Producing Short Counterexamples Using "Crucial Events"

Sujatha Kashyap[1] and Vijay K. Garg[2,⋆]

[1] ECE Department, University of Texas at Austin, Austin TX 78712, USA
kashyap@ece.utexas.edu
[2] IBM India Research Lab., New Delhi, India
vijgarg1@in.ibm.com

Abstract. Ideally, a model checking tool should successfully tackle state space explosion for complete system validation, while providing short counterexamples when an error exists. Techniques such as partial order (p.o.) reduction [1,2] are very effective at tackling state space explosion, but do not produce short counterexamples. On the other hand, directed model checking [3,4] techniques find short counterexamples, but are prone to state space explosion in the absence of errors. To the best of our knowledge, there is currently no single technique that meets both requirements. We present such a technique in this paper.

For a subset of CTL, which we call CETL (Crucial Event Temporal Logic), we show that there exists a unique **minimum** set of events in each program trace whose execution is both **necessary and sufficient** to lead to an error state. These events are called "crucial events". We show how crucial events can be used to produce short counterexamples, while also providing state space reduction.

We have implemented the techniques presented here as an extension to the model checker SPIN, called SPICED (Simple PROMELA Interpreter with Crucial Event Detection). Experimental results are presented.

1 Introduction

Partial order reduction techniques [1,2] successfully tackle state space explosion, but tend to produce lengthy error trails [3,5]. Short error trails greatly reduce debugging effort. Also, the ability to find errors at shorter depths can make it possible to verify larger models, by finding the error before the model checker runs out of computational resources. Recently, there has been much interest in the use of heuristic search techniques to produce short error trails [3,4]. Heuristic search techniques calculate a cost function for each outgoing transition from a state, then explore these transitions in the order of increasing cost. Lower cost transitions are considered to be "closer" to the error state. However, in the absence of errors, these techniques do not reduce state space explosion because they only change the order in which nodes are expanded without reducing the number of nodes to be expanded. While there has been some effort to combine heuristic search with state space reduction techniques, the combination can interfere with the efficiency of the individual techniques[5]. To the best of our knowledge, there is currently

⋆ Part of the work reported here was done while this author was at the Univ. of Texas at Austin.

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 491–503, 2008.

no single technique that achieves both objectives - state space reduction for complete validation, while narrowing down on error states quickly to produce short error trails. We present such a technique in this paper.

The set of reachable states in a (Mazurkiewicz) trace [6] of a program forms a lattice [7]. A lattice is a partial order in which every pair of elements has a unique meet (infimum) and join (supremum). A property is said to be *meet-closed* [8] in a trace if, whenever it holds at any two states in the trace, it also holds at the state given by their lattice meet. It was shown in [8] that, given a trace and a meet-closed property, there exists a unique minimum set of events in the trace whose execution is both *necessary and sufficient* to reach a state satisfying the property. We call these events *crucial events*. Executing crucial events in any order consistent with the dependency relation results in the same state [2]. Thus, for a single trace, it is sufficient to explore any one interleaving comprising entirely of crucial events. We call such an interleaving a *crucial path*. If an error state exists, any crucial path will lead to it through the fewest possible transitions. We show how crucial paths can be used to improve the efficiency of explicit-state model checking, and show how crucial events can be identified.

We identify a subset of CTL, called Crucial Event Temporal Logic (CETL), which contains only meet-closed formulae. CETL includes the existential until and release operators of CTL, and allows conjunction. Atomic propositions are limited to process-local variables. CETL does not allow negation, except for atomic propositions, nor does it allow disjunction. Despite these limitations, CETL can express many reachability, safety, liveness and response properties. In fact, of the 131 properties in the BEEM database [9], which is a large repository of benchmarks for explicit-state model checkers, 101 (77%) can be expressed in CETL.

We have implemented a CETL model checker called SPICED (Simple PROMELA Interpreter with Crucial Event Detection), using the techniques presented here. SPICED is based on the popular model checker SPIN [10]. We provide experimental results from a wide range of examples from the BEEM database [9], and from the SPIN distribution [10]. We ran experiments on 75 different variations (with differences in problem sizes and the location of errors) of 15 different models from the BEEM database. SPICED achieved trail reduction greater than 1x in 93% of the cases, greater than 10x in 55% of the cases, and greater than 100x in 19% of the cases. We completed verification faster than SPIN (with p.o. reduction) in 44% of the cases, with a 10x reduction in time in 9% of the cases. For 3 of the 15 models, we were able to verify problem sizes for which SPIN ran out of resources. We also provide results to show that we achieve state space reduction comparable to p.o. techniques even in the absence of errors.

Lattice theory has previously been applied to the verification of *finite* program traces. A survey of these applications was presented in [11]. In [12], a logic called RCTL was defined, which included the CTL operators EG, AG and EF. RCTL formulae were shown to be meet-closed, and an efficient verification algorithm for RCTL formulae was presented. However, these applications were limited to a single *finite* trace of a program, and required a partial order (implicit) representation of the state space. To the best of our knowledge, this paper is the first time these lattice theoretic concepts have been applied to explicit-state model checking of an entire program.

This paper is organized as follows. Section 2 introduces relevant concepts and notation. In Section 3, we introduce some CTL operators that preserve meet-closure, define the logic CETL, and introduce the notion of crucial events. In Section 4, we show how crucial events can be used for model checking CETL formulae within a single trace of a program, then extend this to model checking the complete program in Section 5. In Section 6, we show how crucial events are identified. Experimental results are presented in Section 7, followed by concluding remarks in Section 8.

