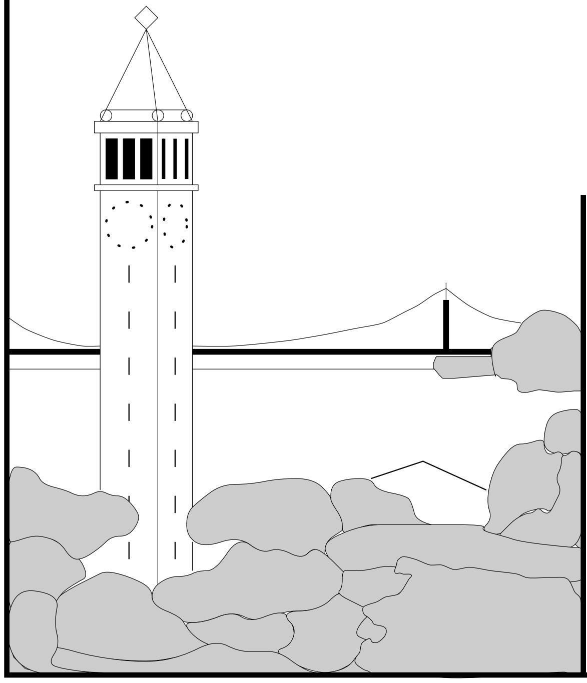
Counterexample Guided Control

Thomas A. Henzinger *Ranjit Jhala* *Rupak Majumdar*



Report No. UCB//CSD-02-1213

November 12, 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Counterexample Guided Control*

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
Department of EECS, UC 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 scheme for counterexample-guided refinement with the following properties. First, our scheme is the first such method for *control*. The main difficulty here is that in control, unlike in verification, counterexamples are strategies in a game between system and controller. Second, our scheme can be implemented symbolically and is therefore applicable to infinite-state systems. Third, in the case that the controller has no choices, our scheme subsumes the known algorithms for counterexample-guided *verification*. In particular, we present a symbolic algorithm that employs counterexample-guided abstraction refinement in a uniform way to check satisfaction as well as controllability for all linear-time specifications (LTL or Buchi automata). Our algorithm is game-based and can be applied in all situations where games provide an adequate model, such as supervisory control, hardware and program synthesis and modular verification.

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, 9, 24, 6]. 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. The refinement process is guided by looking at counterexamples: 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 in order to eliminate the spurious counterexample.

*Supported in part by the DARPA SEC grant F33615-C-98-3614, the DARPA PCES grant F33615-00-C-1693, the MARCO GSRC grant 98-DT-660, the NSF grant CCR-9988172, the NSF ITR grant CCR-0085949, and a Microsoft Research Fellowship.

The method of counterexample-guided abstraction refinement has been developed for the *verification* of linear-time properties [9], and some (universal) branching-time properties [10]. It has been applied successfully in both hardware [17] and software verification [6, 19]. We develop the method of counterexample-guided abstraction refinement, for the first time, 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 LTL 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 LTL 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 [18].

In somewhat more detail, our method proceeds as follows. Given a two-Player game structure (player 1 "system" vs. player 2 "controller"), we wish to check if player 2 has a strategy to achieve a given LTL (or ω-regular *e.g.,* buchi automata) winning condition. Solutions to this problem have applications in supervisory control [23], sequential hardware synthesis and program synthesis [8, 7, 22], modular verification [2], receptiveness checking [14], 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 2 to have a winning strategy. We start with a very coarse abstract game structure and refine it iteratively. First, we check if player 2 has a winning strategy in the abstract game; if so, then the concrete system can be controlled; otherwise, we construct an abstract player 1 strategy that wins against all abstract player 2 strategies. Second, we check if the abstract player 1 strategy corresponds to a winning strategy for player 1 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 1 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. In particular, we find finite-state controllers for infinite-state systems, such as hybrid systems, whenever such controllers exist. It should be noted that LTL verification problems are but special cases of LTL control problems, where player 2 (the controller) has no choice of moves. Our method, therefore, includes as a special case counterexample-guided abstraction 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*) [3, 21], our counterexample-guided abstraction refinement also computes fixpoints on the iteration of two region operators, called *Focus* and *Shatter.* The *Focus* operator, which is used to check if an abstract counterexample is genuine or spurious, sharpens the relation between an abstract state and its successors: if there is a player 1 move leading to abstract successors that have no concrete counterparts, then that move is removed. 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, which constructs and refines abstract models and solves verification and control problems on these models, 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, *Pre*-based algorithms terminate, as *e.g.,* in the control of timed automata [21] and they may terminate in more cases).

2 Games and Abstraction

2.1 Two-player Games

Let P be a set of proposition. A *(two-player) game structure* $\mathcal{G} = (V_1, V_2, \Sigma, \Gamma, \delta, \mathcal{P})$ consists of two (possibly infinite) sets V_1 and V_2 of states, a finite set Σ of moves, a function $\Gamma : V_1 \cup V_2 \to 2^{\Sigma}$ mapping states to subsets of moves enabled at the state, a transition relation $\delta \subseteq (V_1 \times \Sigma \times V_2) \cup (V_2 \times \Sigma \times V_1)$, and a labeling function $\mathcal{P} : (V_1 \cup V_2) \to 2^P$ mapping states to sets of propositions. The set V_1 is the set of player 1 states, the (disjoint) set V_2 is the set of player 2 states. Let $V = V_1 \cup V_2$ denote the set of all states. The transition relation relates states $v \in V_1$ and moves in $\Gamma(v_1)$ to states in V_2, and states $v \in V_2$ and moves in $\Gamma(v_2)$ to states in V_1. Intuitively, for $i = 1, 2$, at state $v \in V_i$, player i chooses a move $l \in \Gamma(v)$, and the game proceeds to some state v' satisfying $\delta(v, l, v')$. We require that each state v has an enabled move ($\Gamma(v) \neq \emptyset$). For a move $l \in \Sigma$, let $R_l = \{v \in V \mid l \in \Gamma(v)\}$ denote the set of states in which move l is enabled. We extend the transition relation to sets via the operator Apre : $2^V \times \Sigma \to 2^V$ by defining $\mathrm{Apre}(X, l) = \{v \in V \mid \forall v'.\delta(v, l, v') \Rightarrow v' \in X\}$. We assume P contains a special proposition *init*. For $p \in P$ let $\langle\langle p \rangle\rangle = \{v \mid p \in \mathcal{P}(v)\}$ and $\langle\langle \neg p \rangle\rangle = V \setminus \langle\langle p \rangle\rangle$, the set $\langle\langle init \rangle\rangle$ is the set of *initial states*. We say v satisfies p if $v \in \langle\langle p \rangle\rangle$.

A *source-v_0 run* of the game $\mathcal{G}$ is an infinite sequence $v_0 v_1 \ldots$ of states in V such that for all $j \geq 0$, there is $l_j \in \Gamma(v_j)$ such that $\delta(v_j, l_j, v_{j+1})$. A *strategy of player 1* is a function $f_1 : V^+ \to \Sigma$ such that $f_1(w \cdot v) \in \Gamma(v)$ for every state sequence $w \in V^*$ and every state $v \in V_1$. A *strategy of player 2* is a pair (f_2, n_2) where $f_2 : V^+ \to \Sigma$ and $n_2 : V^+ \times \Sigma \to V_2$. For every state sequence $w \in V^*$, every state $v \in V_2$, and every state $v' \in V_1$, the function f_2 suggests moves to player 2: $f_2(w \cdot v) \in \Gamma(v)$, and the function n_2 resolves the nondeterminism in the move of player 1: $n_2(w \cdot v', l) \in \{u \mid \delta(v', l, u)\}$. Since the transition relation is (in general) nondeterministic, the games are inherently asymmetric: we allow the adversary to resolve the nondeterminism at every stage. Hence our definition of strategies for players 1 and 2 are also asymmetric: a strategy for player 1 looks at the history of the game and suggests a move; a strategy for player 2 suggests moves at player 2 states, and moreover resolves the nondeterminism by picking some successor at player 1 states (given a move of player 1 at that state). This asymmetry is needed (as explained below) in the abstraction of games. Let f_1 be a strategy of player 1 and (f_2, n_2) a strategy of player 2. The *outcome* $\Pi_{f_1,(f_2,n_2)}(v_0)$ from $v_0 \in V$ of strategies f_1 and (f_2, n_2) is a subset of the source-v_0 runs of $\mathcal{G}$: a run $v_0 v_1 v_2 \ldots$ belongs to $\Pi_{f_1,(f_2,n_2)}(v_0)$ if for all $j \geq 0$, we have $v_{j+1} = n_2(v_0 \ldots v_j, f_1(v_0 \ldots v_j))$ if $v_j \in V_1$, and $\delta(v_j, f_2(v_0 \ldots v_j), v_{j+1})$ if $v_j \in V_2$.

