static-Analysis Symposium*, pages 220–239. LNCS 1824, Springer, 2000.
20. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS: Theoretical Aspects of Computer Science*, LNCS 900, pages 229–242. Springer, 1995.
21. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL: Principles of Programming Languages*, pages 179–190. ACM, 1989.
22. P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25:206–230, 1987.