2 Preliminaries

A *finite-state program* P is a triple (S, T, s_0) where S is a finite set of states, T is a finite set of operations, and $s_0 \in S$ is the initial state. The set of transitions that are executable from a given state $s \in S$ is denoted by $enabled(s)$. A transition $\alpha \in enabled(s)$ transforms the state s into a *unique* state s', denoted by $s' = \alpha(s)$. A state s is *reachable* in a program P iff it can be reached from s_0 by executing only enabled transitions at each state. The *full state space graph* of P is a directed graph whose vertices are the reachable states of P. An edge exists from vertex s to t iff $\exists \alpha \in enabled(s)$ such that $t = \alpha(s)$. A path through the full state space graph consists of a (finite or infinite) sequence of states. Each path has a corresponding *transition sequence*, consisting of the transitions executed along the path.

An independence relation [6,1] $I \subseteq T \times T$ is a symmetric, irreflexive relation such that $(\alpha, \beta) \in I$ iff for every state $s \in S$, (a) if $\alpha \in enabled(s)$, then $\beta \in enabled(s)$ if and only if $\beta \in enabled(\alpha(s))$, and (b) if $\alpha, \beta \in enabled(s)$, then $(\alpha(\beta(s)) = \beta(\alpha(s)))$. Simply put, the execution of α does not affect the enabledness of β, and executing α and β in either order results in the same state. We say that α, β are independent iff $(\alpha, \beta) \in I$. The dependency relation, D, is the reflexive, symmetric relation given by $D = (T \times T) \setminus I$. The independence relation I partitions the set of all transition sequences (correspondingly, paths) of a program into equivalence classes called *traces* [6]. Given two finite transition sequences u and v, we say that u and v are *trace equivalent*, denoted $u \equiv v$, iff they have the same starting state, and v can be derived from u by repeatedly commuting adjacent independent transitions.

Trace equivalence for infinite transition sequences is defined with the help of the relation $\preceq$. Given two (finite or infinite) transition sequences u and v, $u \preceq v$ iff for each finite prefix u' of u, there exists a prefix v' of v, and some w such that $v' \equiv w$, and u' is a prefix of w. For infinite sequences u, v, we have $u \equiv v$ iff $u \preceq v$ and $v \preceq u$.

Each occurrence of a transition in a transition sequence is called an *event*. For example, the transition sequence $\alpha\beta\alpha\beta$ consists of four events. We say that two events are dependent (correspondingly, independent) iff their corresponding transitions are dependent (independent). Every path of a trace starts from the same state, and consists of the same set of events. We will use the notation $\sigma = [s, v]$ to denote a trace with starting state s, and representative transition sequence v. All paths of a trace have the same length, and the same final state [6,1].

The concatenation of a finite trace $\sigma_1 = [s, v]$ with a finite or infinite trace $\sigma_2 = [t, w]$ is defined when t is also the final state of σ_1, and is given by $\sigma_1.\sigma_2 = [s, vw]$. We say that $\sigma_2 = [s, v]$ *subsumes* $\sigma_1 = [s, u]$, denoted $\sigma_1 \sqsubseteq \sigma_2$, iff $u \preceq v$. If σ_1 is finite,

then $\sigma_1 \sqsubseteq \sigma_3$ iff there exists σ_2 such that $\sigma_3 = \sigma_1.\sigma_2$. If $\sigma \sqsubseteq \sigma'$, then the reachable states of σ is a subset of the reachable states of σ'. We say that a trace of a program P is *maximal* iff no other trace of P subsumes it. Clearly, the set of maximal traces of a program contains all its reachable states.

2.1 Traces, Posets and Lattices

A 1-1 correspondence exists between traces and partially ordered sets (posets)[7,6]. Let $\sigma = [s, v]$ be a trace, and E be the set of events in v. We can define a poset $(E, \rightarrow)$, where $\forall e, f \in E : e \rightarrow f$ iff $(e, f) \in D$ and e occurs before f in v. The relation $\rightarrow$ expresses causal dependence. Every transition sequence of σ is a linearization of $(E, \rightarrow)$, and conversely every linearization of this poset is a valid transition sequence of σ. We will use the notation $\sigma = (E, \rightarrow)$ for the poset corresponding to a trace σ.

The same state can be visited multiple times during the execution of a transition sequence, for example, in the case of a cycle in the state space graph. However, each *occurrence* of the state corresponds to a unique prefix of the transition sequence. If an event e is executed as part of a transition sequence, then the events that causally precede e must have been executed before e. A subset $G \subseteq E$ of a poset $(E, \rightarrow)$ is called a *down-set* if, whenever $f \in G$, $e \in E$ and $e \rightarrow f$, we have $e \in G$. In a trace $\sigma = (E, \rightarrow)$, there exists a correspondence between occurrences of states and down-sets. That is, an occurrence of a state in σ corresponds to executing the set of events in some down-set of $(E, \rightarrow)$. Conversely, every state in σ can be reached by executing the events in some down-set of $(E, \rightarrow)$. For simplicity of presentation, in this paper we overload the term "down-set" to mean both a set of events, and an occurrence of a state.