2.2 LTL Winning Conditions

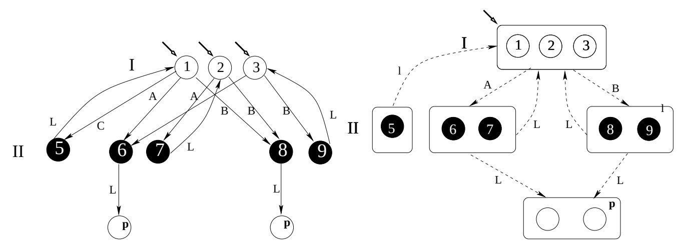
We consider winning conditions expressed by formulas of linear temporal logic (LTL).[1] The *LTL formulas* are generated by the grammar

$$\Psi ::= p \mid \neg\Psi \mid \Psi \vee \Psi \mid \bigcirc\Psi \mid \Psi\,\mathcal{U}\,\Psi$$

where $p \in P$ is a proposition, $\bigcirc$ is the "next" operator, and $\mathcal{U}$ is the "until" operator. Additional constructs such as $\Diamond\Psi = \mathit{true}\,\mathcal{U}\,\Psi$ and $\Box\Psi = \neg\Diamond\neg\Psi$ can be defined in the standard way. A *trace* $\tau : \omega \to 2^P$ is an infinite sequence of sets of propositions. Every LTL formula Ψ has a truth value on each trace, we write $L(\Psi)$ for the set of traces that satisfy Ψ; a formal definition for $L(\Psi)$ may be found in [15]. For a run $\pi = v_0 v_1 \ldots$ of $\mathcal{G}$, we get a trace $\tau_\pi = P_0 P_1 \ldots$ where for all $j > 0$, $P_j = \mathcal{P}(v_j)$ is the set of propositions from P true at v_j. An *LTL game* is a triple $(\mathcal{G}, \langle\langle init \rangle\rangle, \langle 1 \rangle \Psi)$ where $\mathcal{G} = (V_1, V_2, \Sigma, \Gamma, \delta, \mathcal{P})$ is a game structure, $\langle\langle init \rangle\rangle$ denotes the set of initial states, and Ψ is a winning objective for player 1. A run π is winning for player 1 if the corresponding trace τ_π satisfies Ψ. A strategy f_1 is winning for player 1 if for all strategies (f_2, n_2) of player 2, and for all initial states v, all runs in the outcome $\Pi_{f_1,(f_2,n_2)}(v)$ are winning for player 1. Conversely, a strategy (f_2, n_2) is spoiling for player 2 if for all strategies f_1 of player 1, there is an initial v and a run in $\Pi_{f_1,(f_2,n_2)}(v)$ not winning for player 1.

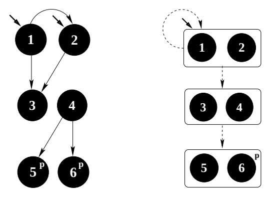
We consider, in particular, *safety games* (and dually *reachability games*) on game structures. A safety game has the objective $\langle 1 \rangle \Box p$. Intuitively, the goal of player 1 is to keep any play starting from any initial state always inside the set of states satisfying p. Formally, a run $\pi = v_0 v_1 \ldots$ is winning for player 1 if for all $j \geq 0$, we have $p \in \mathcal{P}(v_j)$. Π_p denote the set of all runs winning for player 1. The strategy f_1 is *winning for player 1* if for all strategies (f_2, n_2) of player 2 and for all $v \in \langle\langle init \rangle\rangle$, the outcome $\Pi_{f_1,(f_2,n_2)}(v) \subseteq \Pi_p$. Conversely, a strategy (f_2, n_2) is *spoiling for player 2* if for all strategies f_1 of player 1, there is a $v \in \langle\langle init \rangle\rangle$ such that the outcome $\Pi_{f_1,(f_2,n_2)}(v) \cap \Pi_p = \emptyset$. The dual of a safety game is a *reachability game* $\langle 1 \rangle \Diamond p$. Intuitively, the goal of player 1 is to force the play into a state satisfying p. Formally, a run $\pi = v_0 v_1 \ldots$ is winning for player 1 if there exists $j \geq 0$ such that $p \in \mathcal{P}(v_j)$. Note that nondeterministic games are

[1] Our results hold also for winning conditions given by ω-regular languages.



(a) Game (b) Abstraction

Figure 1: Ex-Control



(a) (b)

Figure 2: Ex-Verif

in general not determined: a spoiling strategy of player 2 is not necessarily a winning strategy in the dual game.

In the following, if the game structure and the set of initial moves is understood, we refer to a game only by the winning condition, for example, for a fixed game $\mathcal{G}$ and fixed initial set of states $\langle\!\langle init \rangle\!\rangle$, we refer to the game $(\mathcal{G}, \langle\!\langle init \rangle\!\rangle, \langle 1 \rangle \Psi)$ as $\langle 1 \rangle \Psi$.

Example 1 [**A Safety Game**] As an example of a safety game, consider the game in Figure 1(a). The white states are player 1 states and the black ones are player 2 states. The labels on the edges denote the moves. The objective is $\langle 1 \rangle \Box \neg p$, *i.e.,* the states $\langle\!\langle p \rangle\!\rangle$ are the error states that player 1 seeks to avoid. The player 1 states $1, 2, 3$ are the starting states, *i.e.,* we wish player 1 to win from all those states. Notice that in fact player 1 wins from the states 1, 2, and 3: at state 1, he plays the move C, at state 2, he plays A, and at state 3, he plays B. In each case, the only move L of player 2 brings the game back to the original state. This ensures the game never reaches a state in $\langle\!\langle p \rangle\!\rangle$. ∎

Example 2 [**A Safety Verification Problem**] The standard safety verification (or invariant checking) problem on transition systems is a special case of the safety game where there are only player 2 states. An example is given in Figure 2(a). The starting states are 1,2 and we wish to check that $\Box \neg p$,*i.e.,* that the states 5,6 are never visited. It is easy to see that the system satisfies this criterion. ∎

2.3 Abstractions of Games

It may be computationally infeasible to compute solutions to safety and reachability games on a game structure. Hence we study the abstraction of safety and reachability games on *abstract game structures* [20]. We want the abstraction to be *sound*: *i.e.,* the properties that we establish in the abstract game $\mathcal{G}^\alpha$ should carry over to the actual game, namely, if player 1 wins the safety (or reachability) game on an abstract game, then he wins the corresponding game on the concrete game. To ensure soundness, we restrict the power of player 1 and increase the power of player 2. Therefore we abstract the player 1 abstract states so that *fewer* moves are enabled, and the player 2 abstract states so that *more* moves are enabled.

Definition 1 [**Abstract Game Structures**] Given a game structure $\mathcal{G} = (V_1, V_2, \Sigma, \Gamma, \delta, \mathcal{P})$ with state space $V = V_1 \cup V_2$, an *abstract game structure* $\mathcal{G}^\alpha$ for $\mathcal{G}$ is a tuple $(V_1^\alpha, V_2^\alpha, \Sigma, \Gamma^\alpha, \delta^\alpha, \mathcal{P}^\alpha)$ and a concretization function $[\![\cdot]\!] : V^\alpha \to 2^V$ (where $V^\alpha = V_1^\alpha \cup V_2^\alpha$) such that:

(i) The abstraction preserves the player structure and proposition labels: for each $i \in \{1, 2\}$ we have $\forall v_i^\alpha \in V_i^\alpha . [\![v_i^\alpha]\!] \subseteq V_i$; and for each $v^\alpha \in V^\alpha$, if $v_1, v_2 \in [\![v^\alpha]\!]$ then $\mathcal{P}(v_1) = \mathcal{P}(v_2)$.

(ii) The abstract states "cover" the concrete state space: $\bigcup_{v^\alpha \in V^\alpha} [\![v^\alpha]\!] = V$.