Progress in a computation is measured by the execution of additional events from the current state. For down-sets G and H of a trace $(E, \rightarrow)$, $G \subseteq H$ iff H is reachable from G in the full state space graph. The set of all down-sets of $(E, \rightarrow)$ forms a lattice under the $\subseteq$ relation, with the meet and join operations given by set intersection and union, respectively [13,7]. That is, if G and H are down-sets of $(E, \rightarrow)$, so are $(G \cap H)$ and $(G \cup H)$. We will use $\mathcal{L}(\sigma)$ to denote the lattice of down-sets of a trace σ. Note that, while a vertex in the full state space graph corresponds to a program state, a vertex in $\mathcal{L}(\sigma)$ corresponds to an *occurrence* of a state. Figure 1 illustrates these concepts.

We say that a formula (property) is meet-closed (correspondingly, join-closed) if, whenever any two states of a trace σ satisfy the formula, the state corresponding to their meet (correspondingly, join) in $L(\sigma)$ also satisfies it. For a down-set G and formula ϕ, the notation "$G \models \phi$" means that the state corresponding to G satisfies ϕ.

Definition 1. Meet-closed *[8]: A formula ϕ is meet-closed iff, for every trace σ of a program P: $\forall G, H \in \mathcal{L}(\sigma) : [(G \models \phi) \wedge (H \models \phi) \Rightarrow (G \cap H) \models \phi]$.*

Definition 2. Join-closed*: A formula ϕ is join-closed iff, for every trace σ of a program P: $\forall G, H \in \mathcal{L}(\sigma) : [(G \models \phi) \wedge (H \models \phi) \Rightarrow (G \cup H) \models \phi]$.*

Definition 3. Regular *[14]: A formula ϕ is regular iff it is meet- and join- closed.*

In the next section, we present some CTL operators that preserve meet- and join-closure.

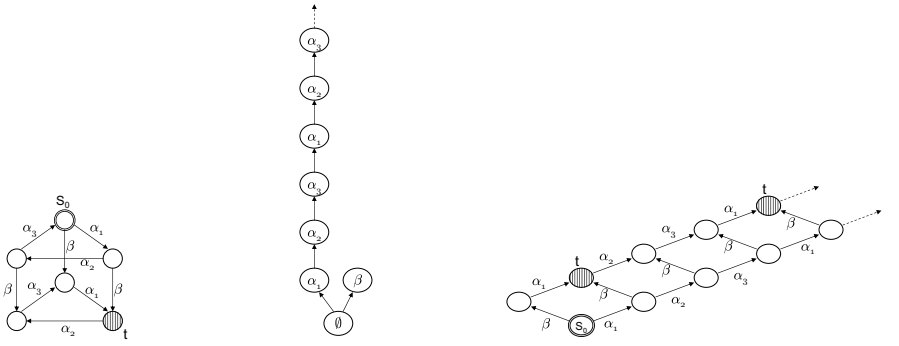

Fig. 1. (a) The full state space graph of a program P. (b) The poset corresponding to a maximal trace $\sigma = [s_0, \beta(\alpha_1\alpha_2\alpha_3)^\omega]$. (c) The lattice $\mathcal{L}(\sigma)$, showing two occurrences of a state t.

3 Meet- and Join-Closure of CTL Operators

We consider concurrent systems, where the system is modeled as a set of processes. Each process P_i has a set of transitions T_i, and a set of *local* variables V_i that can only be changed by transitions in T_i. All the transitions in T_i are pairwise dependent, that is, if $\alpha, \beta \in T_i$, then $(\alpha, \beta) \in D$. A transition in T_i can also change the values of shared (global) variables. A formula ϕ is called a *process-local state formula* iff its truth value is purely determined by the current values of the local variables V_i of some process P_i. Theorems 1 and 3 in this section are proved in [15].

Theorem 1. *Process-local state formulae are regular.*

The following theorem was proved in [14], using set union and intersection.

Theorem 2. *If ϕ_1 and ϕ_2 are regular, then $(\phi_1 \wedge \phi_2)$ is regular.*

On the other hand, disjunction does not preserve meet-closure [14].

Let π^i denote the i^{th} state on the path π. We consider the following CTL operators:

- $s \models EG(\phi)$ iff there exists a path π starting from s such that $\forall i : i \geq 0 : \pi^i \models \phi$.
- $s \models E[\phi_1 U \phi_2]$ iff there exists a path π starting from s such that $\exists j : j \geq 0 : \pi^j \models \phi_2$, and $\forall i : i < j : \pi^i \models \phi_1$.
- $EF(\phi) = E[true\ U\ \phi]$
- $E[\phi_2 R \phi_1] = E[\phi_1 U (\phi_1 \wedge \phi_2)] \vee EG(\phi_1)$

It can be shown that the existential until operator, $E[\phi_1 U \phi_2]$, does not preserve meet-closure [15]. However, a specific flavor of this operator does, as shown in the following theorem. In most cases, the system specification makes it equally valid to check for $E[\phi_1 U (\phi_1 \wedge \phi_2)]$ instead of $E[\phi_1 U \phi_2]$.

Theorem 3. *If ϕ_1 and ϕ_2 are regular, then so are $E[\phi_2 R \phi_1]$ and $E[\phi_1 U (\phi_1 \wedge \phi_2)]$.*

As $EF(\phi_1) = E[true\ U (true \wedge \phi_1)]$, and $EG(\phi_1) = E[false\ R\ \phi_1]$, and $true$ and $false$ are trivially regular, we have:

Corollary 1. *If ϕ_1 is a regular formula, so are $EF(\phi_1)$ and $EG(\phi_1)$.*

We define a logic in which every formula is regular, as are all its subformulae.

Definition 4 Crucial Event Temporal Logic (CETL). *A CETL formula is one that can be generated from the following rules:*

1. *The trivial propositions $true$ and $false$ are CETL formulae.*
2. *Every process-local state formula is a CETL formula.*
3. *If ϕ_1 and ϕ_2 are CETL formulae, so are $(\phi_1 \wedge \phi_2)$, $E[\phi_2 R \phi_1]$, and $E[\phi_1 U (\phi_1 \wedge \phi_2)]$.*

We now explore the relation between meet-closure and *crucial events*.

3.1 Crucial Events

Let G be any down-set of a trace $\sigma = (E, \rightarrow)$. Let ϕ be some meet-closed formula, and $G \not\models \phi$. Let $\mathcal{G}$ be the set of all ϕ-satisfying states that are reachable from G in σ. That is, $\mathcal{G} = \{H \in \mathcal{L}(\sigma) | G \subseteq H \wedge H \models \phi\}$. Now, $\mathcal{G}$ can be an infinite set. Let $\mathcal{H}$ be the set of elements of $\mathcal{G}$ that are minimal under $\subseteq$:

$$\mathcal{H} = \{H \in \mathcal{G} | \forall H' : H \subset H' \Rightarrow H' \notin \mathcal{H}\} \tag{1}$$

$\mathcal{H}$ is necessarily finite for finite-state programs. We now define $K = \bigcap_{H \in \mathcal{H}} H$. By the meet-closure of ϕ, $K \models \phi$. Also, $G \subseteq K$. That is, K is the unique and well-defined ϕ-satisfying state that is reachable from G by executing the *fewest* events. In particular, $K \setminus G$ is the minimum set of events that *must* be executed along any path starting from G, to reach a ϕ-satisfying state in σ. The events in $K \setminus G$ are called **crucial events** [8].

Definition 5. Crucial event: *In a trace σ, an event e is said to be crucial from a state G with respect to a meet-closed formula ϕ, denoted $e \in crucial(G, \phi, \sigma)$ iff:*

$$\forall H \in \mathcal{L}(\sigma) : (G \subseteq H) \wedge (G \not\models \phi) \wedge (H \models \phi) \Rightarrow (e \in H \setminus G)$$

A transition sequence starting from G and comprising exactly of the events in $crucial(G, \phi, \sigma)$ gives us a path of shortest length from G to a ϕ-satisfying state in σ. Such a path is called a **crucial path**. A special case arises when $\mathcal{H} = \emptyset$. In this case, we define $K = E$ (the set of all events), and any maximal path starting from G in $\mathcal{L}(\sigma)$ constitutes a crucial path. The proof for the following theorem is straightforward.

Theorem 4. *Let $\mathcal{H}$ be as defined in (1). If $\mathcal{H} \neq \emptyset$, then a crucial path for ϕ starting from G cannot contain a cycle.*

Recall that a down-set is an occurrence of a state. Suppose the down-set G is an occurrence of the state s. Executing the events in $crucial(G, \phi, \sigma)$ from s will lead to a ϕ-satisfying state in the full state space graph. The state s can have multiple occurrences in σ (for example, in Figure 1(c), the state t occurs multiple times in σ_2). Let G' be another down-set of σ that is also an occurrence of s. It is easy to see that $crucial(G, \phi, \sigma) = crucial(G', \phi, \sigma)$. Thus, every occurrence of s in σ has the same set of crucial events w.r.t. ϕ. Based on this observation, we define $crucial(s, \phi, \sigma) \equiv crucial(G, \phi, \sigma)$, where G is any down-set of σ that is an occurrence of s.

The complexity of identifying the exact set of events that constitutes $crucial(s, \phi, \sigma)$ for a given CETL formula ϕ is an open problem. However, we can identify a *subset* of $crucial(s, \phi, \sigma)$ in most cases, as we shall see in Section 6.

If G is a down-set of $\mathcal{L}(\sigma)$, and H is an immediate successor of G in $\mathcal{L}(\sigma)$, we denote this by $G \triangleright H$. Formally, if $G, H \in \mathcal{L}(\sigma)$, and $\exists e \notin G$, and $H = G \cup \{e\}$, then $G \triangleright H$. The notation $G \trianglerighteq H$ means $(G \triangleright H) \vee (G = H)$. The following lemmas are used in the proofs presented in Sections 4.1 and 4.2, and are proved in [15].

Lemma 1. *For a trace σ and $C, D, F \in \mathcal{L}(\sigma)$, if $C \triangleright F$ and $D \subseteq F$, then $(C \cap D) \trianglerighteq D$.*

Lemma 2. *For a trace σ and $C, D, F \in \mathcal{L}(\sigma)$, if $F \triangleright C$ and $F \subseteq D$, then $D \trianglerighteq (C \cup D)$.*

We now show how the concepts presented so far can be used to prune the state space while model checking CETL formulae. In particular, we show that it is sufficient to explore only crucial paths in order to verify a CETL formula. For better presentation, we start with the problem of verifying a CETL formula on a single trace of a program. We will consider CETL model checking for the complete program in Section 5.

4 Model Checking CETL in a Program Trace

The approach we present here can be used to enhance any local, recursive CTL model checking algorithm, such as ALMC [16]. A *local* model checking algorithm starts from an initial program state and performs a state space exploration using either depth-first or breadth-first search. *Recursive* means that the truth value of subformulae are determined at a state before the top formula is evaluated. In this section, we show that rather than exploring all enabled events from a state, it is sufficient to explore only a subset of these events in order to verify a CETL formula in a given trace. This explored subset is called an "ample set" [1]. The ample set chosen at a state s while verifying a CETL formula ϕ is denoted by $ample(s, \phi)$. In the non-reduced (baseline) case, $ample(s, \phi) = enabled(s)$.