(iii) For a player 1 abstract state $v_1^\alpha \in V_1^\alpha$, the set of enabled moves $\Gamma^\alpha(v_1^\alpha) = \bigcap_{v_1 \in [\![v_1^\alpha]\!]} \Gamma(v)$. Dually for a player 2 abstract state $v_2^\alpha \in V_2^\alpha$, we have $\Gamma^\alpha(v_2^\alpha) = \bigcup_{v_2 \in [\![v_2^\alpha]\!]} \Gamma(v)$.

(iv) The abstract transition relation is $\delta^\alpha \subseteq V^\alpha \times \Sigma \times V^\alpha$ such that $(v^\alpha, l, w^\alpha) \in \delta^\alpha$ iff $l \in \Gamma^\alpha(v^\alpha)$ and $\exists v \in [\![v^\alpha]\!], v' \in [\![w^\alpha]\!].\ (v, l, v') \in \delta$.

(v) The abstract labeling function $\mathcal{P}^\alpha : V^\alpha \to 2^P$ maps an abstract state v^α to $\mathcal{P}(v)$ where $v \in [\![v^\alpha]\!]$. This is well-defined by (i).

∎

Remark 1 [V^α **generates** $\mathcal{G}^\alpha$] (1) Notice that the abstract transition relation δ^α and Γ^α are functions of just the concrete transition relation δ and the abstract state space V^α. Hence, given a game structure $\mathcal{G}$ and the set of abstract states $V^\alpha = V_1^\alpha \cup V_2^\alpha$, we say $\mathcal{G}^\alpha$ *is generated by the game* $\mathcal{G}$ *and* V^α if $\mathcal{G}^\alpha$ is the abstract game for $\mathcal{G}$ defined on the states V^α in Definition 1. (2) For any set $W \subseteq V$ of a game structure $\mathcal{G}$, we denote by $W^\alpha \subseteq V^\alpha$ the set $\{v^\alpha \mid [\![v^\alpha]\!] \cap W \neq \emptyset\}$.

Intuitively, each abstract state represents a set of concrete states of the original game $\mathcal{G}$; we sometimes identify (with abuse of notation) the concretization of an abstract state with the abstract state. The abstract transition relation can be extended to sets of abstract states via the operators Pre^α and Post^α as for concrete games. The soundness of the abstraction ensures that if player 1 can win the abstract game, he can win the concrete game as well; in fact a winning strategy in the abstract game can be directly translated to a winning strategy in the concrete game.

Proposition 1 [**Soundness of Abstraction**] *Let $\mathcal{G}^\alpha$ be an abstract game structure corresponding to a concrete game structure $\mathcal{G}$ over the set of states V. For any LTL objective Ψ, if player 1 wins the LTL game $(\mathcal{G}^\alpha, \{v^\alpha\}, \langle 1\rangle\Psi)$ from a state v^α on the abstract game structure, then he wins the LTL game $(\mathcal{G}, [\![v^\alpha]\!], \langle 1\rangle\Psi)$ on the concrete game $\mathcal{G}$.*

Example 3 [**Abstractions for** Ex-Safety, Ex-Verif] In Figure 1(b) we show one particular abstraction for Ex-Safety. The boxes denote abstract states with the states they represent drawn inside them. The dashed arrows are the abstract transitions. Notice that from the starting player 1 box, the move C is not enabled 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 $\langle\!\langle p\rangle\!\rangle$. In Figure 2(b), we show an abstraction for Ex-Verif. Notice this is exactly the standard existential abstraction for transition systems. ∎

2.4 Solving LTL Games

We now turn our attention to solving LTL games, *i.e.,* determining the states from which player 1 can win. From the predecessor operator Apre, we can define the one step *controllable predecessor* operator $\text{Cpre}_1 : 2^V \to 2^V$, denoting, for a set $X \subseteq V$, the set of states from which player 1 can force the game into X in one step. Player 1 can force the game into X in one step from a state $v_1 \in V_1$ iff in that state he has some enabled move l such that *all* the l-successors of v_1 are in X, and player 1 can force the game into X from an state $v_2 \in V_2$ iff for all moves l enabled at v_2, all l-successors of v_2 are in X. Hence we have:

$$\text{Cpre}_1(X) = \bigcup_{l\in\Sigma} \{v_1 \in V_1 \mid \forall v_2.\delta(v_1, l, v_2) \implies v_2 \in X\} \cup \bigcap_{l\in\Sigma} \{v_2 \in V_2 \mid \forall v_1.\delta(v_2, l, v_1) \implies v_1 \in X\} \quad (1)$$

Note that player 2 resolves the nondeterminism, this is necessary to ensure soundness.

Given a game with an LTL winning condition, we can construct a μ-calculus formula with the Cpre_1 operator that characterizes the set of states from which player 1 wins the game [13]. Moreover, from the fixpoint computation, one can symbolically construct a winning strategy for player 1 [21, 13]. In particular, the solution of reachability and safety games can be constructed by iterating the one step controllable predecessor Cpre_1 until fixpoint. For example, the set of states from which player 1 can control for $\Box p$ is exactly the greatest fixpoint $\nu X.\langle\!\langle p\rangle\!\rangle \wedge \text{Cpre}_1(X)$, so player 1 wins the safety game $(\mathcal{G}, \langle\!\langle init\rangle\!\rangle, \langle 1\rangle\Box p)$ if $\langle\!\langle init\rangle\!\rangle \subseteq (\nu X.\langle\!\langle p\rangle\!\rangle \wedge \text{Cpre}_1(X))$. Similarly, the set of states from which player 1 can control for $\Diamond p$ is exactly the least fixpoint $\mu X.\langle\!\langle p\rangle\!\rangle \vee \text{Cpre}_1(X)$.

Given a game $\mathcal{G}$ we construct and model check its abstraction $\mathcal{G}^\alpha$. The soundness of the abstraction ensures that if player 1 can win the abstract game, then he can win the concrete game as well, moreover, a strategy for player 1 in the abstract game can be used to synthesize (symbolically) a strategy for player 1 in the concrete game [21, 13]. On the other hand, if player 1 cannot win the abstract game, the fixpoint iteration can be used to return an abstract counterexample corresponding to a spoiling strategy of player 2 in the abstract game.

3 Counterexample-Guided Refinement

We now describe what the abstract counterexamples look like and how they may be analyzed. For clarity, we shall first focus on synthesizing controllers for *safety games*; Section 4.2 discusses how a similar analysis works for general LTL (or ω-regular) control objectives. We give an algorithm to decide whether an abstract counterexample corresponds to a player 2 spoiling strategy for the concrete safety game (the counterexample is "genuine"), or if it arises due to the coarseness of the abstraction (the counterexample is "spurious"). In the first case, no controller can be synthesized; in the second, we must refine the abstraction in order to rule out this counterexample (and similar ones). In essence our algorithm amounts to model checking the counterexample to find which of the states corresponding to the abstract set of states represented by the counterexample can actually be a part of a real counterexample. In the sequel, we show the different steps on a fixed safety game $(\mathcal{G}, \langle\langle init \rangle\rangle, \langle 1 \rangle \Box \neg p)$. We write $U = \langle\langle init \rangle\rangle$ and $W = \langle\langle p \rangle\rangle$.

3.1 Abstract Counterexamples

A *counterexample* for a safety game $\Box p$ is a strategy of player 2 that ensures that for every strategy of player 1, some state of $\langle\langle p \rangle\rangle$ is reached eventually. Finite trees form a natural representation of such counterexamples. In fact, since memoryless strategies suffice for a reachability game, such a tree will contain states only once along any path. This is unlike counterexample driven verification of safety properties [9, 24, 6] where the counterexample is just a trace.

In the sequel we use the following notation. By *tree*, we mean a rooted directed finite tree with labels on both nodes and edges. Each edge is labeled with a move from Σ. If $\mathtt{n} \xrightarrow{l} \mathtt{n}'$ is an edge in the tree we say $\mathtt{n}'$ is an l-child of $\mathtt{n}$. Nodes are marked with either an abstract state (denoted by v^α), a concrete state (denoted by v), or a set of concrete states (denoted by r). We write $\mathtt{n} : \mathtt{M}$ for node $\mathtt{n}$ marked with $\mathtt{M}$. A *leaf* is a node with no children.

Definition 2 [Abstract counterexample trees] An abstract counterexample tree T^α for $\mathcal{G}^\alpha$ is a finite tree where each node is labeled by an abstract state such that $\mathtt{n}_1 : v_1^\alpha$ is an l-child of $\mathtt{n} : v^\alpha$ only if $(v^\alpha, l, v_1^\alpha) \in \delta^\alpha$ and:

(i) The root of the tree is labeled by an abstract initial state.