4.1 $E[\phi_1 U(\phi_1 \wedge \phi_2)]$

Let G_0 be some down-set of σ that satisfies $E[\phi_1 U(\phi_1 \wedge \phi_2)]$. Let π be the corresponding witness path with $\pi^l = H$ as its final state. Then, $\forall j : 0 \leq j \leq l : \pi^j \models \phi_1$, and $H \models (\phi_1 \wedge \phi_2)$. Let $\mathcal{J}$ be the set of all down-sets of σ that are reachable from G_0, are minimal under $\subseteq$ (this ensures that $\mathcal{J}$ is finite), and satisfy $(\phi_1 \wedge \phi_2)$. Define:

$$G = \bigcap_{J \in \mathcal{J}} J \tag{2}$$

Since $(\phi_1 \wedge \phi_2)$ is regular, $G \models (\phi_1 \wedge \phi_2)$.

Theorem 5. *There exists a path from G_0 to G such that every state along the path satisfies ϕ_1.*

Proof. We will construct a path λ from G_0 to G, consisting entirely of ϕ_1-satisfying states. We construct this path backwards, starting from $\lambda^k = G$, towards $\lambda^0 = G_0$.

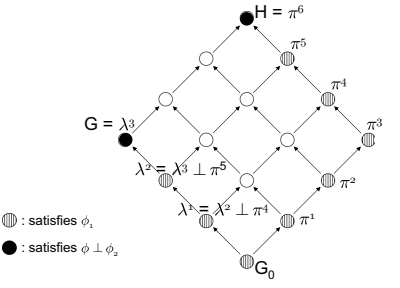

Fig. 2. Example illustrating the construction of Theorem 5

We show that, if $\lambda^i \models \phi_1$ for any $1 \leq i \leq k$, there exists a $G' \triangleright \lambda^i$ such that $G' \models \phi_1$. We can then extend λ with $\lambda^{i-1} = G'$, and proceed with our construction. For the base case, we have $\lambda^k = G$, and $G \models \phi_1$.

Let $1 \leq j \leq l$ be the *least* j such that $\lambda^i \subseteq \pi^j$. First, we show that such a j must exist. Recall that $\pi^l = H$, and $\lambda^i \subseteq G \subseteq H$. Therefore, for some $j \leq l$, $\lambda^i \subseteq \pi^j$. Also, $\pi^0 = \lambda^0 = G_0$, so $\forall i : i \geq 1 : \lambda^i \not\subseteq \pi^0$. Therefore, $j \geq 1$. Since j is the least such, we have $\lambda^i \not\subseteq \pi^{j-1}$. So, we have $\pi^{j-1} \triangleright \pi^j$, and $\lambda^i \subseteq \pi^j$. From Lemma 1, this implies $(\lambda^i \cap \pi^{j-1}) \trianglerighteq \lambda^i$. We cannot have $(\lambda^i \cap \pi^{j-1}) = \lambda^i$, because this would imply $\lambda^i \subseteq \pi^{j-1}$, which is a contradiction. Therefore, $(\lambda^i \cap \pi^{j-1}) \triangleright \lambda^i$. Set $G' = (\lambda^i \cap \pi^{j-1})$. Since $\lambda^i \models \phi_1$, and $\pi^{j-1} \models \phi_1$, by the meet-closure of ϕ_1, $G' \models \phi_1$. □

Theorem 5 tells us that if $G_0 \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$, then a crucial path for $(\phi_1 \wedge \phi_2)$ can act as a witness. Since $G_0 \models \phi_1$, and every state along the witness path satisfies ϕ_1, it is easy to see that $crucial(G_0, (\phi_1 \wedge \phi_2), \sigma) = crucial(G_0, \phi_2, \sigma)$. The following theorem shows how we can construct this path "forward", that is, starting from G_0.

Theorem 6. *To construct the path of Theorem 5, starting from G_0, at each state H we execute a single enabled event α such that $\alpha \in crucial(H, \phi_2, \sigma)$, and $H \cup \{\alpha\} \models \phi_1$.*

Proof. Let G be as in (2). From Theorem 5, there exists *some* path λ such that $\lambda^0 = G_0$, $\lambda^k = G$, and $\forall j : 0 \leq j \leq k : \lambda^j \models \phi_1$. We need to show that we can construct such a path by choosing, at each state, any crucial event that leads to a ϕ_1-satisfying successor.

Clearly, if every event along our path is crucial for ϕ_2, then our path will lead to G. We need to show that at any state H along our constructed path, there exists a successor J such that $J \models \phi_1$. To begin with, $H = G_0$. Of course, our construction ends when $H = G$, so any H for which a successor needs to be found must be a strict subset of G.