(ii) If node $\mathtt{n} : v^\alpha$ is a leaf, then either $v^\alpha \in V_1^\alpha$ and there is no enabled move ($\Gamma^\alpha(v^\alpha) = \emptyset$), or $[\![v^\alpha]\!] \subseteq \langle\langle p \rangle\rangle$.

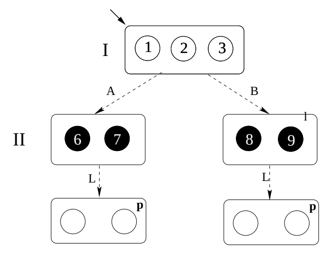
(iii) If node $\mathtt{n} : v^\alpha$ is an internal player 1 node, then *for each* $l \in \Gamma^\alpha(v^\alpha)$, $\mathtt{n}$ has *at least one* l-child.

(iv) If node $\mathtt{n} : v^\alpha$ is an internal player 2 node, then *for some* $l \in \Gamma^\alpha(v^\alpha)$, $\mathtt{n}$ has *at least one* l-child.

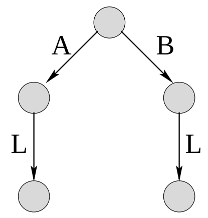
∎

We define a partial order on counterexample trees as follows: $T_1 \subseteq T_2$ iff the rooted tree T_1 is equal to some subgraph of T_2 (with the same root). An abstract counterexample tree T^α is *maximal* (respectively, minimal) for $\mathcal{G}^\alpha$ if there is no counterexample tree T_1^α for $\mathcal{G}^\alpha$ such that $T^\alpha \subsetneq T_1^\alpha$ (respectively, $T^\alpha \supsetneq T_1^\alpha$). The *type* of a counterexample tree T^α is a tree *type*(T^α) that is identical to T^α, except that only the edges are labeled, not the nodes. The set of types of a counterexample tree T^α is denoted *Types*$(T^\alpha) = \{$*type*$(T_1^\alpha) \mid T_1^\alpha$ is a counterexample tree and $T_1^\alpha \subseteq T^\alpha\}$.

An abstract counterexample tree T^α corresponds to a *set* of spoiling strategies for player 2 in the abstract game. At each player 1 state $\mathtt{n} : v^\alpha$, for each enabled move l that player 1 can choose, player 2 can choose at



(a) T^α



(b) $type(T^\alpha)$

Figure 3: Abstract counterexample for Ex-Safety

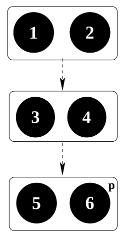


Figure 4: An abstract counterexample τ^α for Ex-Verif

least one successor abstract state from which she wins by subsequently forcing it into W^α, namely, any one of the l-children of $\mathtt{n} : v^\alpha$. Depending on how player 2 chooses to resolve the nondeterminism, we get a set of spoiling strategies for player 2. If player 1 cannot win the safety game $\langle 1 \rangle \Box V^\alpha \setminus W^\alpha$, the model checker produces a maximal counterexample tree corresponding to a spoiling strategy for player 2.

Example 4 [Abstract Counterexamples] Figure 3(a) shows an abstract counterexample T^α for the abstraction of Ex-Safety. Notice that the counterexample for a safety game is an And-Or tree. Figure 3(b) shows the *type* of the given counterexample. Figure 4 shows an abstract counterexample for the abstraction of Ex-Verif. The counterexample for invariant verification is just a trace τ^α. ∎

Note that there may be player 1 abstract states in the abstract game with no abstract successors. To ensure soundness, such states must be considered losing for player 1. Since the original game structure is total, an abstract strategy of player 1 that wins by going to a state with no successors cannot be translated to a concrete strategy; hence any such winning strategy must be removed in the abstract game. Consider for example a game with three states s, t, u, where the first two are player 1 states. At s, player 1 can play only the move A and go to u, and at t he can play the move B and go to u. Now consider the safety game $\langle 1 \rangle \Box \{s, t\}$ with starting states $\{s, t\}$. Clearly player 1 loses. Consider an abstraction made of the abstract states $\{s, t\}, \{u\}$. In the abstract state $\{s, t\}$ no moves are enabled, so the abstract game never goes out of $\{s, t\}$. However, state $\{s, t\}$ is still losing for player 1. Note that since the original game structure is total, there cannot be abstract player 2 states with no abstract successors. Therefore, we assume (by splitting abstract states if necessary) that each leaf state of a given abstract counterexample has at least one enabled move. Thus, condition (ii) in Definition 2 becomes: (ii)' if node $\mathtt{n} : v^\alpha$ is a leaf, then $[\![v^\alpha]\!] \subseteq \langle\!\langle p \rangle\!\rangle$. This assumption, while not technically necessary, simplifies the presentation.

3.2 Concretizing Abstract Counterexamples

An *abstract counterexample* may not be realizable in the concrete game, *i.e.,* even though player 2 has a strategy to win the abstract game, it may be the case that that strategy does not correspond to a winning strategy for player 2 in the concrete game.

Definition 3 [Concrete Counterexamples] A *concrete counterexample tree* T is a finite tree where each node is labeled by a concrete state such that:

(i) the root is labeled with an initial state,

(ii) if $(\mathtt{n} : v_\mathtt{n}) \xrightarrow{l} (\mathtt{n}' : v_{\mathtt{n}'}) \in T$ then $(v_\mathtt{n}, l, v_{\mathtt{n}'}) \in \delta$, and

(iii) for each player 1 internal node $\mathtt{n}$ in T, if the children of $\mathtt{n}$ are labeled by moves $C(\mathtt{n})$ then $\Gamma(\mathtt{n}) = C(\mathtt{n})$, *i.e.,* exactly those moves are enabled at v_n, and

(iv) for each leaf node n in T, $v_n \in W$.

An abstract counterexample T^α is realized by the concrete counterexample T if $\mathit{type}(T) = \mathit{type}(T^\alpha)$ and each node $\mathtt{n}$ that was marked with abstract state v^α in T^α is marked in T with a *single* state $v_n \in [\![v^\alpha]\!]$. An abstract counterexample tree T^α is *genuine* if there is some $T_1^\alpha \subseteq T^\alpha$ such that T_1^α is realized by a concrete counterexample; it is *spurious* otherwise. ∎

From the definition it follows that a counterexample tree is spurious iff it does not correspond to a winning strategy for player 2. Given an abstract counterexample tree, the counterexample analysis procedure must determine if the counterexample is genuine. We now give a nondeterministic closure procedure on the abstract counterexample tree that characterizes precisely if the tree is a genuine counterexample. We then resolve the nondeterminism to provide a bottom-up tree-marking algorithm linear in the size of the counterexample tree.

For the abstract counterexample tree T^α, the basic step in counterexample analysis is the Focus operation $\text{Focus}_{T^\alpha} : (T^\alpha \times 2^V) \to 2^V$, that takes a node $\mathtt{n}$ in T^α and a set of concrete states, and returns a subset of the concrete states. Let $C(\mathtt{n})$ be the labels on edges leaving $\mathtt{n} \in T^\alpha$, and let $\mathtt{n}_{l,i} : r_{l,i}$ be the various l-children of $\mathtt{n}$ (indexed by i), where each of the $r_{l,i}$ denotes a set of concrete states. Define the operation Focus_{T^α} as:

$$\text{Focus}_{T^\alpha}(\mathtt{n}, r) = \begin{cases} r & \text{if } \mathtt{n} \text{ is a leaf node} \\ r \,\cap\, \Big(\bigcap_{l\in C(\mathtt{n})} (\cup_{n_{l,i}} \text{Apre}(r_{l,i}, l))\Big) \,\cap\, \neg(\bigcup_{l \notin C(\mathtt{n})} R_l) & \text{if } \mathtt{n} \text{ is a player 1 node,} \\ r \,\cap\, \Big(\bigcup_{l\in C(\mathtt{n})} (\cup_{n_{l,i}} \text{Apre}(r_{l,i}, l))\Big) & \text{if } \mathtt{n} \text{ is a player 2 node.} \end{cases} \tag{2}$$

Intuitively, the Focus operator does the following. At each point, each of the nodes of the tree will be marked by those (concrete) states that we know are an upper approximation of the the set of states that can actually be a part of a concrete counterexample of the type of the given one. One step of the Focus operator further sharpens this set by finding exactly which of the states in the present overapproximation can actually have successors which can lead to counterexamples. Thus the fixpoint of this operator contains exactly those states that can be part of the counterexample. For any leaf node (since we assume every leaf node has at least one enabled move), all states are in $\langle\!\langle p \rangle\!\rangle$, and therefore can actually be in a counterexample. For player 1 internal nodes, the only states that can be part of a counterexample are those (i) where the *only* enabled moves are the moves labeling the child-edges on the tree and moreover (ii) where *all* their l-successors are in fact such that from them player 2 has a spoiling strategy corresponding to the given tree, *i.e.,* those states that lie in the intersection of the Apres of the overapproximations presently labeling the children. Similarly for player 2 nodes, only those nodes can actually be a part of a counterexample, which have at least one successor from which subsequently player 2 can spoil, which is exactly the union of the Apres of the overapproximations of the child nodes.