Let $0 \leq i < k$ be the *greatest* i such that $\lambda^i \subseteq H$. We first show that such an i exists. Note that $\lambda^0 = G_0 \subseteq H$. Thus, for some $i \geq 0 : \lambda^i \subseteq H$. Also, $\lambda^k = G$, and $H \subset G$. Therefore, $\lambda^k \not\subseteq H$, so $i < k$. Since i is the greatest such, we have $\lambda^{i+1} \not\subseteq H$. Now, $\lambda^i \triangleright \lambda^{i+1}$, and $\lambda^i \subseteq H$. By Lemma 2, $H \trianglerighteq (\lambda^{i+1} \cup H)$. If $H = (\lambda^{i+1} \cup H)$, then $\lambda^{i+1} \subseteq H$, which is a contradiction. Therefore, $H \triangleright (\lambda^{i+1} \cup H)$. Also, $H \models \phi_1$, and $\lambda^{i+1} \models \phi_1$, so by the join-closure of ϕ_1, $\lambda^{i+1} \cup H \models \phi_1$. Hence, $J = \lambda^{i+1} \cup H$ is the required successor for H. □

4.2 $E[\phi_2 R \phi_1]$

Recall that $E[\phi_2 R \phi_1] \stackrel{def}{\equiv} E[\phi_1 U(\phi_1 \wedge \phi_2)] \vee EG(\phi_1)$. Theorem 6 showed how to construct a witness for $G_0 \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$. The following theorem shows how to construct a witness for $G_0 \models EG(\phi_1)$.

Theorem 7. *Let $G_0 \in \mathcal{L}(\sigma)$ such that $G_0 \models EG(\phi_1)$ in σ. We can construct a witness path as follows. Starting from G_0, at each state H, we execute a single enabled event α such that $H \cup \{\alpha\} \models \phi_1$.*

Proof. We simply need to show that, for every state H on the constructed path, there exists a ϕ_1-satisfying successor state. The proof is similar to that of Theorem 6. □

Theorems 6 and 7 show that, given a formula ϕ of the form $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$, and a trace $\sigma = [s, v]$, we can decide if $s \models \phi$ by exploring only ample sets satisfying the following condition:
(C0) If $ample(s, \phi) \neq enabled(s)$, then $ample(s, \phi) = \{\alpha\}$, where $\alpha \in crucial(s, \phi_2, \sigma)$, and $\alpha(s) \models \phi_1$.

5 Model Checking CETL in a Program

Let ϕ be a CETL formula of the form $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$. We now consider the problem of deciding whether a given state s satisfies ϕ in a program. In this case, we need to explore a crucial path for every maximal program trace starting from s. That is, for every maximal trace σ starting from s, $ample(s, \phi)$ must contain some event from $crucial(s, \phi_2, \sigma)$. In [1], it was shown that if $ample(s, \phi)$ satisfies the following condition (C1), then it contains a successor for each maximal trace starting from s.

(C1) Along every path starting from s in the full state space graph, a transition that is dependent on a transition from $ample(s, \phi)$ cannot be executed without a transition from $ample(s, \phi)$ occurring first.

Theorem 8. *[1] If $ample(s, \phi)$ satisfies condition (C1), then for every maximal trace σ starting from s, there exists some $\alpha \in ample(s, \phi)$ such that $[s, \alpha] \sqsubseteq \sigma$.*

By Theorem 8, an ample set satisfying condition (C1) will generate some successor for each maximal trace starting from s. We now need to ensure that each event in the ample set is crucial in *every* trace to which it belongs.

Definition 6. Universally crucial event: *An event α is said to be universally crucial from a state s for a meet-closed formula ϕ_2, denoted $\alpha \in ucrucial(s, \phi_2)$, iff for every trace σ such that $[s, \alpha] \sqsubseteq \sigma$, $\alpha \in crucial(s, \phi_2, \sigma)$.*

The following is a straightforward extension of condition (C0) from Section 4.

(C2) If $ample(s, \phi) \neq enabled(s)$, then for each $\alpha \in ample(s, \phi)$, $\alpha \in ucrucial(s, \phi_2)$ and $\alpha(s) \models \phi_1$.

The following theorem is proved in [15].

Theorem 9. *To determine whether $s \models \phi$, it is sufficient to explore ample sets satisfying (C1) and (C2).*

To construct an ample set satisfying (C1) and (C2), we need to identify a subset of $enabled(s)$ that satisfies these conditions. Condition (C1) is used in p.o. reduction [1], and we use the techniques from [17] to construct a subset of $enabled(s)$ that satisfies (C1). Condition (C2) requires us to identify a subset of $enabled(s)$ that consists of universally crucial events. We now show how universally crucial events can be identified.

6 Identifying Universally Crucial Events

In the previous section, we derived the conditions for $ample(s, \phi)$, where ϕ is a CETL formula of the form $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$. Note that our construction of a witness path for $s \models \phi$ ends with success when we encounter a state which satisfies ϕ_2. Therefore, we are only interested in constructing ample sets for states at which ϕ_2 does not hold. That is, we are interested in $ucrucial(s, \phi_2)$ when $s \not\models \phi_2$. The problem of identifying $ucrucial(s, \phi_2)$ for a general CETL formula ϕ_2 remains open. In this section, we identify some cases for which we can determine universally crucial events. Recall that T_i is the set of transitions of process P_i.

Theorem 10. *If ϕ_2 is a process-local state formula in process P_i, then $T_i \cap enabled(s) \subseteq ucrucial(s, \phi_2)$.*

Proof. Only transitions in T_i can change the truth value of ϕ_2. Therefore, we must execute some transition in T_i from s in order to reach a state in which ϕ_2 holds. Recall that transitions from the same process are pairwise-dependent. Therefore, each transition in $T_i \cap enabled(s)$ gives rise to a different trace. Therefore, each event $\alpha \in T_i \cap enabled(s)$ is crucial in every trace that subsumes $[s, \alpha]$. Thus, $T_i \cap enabled(s) \subseteq ucrucial(s, \phi)$. □

Recall, from Section 4, that our approach applies to *recursive* model checking algorithms. Thus, in the following two theorems, ψ_1 and ψ_2 have already been evaluated at s before ϕ_2 is evaluated. The proof of Theorem 11 is straightforward.