Algorithm 1 AnalyzeCounterex(T^α)

```
INPUT: An abstract counterexample tree $T^\alpha$, with root root.
OUTPUT: Whether the abstract counterexample $T^\alpha$ is SPURIOUS or GENUINE
for each $\mathtt{n} : v^\alpha \in T^\alpha$ do
  Relabel n with $[\![v^\alpha]\!]$
while there is some node $\mathtt{n} : r$ with $r \not\equiv \text{Focus}_{T^\alpha}(\mathtt{n}, r)$ do
  pick some node $\mathtt{n} : r$ to focus
  replace $\mathtt{n} : r$ with $\mathtt{n} : \text{Focus}_{T^\alpha}(\mathtt{n}, r)$ in the tree
  if root : $r$ with $r = \emptyset$ then
    return SPURIOUS
return GENUINE
```

The counterexample analysis procedure AnalyzeCounterex iterates Focus on nodes until there is no change, that is, it computes the greatest fixpoint of the Focus operator on the abstract tree. Notice that in fact the procedure AnalyzeCounterex is *model checking the counterexample* tree (T^α) to find if it is genuine or not. The greatest fixpoint that it computes is exactly the set of states that can be a part of a concrete counterexample,

Algorithm 2 Focus*($\mathtt{n} : v^\alpha, T^\alpha$)

```
INPUT: An abstract counterex. tree $T^\alpha$ and a node $\mathtt{n} \in T^\alpha$ marked by the abstract state $v^\alpha$ .
OUTPUT: The region that is the greatest fixpoint of Focus for node $\mathtt{n}$.
if n is a leaf node then
  return $[\![v^\alpha]\!]$
else if n is a player 1 node then
  return $[\![v^\alpha]\!] \cap \left(\bigcap_{l \in C(\mathtt{n})} (\cup_i \mathrm{Apre}(\mathrm{Focus}^*(n_{l,i} : v^\alpha_{l,i}, T^\alpha), l))\right) \cap \neg(\bigcup_{l \notin C(\mathtt{n})} R_l)$
else if n is a player 2 node then
  return $[\![v^\alpha]\!] \cap \left(\bigcup_{l \in C(\mathtt{n})} (\cup_i \mathrm{Apre}(\mathrm{Focus}^*(n_{l,i} : v^\alpha_{l,i}, T^\alpha), l))\right)$
```

and thus if at some point that set becomes empty it means that there is no concrete counterexample that realizes T^α.

The procedure AnalyzeCounterex is nondeterministic in its choice of nodes to refine. One can resolve the nondeterminism of the **while** loop by a bottom-up strategy starting from the leaves: each node is focused after all of its children. This ensures that after a node is focused for the first time, for all states v in the focus, there is a concrete counterexample tree rooted at v. The algorithm Focus* computes this fixpoint in time linear in the size of the counterexample tree. Note that AnalyzeCounterex(T^α) returns SPURIOUS iff Focus*($\mathtt{root}, T^\alpha$) is empty. In the sequel Focus*($\mathtt{n}$) refers to the set of states computed by the above function, namely Focus*($\mathtt{n} : v^\alpha, T^\alpha$), which is exactly the same as the fixpoint of the Focus operator as computed by the **while** loop of AnalyzeCounterex. We drop the parameters v^α, T^α where they are clear from the context. The correctness and complexity of the counterexample analysis routine is stated by the following proposition.

Proposition 2 [**Correctness of** AnalyzeCounterex] *The counterexample T^α is spurious iff the procedure* AnalyzeCounterex(T^α) *returns* SPURIOUS. *Checking if a counterexample tree is spurious can be done in time linear in the size of the counterexample tree.*

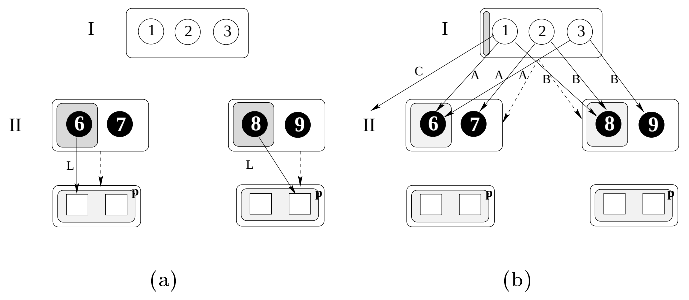


Figure 5: Focusing the nodes of T^α.

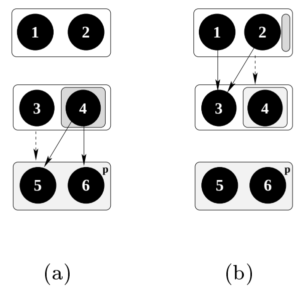


Figure 6: Focusing nodes of τ^α.

Example 5 [**Counterexample Analysis:** EX-SAFETY] In Figure 5 we see the result of running AnalyzeCounterex (or Focus*) on the counterexample T^α. The lightly shaded parts of the boxes denote the states which may still be a part of a counterexample *i.e.,* the states that we take the Pres of when computing the Focus. The dashed arrows indicate abstract transitions and the solid arrows the concrete transitions occurring between the respective concrete states. In Figure 5(a) we see the player 2 nodes getting focused. Note that all the states in the leaf node are error states (satisfying p) hence they are all in the lightly shaded box. Only 6 (and respectively 8) can go to the error region from the two abstract states hence only they are in the focused region. In Figure 5(b) we see the effect of doing a subsequent Focus on the root node. None of the states in the root node are such that they can play *only* the moves A, B and subsequently go to states from which player 2 can subsequently spoil. Hence neither of those states can serve as the root of a concrete counterexample tree of the same type as the abstract counterexample. Thus the shaded boxes in Figure 5(b) show also the fixpoint of Focus, we see that Focus* is empty and we can conclude the counterexample is spurious. ∎

Example 6 [Counterexample Analysis: Ex-Verif**]** In the Figure 6 we see the result of running AnalyzeCounterex (or Focus*) on the counterexample τ^α. In Figure 6(a) we see the effect of doing a Focus on the second abstract state in τ^α. All the concrete states corresponding to the last abstract state are error states (*i.e.,* satisfy p) hence they are all shaded. Only state 4 can go to one of the error states (*i.e.,* lies in the Pre of the error states) hence it is the only state in the focused region of that node. In Figure 6(b) we see the second application of Focus, this time to the root of the trace — none of the states 1,2 in the root go to 4 (which is the only candidate state for the second abstract state) hence the focused region is empty, and the counterexample spurious. The fixpoint of Focus is shown by the shaded portions in Figure 6(b). ∎

3.3 Counterexample-Guided Refinement

If the counterexample tree returned by the model checker is spurious, we can conclude that the abstraction is too coarse to find a winning strategy for player 1, and hence the abstraction must be refined, in order to rule out this particular spoiling strategy of player 2.

The refinement procedure uses the following observation: since the focus of the root is empty we can conclude that there is no way to get a concrete subtree corresponding to the given counterexample tree (or any subset thereof), using *any* of the states corresponding to the abstract state of the node. We shall split the abstract states labeling the nodes of the tree so as to rule out all possible counterexamples whose type is in *Types*(T^α). The splitting uses a Shatter operation that is dual to the Focus operation used in checking for genuine counterexamples. The Shatter operator takes an abstract state v^α and returns a set of abstract states which will replace v^α in the refined abstraction. This is how we refine the abstract state space. Recall that for a node $\mathtt{n} : v^\alpha$ in the abstract counterexample tree, Focus$^*(\mathtt{n}, v^\alpha)$ represents the set of concrete states that can actually be in some concrete counterexample.