Theorem 11. *Let $\phi_2 = \psi_1 \wedge \psi_2$. If $s \not\models \psi_1$ then $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$, else $ucrucial(s, \psi_2) \subseteq ucrucial(s, \phi_2)$.*

Theorem 12. *Let ϕ_2 be of the form $E[\psi_1 U(\psi_1 \wedge \psi_2)]$ or $E[\psi_2 R \psi_1]$. If $s \not\models \psi_1$, then $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$. Else, if $s \models \psi_1$ and $\neg\psi_1$ is meet-closed, then $ucrucial(s, \neg\psi_1) \subseteq ucrucial(s, \phi_2)$.*

Proof.
- **Case 1:** $s \not\models \psi_1$. Any state that satisfies ϕ_2 must also satisfy ψ_1. Therefore, we first need to execute the minimum set of events that will lead to a state satisfying ψ_1. Hence, $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$.
- **Case 2:** $s \models \psi_1$. Recall that $s \not\models \phi_2$, and we are interested in reaching a state that satisfies ϕ_2. Assume there exists a state t, reachable from s, that satisfies ϕ_2. Let w be a witness for $t \models \phi_2$. Then, along every path v from s to t, there must exist some state s' such that $s' \not\models \psi_1$. If not, then $v.w$ would have been a witness for $s \models \phi_2$. Thus, in order to reach a state that satisfies ϕ_2, it is necessary to go through some state that satisfies $\neg\psi_1$. If $\neg\psi_1$ is meet-closed, then the execution of the events in $ucrucial(s, \neg\psi_1)$ is necessary to reach a state satisfying $\neg\psi_1$. Therefore, $ucrucial(s, \neg\psi_1) \subseteq ucrucial(s, \phi_2)$.

7 Implementation and Experimental Results

We have implemented our approach as an extension to the SPIN model checker [10], called SPICED (Simple PROMELA Interpreter with Crucial Event Detection). Our implementation incorporates the ample set selection techniques presented in this paper into ALMC [16], a local CTL model checking algorithm based on depth-first search. The complete algorithm can be found in [16], as well as in the technical report version of this paper [15]. Our implementation of SPICED, along with detailed experimental results, is available at: `http://maple.ece.utexas.edu/spiced`.

We ran SPICED against a large set of examples from the BEEM database [9], which contains PROMELA (the input language for SPIN) models with errors injected into them, and lists the properties to be verified on these models. All experiments were performed on a 1-cpu 2.8 GHz Intel Pentium 4 machine with 512 MB RAM, running Red Hat Enterprise Linux WS Rel 4.

Table 1. Trail reduction with SPICED, compared to SPIN with p.o. reduction

Model	Tool	Time (sec)	States	Memory (MB)	Formula	Trail length
phils.7	SPICED	0.01	15	3.15	$EF(P_0.req \wedge EG(!P_0.grant))$	6
	SPIN	**Could not complete**			$\neg\Box(req0 \Rightarrow \Diamond grant0)$	-
szymanski.9	SPICED	0.02	256	3.15	$EF(P_0.wait \wedge EG(!P_0.cs))$	43
	SPIN	**Could not complete**			$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	-
fischer.18	SPICED	0.02	28	3.15	$EF(P_0.wait \wedge EG(!P_0.cs))$	19
	SPIN	**Could not complete**			$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	-
mcs.5	SPICED	0.09	30227	4.89	$EF(P_0.wait \wedge EG(!P_0.cs))$	14
	SPIN	0.03	2821	2.72	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	5646
anderson.7	SPICED	0.03	65387	7.03	$EF(P_0.wait \wedge EG(!P_0.cs))$	82
	SPIN	0.13	15692	6.63	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	31389
peterson.7	SPICED	0.09	29080	4.89	$EF(P_0.wait \wedge EG(!P_0.cs))$	159
	SPIN	0.1	9992	9.93	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	19984
lamport.7	SPICED	0.06	6850	3.45	$EF(P_0.wait \wedge EG(!P_0.cs))$	30
	SPIN	0.02	665	2.62	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	1330
at.7	SPICED	0.02	19	3.15	$EF(P_0.wait \wedge EG(!P_0.cs))$	11
	SPIN	0.01	182	2.62	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	370
bakery.6	SPICED	0.01	69	3.15	$EF(P_0.wait \wedge EG(!P_0.cs))$	46
	SPIN	0.02	896	2.62	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	856
gear.2	SPICED	0.03	4185	3.13	$EF(Clutch.err\_open)$	5056
	SPIN	0.13	22386	5.5	local assert()	19396
needham.4	SPICED	0.01	27	2.72	$EF(init0.fin \wedge resp0.fin)$	15
	SPIN	0.04	4003	3.03	$\neg\Diamond(init\_fin \wedge resp\_fin)$	52
msmie.2	SPICED	0.02	83	2.72	$EF(P_0.wait \wedge EG(!P_0.cs))$	63
	SPIN	0.01	370	2.62	$\neg\Box(wait0 \Rightarrow \Diamond cs0)$	214
loyd.2	SPICED	0.19	50931	9.24	$EF(Check.done)$	52597
	SPIN	0.63	166133	17.61	local assert()	84418
driving_phils.4	SPICED	0.01	212	3.15	$EF(P_0.req \wedge EG(!P_0.grant))$	123
	SPIN	0.01	85	2.62	$\neg\Box(req0 \Rightarrow \Diamond grant0)$	170

Table 2. State space reduction with SPICED

Model	Tool	Time (sec)	States	Memory (MB)	Formula
sort	SPIN, no reduction	1.19	107713	20.64	-
	SPIN, p.o. reduction	0.1	135	2.62	-
	SPICED	0.1	148	2.72	$EG(!left.tstvar)$
leader	SPIN, no reduction	0.17	15779	3.35	-
	SPIN, p.o. reduction	0.01	97	2.62	-
	SPICED	0.05	104	2.72	$EG(!node[4].tstvar)$
eratosthenes	SPIN, no reduction	0.52	49790	9.07	-
	SPIN, p.o. reduction	0.02	3437	3.03	-
	SPICED	0.02	2986	3.13	$EG(!sieve[0].tstvar)$
snoopy	SPIN, no reduction	0.53	81013	11.34	-
	SPIN, p.o. reduction	0.06	14169	4.06	-
	SPICED	0.4	58081	9.69	$EF(cpu0.tstvar)$

Table 1 shows the results for the largest problem sizes, for each of the verified models. A comprehensive list of results is available from our website: `http://maple.ece.utexas.edu/spiced`. In our experiments, SPICED produced error trails that were shorter than SPIN's in 93% of the cases, with no deterioration in trail length for the remaining 7% of cases. Our error trails were $>$10x shorter in 55% of the cases and $>$100x shorter in 19% of the cases. For 44% of the cases, SPICED completed verification faster than SPIN, with $>$10x reduction in time in 9% of the cases. Although CETL is a branching-time logic, in these examples, the properties were in LTL $\cap$ CETL, so the error trails were non-branching. The error trails were produced in the same format as those of SPIN's, and can be examined using SPIN's guided simulation feature. For SPIN, never claims were used for the verification of LTL properties, and simple assert() statements were used for reachability detection. For SPICED, the CETL formulae were specified a separate file, and fed directly as input to our model checking algorithm.

Table 2 shows the state space reduction achieved by SPICED, compared to SPIN with p.o. reduction, in the absence of errors. The examples in Table 2 are from the SPIN distribution [10], and have previously been used to showcase the effectiveness of p.o. reduction [18]. For SPIN, no LTL properties were specified during verification, which is optimal for maximizing the effectiveness of p.o. reduction. Since our algorithm is based on choosing crucial events, it requires the specification of a property. For each example, we chose a property that is never satisfied in the program, and forces exhaustive validation. Our results show state space reduction comparable to p.o. techniques.

8 Conclusions and Future Work

We have presented a model checking technique that produces short error trails, while achieving state space reduction. Experimental results confirm that our approach can significantly outperform SPIN in the presence of errors, while providing state space reduction comparable to partial order techniques. The effectiveness of our approach depends on the ability to identify crucial events. We have shown how crucial events can be identified in some cases. The problem of finding crucial events for a general CETL formula is a direction for future research.

References

1. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
2. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, p. 142. Springer, New York (1996)
3. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. International Journal on Software Tools for Technology Transfer 6(4) (2004)
4. Tan, J., Avrunin, G.S., Clarke, L.A., Zilberstein, S., Leue, S.: Heuristic-guided counterexample search in FLAVERS. SIGSOFT Softw. Eng. Notes 29(6), 201–210 (2004)
5. Lluch-Lafuente, A., Edelkamp, S., Leue, S.: Partial order reduction in directed model checking. In: Proceedings of the 9th International SPIN Workshop on Model Checking of Software, London, UK, pp. 112–127. Springer, Heidelberg (2002)
6. Mazurkiewicz, A.W.: Basic notions of trace theory. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, London, UK, pp. 285–363. Springer, Heidelberg (1989)
7. Winskel, G.: Event structures. In: Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency, New York, NY, USA, pp. 325–392. Springer-Verlag New York, Inc, Heidelberg (1987)
8. Chase, C.M., Garg, V.K.: Efficient detection of restricted classes of global predicates. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, pp. 303–317. Springer, Heidelberg (1995)
9. Pelanek, R.: BEEM: BEnchmarks for Explicit Model checkers (2007), `http://anna.fi.muni.cz/models/index.html`
10. Holzmann, G.: On-the-fly LTL model checking with SPIN (2007), `http://spinroot.com/spin/`
11. Garg, V.K., Mittal, N., Sen, A.: Applications of lattice theory to distributed computing. ACM SIGACT Notes 34(3), 40–61 (2003)
12. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, pp. 171–183. Springer, Heidelberg (2004)
13. Davey, B., Priestley, H.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
14. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: ICDCS 2001: Proceedings of the The 21st International Conference on Distributed Computing Systems, p. 322. IEEE Computer Society, Washington (2001)
15. Kashyap, S., Garg, V.K.: Producing short counterexamples using "crucial events". Technical Report TR-PDS-2008-002, ECE Dept, University of Texas at Austin (2008), `http://maple.ece.utexas.edu/TechReports/2008/TR-PDS-2008-002.pdf`
16. Vergauwen, B., Lewi, J.: A linear local model checking algorithm for CTL. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 447–461. Springer, Heidelberg (1993)
17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
18. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. International Journal on Software Tools for Technology Transfer (STTT) 2(3), 279–287 (1999)