Intuitively, the Shatter operation must break up an abstract state v^α into a "good" part Focus$^*(\mathtt{n}, v^\alpha)$ that can actually lead to an error, and its complement $v^\alpha \setminus$Focus$^*(\mathtt{n}, v^\alpha)$ (the "bad" part). Second, the "bad" part must be further broken down so that the refined abstraction cannot have a similar counterexample using the "bad" partitions. Thus, the bad parts must be broken up into fragments, such that it is clear that each fragment cannot lead to the given counterexample.

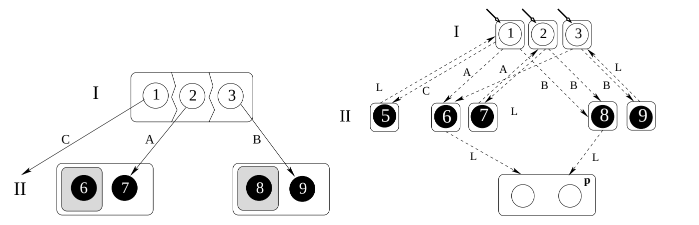
For a player 1 node, a part is bad because every concrete state in it either (1) has some other move enabled that is not in the abstract counterexample, or (2) has some successor from which player 2's given strategy fails *i.e.,* has a successor in a "bad" block. Thus we further split the bad part into little pieces where each piece has some other move enabled or has successors only in some bad block. For a player 2 node, the bad part is the set of states from which there is no l-successor in a "good" block, from such states, player 2 cannot spoil using the given counterexample strategy. It can be checked that player 2 bad blocks need not be further broken up.

We must of course update the operation Cpre$_1^\alpha$ for each of the refined abstract state space, but those are defined by the abstract transition relation, which in turn is defined by the concrete transition relation and the abstract state space.[2]

The Shatter operator takes a node $\mathtt{n} : v^\alpha$ of the abstract counterexample tree and returns a set of subsets of $[\![v^\alpha]\!]$. One of those subsets is the "good" part from which player 2 does indeed have a spoiling strategy given by the current counterexample tree. The other parts are the "bad" parts from which the present spoiling strategy fails. Each part is small enough that it is clear why the present strategy fails when each of the bad parts is an abstract state.

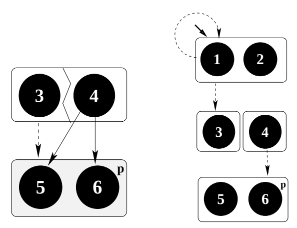
For the node $\mathtt{n} : v^\alpha$ in T^α denote by r^+ the set Focus$^*(\mathtt{n} : v^\alpha)$ and by r^- the set $[\![v^\alpha]\!] \setminus r^+$. Recall that $C(\mathtt{n})$ is the set of labels on edges leaving $\mathtt{n} \in T^\alpha$, and $\mathtt{n}_{l,i} : r_{l,i}$ (for index i) are the various l-children of $\mathtt{n}$,

[2]Recall that the abstract game is generated by its state space and the concrete game.



(a) Shatter (b) Refinement

Figure 7: Refinement for Ex-Safety



(a) Shatter (b) Refinement

Figure 8: Refinement for Ex-Verif

where $r_{l,i} = [\![v^\alpha_{l,i}]\!]$. Let $r^+_{l,i} = \text{Focus}^*(\mathtt{n}_{l,i})$ and $r^-_{l,i} = r_{l,i} \setminus r^+_{l,i}$.

$$\text{Shatter}_{T^\alpha}(\mathtt{n} : v^\alpha) = \begin{cases} \{r^+\} \cup \{r^- \cap R_l \mid l \notin C(\mathtt{n})\} \cup \{r^- \cap (\cup_i \text{Apre}(r^-_{l,i}, l)) \mid l \in C(\mathtt{n})\} & \text{if } \mathtt{n} \text{ is a player 1 node} \\ \{r^+, r^-\} & \text{if } \mathtt{n} \text{ is a player 2 node} \end{cases} \quad (3)$$

Example 7 [Shatter: Ex-Safety] In Figure 7 is shown the effect of the Shatter operator on the root node, and the final refined game in which the counterexample is removed. For the other nodes the shatter is trivial, namely into r^+, r^-. We break up the states of the root node into (i) those that can execute some different move, which we then break into groups such that each state in the group can play some new move here only the state 1 which can play the new move C and (ii) those that can execute only A or B but can go to a state from which player 2's strategy fails (escape) *i.e.,* can play A or B and land in a state not in the fixpoint of Focus, for this latter split we further split into those that can play A and escape namely the state 2 and those that can play B and escape, namely the state 3. Notice that *any* abstraction in which any two of $1, 2, 3$ are together admits a counterexample of this type. ∎

Example 8 [Shatter: Ex-Verif] In Figure 8 is shown the effect of the Shatter operator on the nodes of the abstract states of τ^α. It is easy to check that only the second state gets shattered into $\{3\}, \{4\}$, which are the r^-, r^+ respectively of that abstract state, as the fixpoint of Focus for that state is $\{4\}$. The same figure shows the refined transition system. Again, it is trivial to see that there is no counterexample in the refined system. ∎

For any abstract counterexample tree T^α, and any node $\mathtt{n} : v^\alpha$ in T^α, it can be shown that the union of the concretizations of the shattered states of v^α is equal to the concretization of r; *i.e.,* if $R = \text{Shatter}_{T^\alpha}(\mathtt{n} : v^\alpha)$ then we have: $[\![r]\!] = \bigcup_{r' \in R} [\![r']\!]$ – we omit the proof for brevity.

The refinement step then is as follows: given the old abstraction structure $\mathcal{G}^\alpha$, we replace each "shattered" state $v^\alpha \in V^\alpha$ with the smaller states in $\text{Shatter}_{T^\alpha}(\mathtt{n}, [\![v^\alpha]\!])$, to get a *finer* abstraction structure, and recompute δ^α and Γ^α for the refined state space.

Definition 4 [Refinement] For any abstraction $\mathcal{G}^\alpha$, and a spurious abstract counterexample T^α for $\mathcal{G}^\alpha$, define the *refined game* $\text{Refine}(\mathcal{G}, \mathcal{G}^\alpha, T^\alpha)$ as the abstract game generated by $\mathcal{G}$ and the abstract state space $V^\alpha \cup V'_{T^\alpha} \setminus V_{T^\alpha}$, where $V_{T^\alpha} = \{v^\alpha \mid (\mathtt{n} : v^\alpha) \in T^\alpha\}$ and $V'_{T^\alpha} = \{\text{Shatter}(\mathtt{n}, v^\alpha) \mid (\mathtt{n} : v^\alpha) \in T^\alpha\}$. ∎

The following proposition states that the spurious counterexample is ruled out in the refined abstraction. Given a set of types (of counterexamples) $\mathcal{T}$ we say that abstract game $\mathcal{G}^\alpha$ has a counterexample in $\mathcal{T}$ iff there is some T^α for $\mathcal{G}^\alpha$ such that $\textit{type}(T^\alpha) \in \mathcal{T}$.

Proposition 3 [Shatter removes counterexamples] *For a game $\mathcal{G}$ and every spurious counterexample T^α on $\mathcal{G}^\alpha$, the refined abstraction Refine$(\mathcal{G}, \mathcal{G}^\alpha, T^\alpha)$ has no counterexample in Types(T^α). Moreover, no further refinement of Refine$(\mathcal{G}, \mathcal{G}^\alpha, T^\alpha)$ has a counterexample in Types(T^α).*

Algorithm 3 AnalyzeRefineCounterex($\mathcal{G}$,T^α)

```
**INPUT:** An abstract counterexample tree $T^\alpha$, with root root.
**OUTPUT:** If the counterex. is spurious then a refined game not containing the counterex. else reports that the counterex. is genuine
if AnalyzeCounterex($T^\alpha$) = SPURIOUS then
  $\mathcal{G}^{\alpha\prime} :=$ REFINE($\mathcal{G}, \mathcal{G}^\alpha, T^\alpha$)
  return (SPURIOUS, $\mathcal{G}^{\alpha\prime}$)
else
  return GENUINE
```

Algorithm 4 CGSafetyControl($\mathcal{G}, r_0, \Psi$)

```
**INPUT:** A game structure $\mathcal{G}$, an initial set of states $r_0$, a safety objective $\Psi = \Box\neg p$.
**OUTPUT:** A synthesized controller exhibited by control strategy $T^\alpha$ or no controller possible exhibited by adversary strategy $T^\alpha$
$\mathcal{G}^\alpha :=$ InitialAbstraction($\mathcal{G}, r_0, \Psi$)
repeat
  ($winner$, $T^\alpha$) := ModelCheck($\mathcal{G}^\alpha, r_0, \Psi$)
  if $winner = 2$ then
    if AnalyzeRefineCounterex($\mathcal{G}$,$T^\alpha$) = ($Spurious$, $\mathcal{G}^{\alpha\prime}$) then
      $\mathcal{G}^\alpha := \mathcal{G}^{\alpha\prime}$
      $winner := \perp$
until $winner \neq \perp$
if $winner = 1$ then
  return SYNTHESIZED CONTROLLER,$T^\alpha$
else
  return NO CONTROLLER POSSIBLE,$T^\alpha$
```

Intuitively, the above proposition states that in the refined abstraction there is no counterexample that has the same type as a spurious counterexample in the coarser abstraction, *i.e., all* the spurious counterexamples in the coarser abstraction are removed in the refined abstraction (and refinements thereof). We then continue the algorithm with the new set of abstract states, by repeating the model checking to search for a winning strategy for player 1 in the refined abstraction.

We need not actually shatter all the nodes in the counterexample tree. In particular if the counterexample tree is minimal, we can refine only the subtree rooted at *any* node `n` : r such that Focus$^*(n, r) = \emptyset$. The above procedure does not shatter the *fewest* nodes in order to eschew the counterexamples, as we might want, but we describe this for brevity. Indeed, the problem of finding the "best" explanation and refinement of a spurious counterexample is an interesting problem in its own right.

4 Counterexample-Guided Controller Synthesis

4.1 Safety Control

Our algorithm for safety control generalizes the "abstract-check-refine" loop described by [5, 9, 24, 6]. Given a game structure $\mathcal{G}$, a set of initial states $r_0 = \langle\langle init\rangle\rangle$, and a proposition p with $W = \langle\langle p\rangle\rangle$, wish to solve the safety game $(\mathcal{G}, r_0, \langle 1\rangle\Box\neg p)$, *i.e.,* we wish to find if player 1 wins from all initial states. Informally, the algorithm is as follows.

Step 1 ("abstraction") We first construct an initial abstract game. One such abstraction could be the trivial abstraction where all player i states are partitioned according to the labeling of propositions, *i.e.,* into those satisfying both p and *init*, those satisfying only p, those satisfying only *init*, and those satisfying neither. Recall that the abstract transition relation is generated by the abstract state space and the concrete game.

Step 2 ("model checking") Next, we model check the abstract game to find if player 1 can keep the game inside the desired region $V^\alpha \setminus W^\alpha$ in the abstract game, starting at all of the abstract states r_0^α. If player 1 can win the abstract safety game from the states r_0^α, then the model checker gives us a winning strategy on the abstract game, from which a winning strategy in the concrete game can be easily constructed [13]. If not, *i.e.,* if player 2 has a strategy to reach W^α no matter what player 1 does, then the model checker produces an abstract counterexample symbolically [11]. Note that as the abstract state space is finite, the model checking is guaranteed to terminate.

Step 3 ("counterexample-driven refinement") If model checking returns an abstract counterexample, we analyze this counterexample strategy to see if it is genuine. If it is genuine, no controller can be synthesized. If instead the counterexample is spurious, then we have to *refine* the abstraction so that this counterexample (and similar ones) do not arise on subsequent model checking runs.

Goto Step 2. ("loop") We then repeat the process with the refined abstraction, until either we get a player 1 winning strategy and hence a controller by which player 1 wins the safety game, or we get a genuine counterexample at which point we know that there is no controller possible.

We summarize the counterexample driven control procedure in Algorithm 4. The algorithm $\mathsf{ModelCheck}$ returns a pair $(1, \text{strategy})$ if player 1 can win the game where strategy is his winning strategy, or it returns $(2, T^\alpha)$ when player 2 has a spoiling strategy, where T^α is an abstract counterexample for the safety game for $\mathcal{G}^\alpha$. The function $\mathsf{InitialAbstraction}$ just returns the trivial abstraction for the game that respects the propositional labeling. From the soundness of abstract interpretation, we get the soundness of the algorithm.

Theorem 1 [Correctness] *For any initial region r_0, safety objective $\langle 1 \rangle \Box \neg p$, and for any terminating execution of Algorithm $\mathsf{CGSafetyControl}(\mathcal{G}, r_0, \Box\neg p)$, we have:*

(i) If $\mathsf{CGSafetyControl}(\mathcal{G}, r_0, \Box\neg p)$ returns an error tree, then there is a state $v_0 \in [\![r_0]\!]$ such that player 2 has a spoiling strategy for the safety game $(\mathcal{G}, \{v_0\}, \langle 1 \rangle \Box \neg p)$.

(ii) Otherwise, $\mathsf{CGSafetyControl}(\mathcal{G}, r_0, \Box\neg p)$ returns a set of states r that satisfies both $r_0 \subseteq r$ and player 1 wins the safety game $(\mathcal{G}, r, \langle 1 \rangle \Box \neg p)$.

In general, Algorithm $\mathsf{CGSafetyControl}$ will 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 [4, 13]. As in [13], we note that in the course of Algorithm $\mathsf{CGSafetyControl}$, the abstract state space always consists of blocks of the alternating bisimilarity relation. Hence, if the game has an alternating bisimilarity relation of finite index, then termination is guaranteed.

4.2 LTL Control

We now generalize the counterexample guided safety control procedure to LTL games.

First, the procedure for solving games must implement a symbolic model checker for LTL games: it should compute the set of winning states of player 1 for an LTL objective, or return a counterexample strategy for player 2. Notice that counterexamples for LTL control (*i.e.,* strategies for player 2) are directed graphs (rather than trees). So we must generalize abstract counterexample trees to *abstract counterexample graphs*, which are rooted, directed graphs, with nodes labeled by abstract states, satisfying conditions (i), (iii), (iv) of Definition 2. The operators *type* and *Types* and the partial order $\subseteq$ are trivially extended to abstract counterexample graphs.

Second, for analyzing counterexamples, we have to generalize the Focus and Shatter operators by considering successors of a node in the graph.

However, for general graphs we cannot apply a bottom-up marking strategy: in the presence of cycles, the fixpoint computation may require focusing a node several times before the fixpoint is reached. This

[1]Shatter defined as before except $r^+ = \text{Focus}_{\mathcal{G}^\alpha}(\mathtt{n}, r)$, $r^- = r \setminus r^+$, and $r^-_{l,i} = F(\mathtt{n}_{l,i}) \setminus r_{l,i}$.

Algorithm 5 AnalyzeRefineLTLCounterex($\mathcal{G}, G^\alpha$)

```
INPUT: An abstract counterexample graph $G^\alpha$, with root root.
OUTPUT: If the counterex. is spurious then a refined game not containing the counterex. else reports that the counterex. is genuine
define a map $F : \mathtt{n} \mapsto v^\alpha$ that maps nodes $\mathtt{n} \in G^\alpha$ to the abstract state labeling them
for each $\mathtt{n} : v^\alpha \in G^\alpha$ do
  Relabel n with $[\![v^\alpha]\!]$
$W^\alpha := V^\alpha$                    {$W^\alpha$ will be the abstract state space of the refinement}
while there is some node $\mathtt{n} : r$ with $r \not\equiv \mathrm{Focus}_{G^\alpha}(\mathtt{n}, r)$ do
  pick some node $\mathtt{n} : r$ to focus
  replace $\mathtt{n} : r$ with $\mathtt{n} : \mathrm{Focus}_{G^\alpha}(\mathtt{n}, r)$ in the tree
  $W^\alpha := W^\alpha \setminus \{r\} \cup \mathrm{Shatter}(\mathtt{n} : r)$ 1
  if root : $r$ with $r = \emptyset$ then
    return (SPURIOUS, Game generated by $(\mathcal{G},\ W^\alpha)$)
return GENUINE
```

also implies that a node may be shattered more than once: once for each focus operation. We therefore generalize the AnalyzeCounterex and AnalyzeRefineCounterex procedures of Section 3 to the procedure AnalyzeRefineLTLCounterex that simultaneously shatters nodes while computing the fixpoint of the Focus operation. The procedure AnalyzeRefineLTLCounterex takes an abstract counterexample graph as input and returns "Genuine" if the counterexample can be realized in the concrete game, or "Spurious" if the counterexample is spurious, along with a refinement of the abstract state space that rules out this counterexample. The Shatter procedure of Section 3 can be seen as a special case, when each node is focused (and shattered) exactly once.

With these modifications, we can prove the analogues of Propositions 2 and 3.

Proposition 4 [Correctness of AnalyzeRefineLTLCounterex] *Given an abstract counterexample graph G^α if procedure* AnalyzeRefineLTLCounterex($\mathcal{G}, G^\alpha$) *terminates then:*

1. *If the procedure returns* GENUINE *then player 2 has a spoiling strategy in the game, i.e., no controller can be synthesized.*

2. *If the procedure returns* (SPURIOUS, $\mathcal{G}^{\alpha'}$) *then the game $\mathcal{G}^{\alpha'}$ has no counterexample in Types(G^α), and no further refinement of $\mathcal{G}^{\alpha'}$ has a counterexample in Types(G^α).*

Moreover, if the state space is finite, then the procedure AnalyzeRefineLTLCounterex *will terminate.*

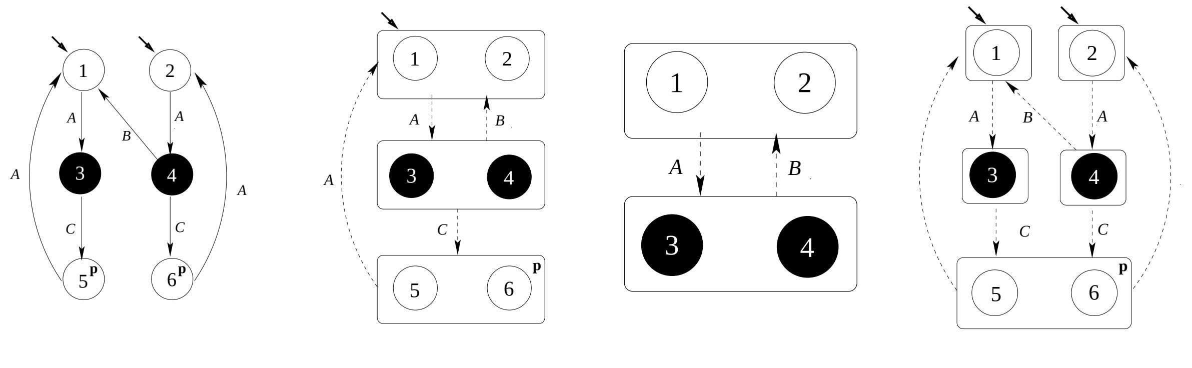
From the procedure CGSafetyControl from Section 3 we obtain an algorithm CGLTLControl for counterexample-guided LTL control, by simply replacing the procedure AnalyzeRefineCounterex with the procedure AnalyzeRefineLTLCounterex and replacing the safety objective Ψ with any arbitrary LTL objective.

Theorem 2 *For any game $\mathcal{G}$, initial region r_0, LTL objective $\langle 1 \rangle \Psi$, and for any terminating execution of Algorithm* CGLTLControl($\mathcal{G}, r_0, \Psi$), *we have:*

(i) If CGLTLControl($\mathcal{G}, r_0, \Psi$) *returns an error, then there is a state $v_0 \in [\![r_0]\!]$ such that player 2 has a spoiling strategy in the game $(\mathcal{G}, \{v_0\}, \langle 1 \rangle \Psi)$.*

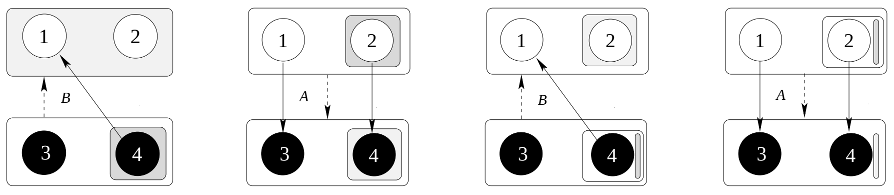
(ii) Otherwise, CGLTLControl($\mathcal{G}, r_0, \Psi$) *returns a region r that satisfies both $r_0 \subseteq r$ and player 1 wins the game $(\mathcal{G}, r, \langle 1 \rangle \Psi)$.*

Example 9 [LTL Control] In Figure 9(a) we show an example of an LTL game. We wish to check whether: $\langle 1 \rangle \Box \Diamond p$, *i.e.,* that player 1 can force the game into a p-state infinitely often. Figure 9(b) shows an abstraction for such a game, and Figure 9(c) shows the result of solving the game on that abstraction: an abstract counterexample where player 2 forces a loop not containing a p state. We show how the



(a) Game (b) Abstraction (c) Counterex. $\mathcal{G}^{\alpha}$ (d) Refinement

Figure 9: Ex-Control



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

Figure 10: AnalyzeRefineLTLCounterex on G^{α}

counterexample is analyzed and discovered to be spurious in Figure 10. In the first figure we see the effect of doing a Focus on the second (lower) node of G^α. As only the state $\{4\}$ has a move that lands it in $\{1,2\}$ (the region of the B-child of the node) the focused region for the node is $\{4\}$, Also, the abstract state is shattered into two states $\{4\}$ and $\{3\}$ (respectively r^+, r^-). Next, in Figure 10(b) we see the effect of doing a Focus on the upper node of G^α. Only the state 2 has an A successor in the focused region of the lower node, hence the focused region becomes $\{2\}$, and the upper node gets shattered into $\{1\}$ and $\{2\}$. In Figure 10(c) we again do a Focus on the lower node. Since no state (in particular 4) has a B-move to the focused region of the upper node, the focused region of the lower node becomes empty. In the next Figure 10(d) we see that when we do a last Focus on the upper node, it becomes empty as well. In Figure 9(d) we see the refined abstraction for the game; it is easy to see that player 2 has no spoiling strategy. ∎

In [10], the authors consider counterexample based model checking for ACTL formulas. In that case (and in fact, for some more expressive logics considered in [10]), the counterexample graphs are tree-like, and our algorithm for analyzing counterexamples and refining the abstraction specializes to theirs; in fact since the counterexamples are models of ECTL formulas, the counterexample graph in that case contains only player 2 nodes. Similarly, in model checking the μ-calculus, one can reduce the model checking question to solving a parity game [16]. Hence, the above method provides a counterexample driven model checking procedure for the μ-calculus.

Finally, we note that the operators Focus and Shatter can be defined using the operator Apre and boolean operations. Hence, all the counterexample driven algorithms in this paper can be implemented completely symbolically using a symbolic representation of game structures [13].

References

[1] K. Altisen, G. Gössler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *RTSS 99: IEEE Real-Time Systems Symposium*, pages 154–163. IEEE Computer Society Press, 1999.

[2] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In J.C.M. Baeten and S. Mauw, editors, *CONCUR 99: Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 82–97. Springer-Verlag, 1999.

[3] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[4] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.

[5] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[6] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: 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 *Proceedings of the 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 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[10] E. M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *LICS 02: Logic in Computer Science*, pages ???–??? IEEE Press, 2002.

[11] E.M. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC 95: Proceedings of the 32nd Design Automation Conference*, pages 427–432. IEEE Computer Society Press, 1995.

[12] L. de Alfaro and T.A. Henzinger. Interface automata. In *Foundations of Software Engineering*, pages ??–?? ACM Press, 2001.

[13] L. de Alfaro, T.A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite state games. In *CONCUR: Concurrency Theory*, pages ??–?? LNCS ????, Springer-Verlag, 2001.

[14] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* The MIT Press, 1989.

[15] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[16] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model checking for fragments of μ-calculus. In *CAV 93: Computer-aided Verification*, LNCS 697, pages 385–396. Springer-Verlag, 1993.

[17] S.G. Govindaraju and D.L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *ICCAD 00: International Conference on Computer-Aided Design*, pages ??–?? ACM Press, 2000.

[18] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proceedings of the 14th Annual Symposium on Theory of Computing*, pages 60–65. ACM Press, 1982.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[20] T.A. Henzinger, R. Majumdar, F.Y.C. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In J. Palsberg, editor, *SAS 00: Static Analysis*, Lecture Notes in Computer Science 1824, pages 220–239. Springer-Verlag, 2000.

[21] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95: Theoretical Aspects of Computer Science*, LNCS 900, pages 229–242. Springer-Verlag, 1995.

[22] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th Annual Symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.

[23] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[24] H. Saidi. Model checking guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*, pages 377–396. LNCS 1824, Springer-Verlag, 2000.