# 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 be able to successfully tackle state space explosion for complete validation of the system, while providing **short** counterexamples in case an error exists. Techniques such as partial order (p.o.) reduction [1, 2] are very effective at tackling state space explosion, but do not concern themselves with the production of short counterexamples. On the other hand, directed model checking [3, 4] techniques use heuristic search strategies to find short counterexamples, but are not suited for exhaustive validation, because they 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.

We identify a subset of CTL, which we call CETL (Crucial Event Temporal Logic), that exhibits some interesting properties. Given any (Mazurkiewicz) trace of a program and any CETL formula, there exists a unique set of events, called **crucial events**, whose execution is both necessary and sufficient to lead to a state satisfying the formula. These crucial events can be executed in any order that is consistent with the dependency relation. Thus, for exhaustive validation, it is sufficient to explore a single interleaving, consisting entirely of crucial events, per maximal trace of the program. This results in significant state space reduction, comparable to partial order techniques. Additionally, by executing only crucial events, we narrow in on the error quickly, resulting in faster verification and short counterexamples. We present an explicit-state model checking algorithm for CETL, and show how crucial events can be identified.

We have implemented our algorithms as an extension to SPIN, called SPICED (Simple PROMELA Interpreter with Crucial Event Detection). We present experimental results comparing our performance against that of SPIN with p.o. reduction. The experimental results presented show that we consistently produce significantly shorter error trails, often resulting in faster verification times, while achieving state space reduction similar to that of p.o. reduction.

## 1 Introduction

The ability to tackle state space explosion is a fundamental requirement in a model checking tool. Partial order reduction [1, 2] has been particularly useful for state space reduction in concurrent and distributed systems, but tends to produce lengthy error trails [3, 5]. The ability to produce error trails is one of the great strengths of model checking. Shorter error trails greatly reduce debugging effort, because they are easier to comprehend. Additionally, the ability to find errors at shorter depths can make it possible to verify larger models, by finding the error state before the model checker runs out of available 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 the current 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, heuristic search techniques do nothing to reduce state space explosion, because they simply change the order in which nodes are expanded, without reducing the number of nodes to be expanded. Thus, these methods are not suitable for complete validation of the model.

There has been some effort to combine heuristic search techniques with state space reduction techniques [5, 6]. However, the combination can interfere with the efficiency of the individual techniques, either resulting in less reduction [6], or lengthier error trails [5]. To the best of our knowledge, there is currently 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.

Our approach is based on lattice theory. It has been shown that the set of reachable states in a (Mazurkiewicz) trace [7] of a program forms a lattice [8, 9]. A lattice is a partial order in which every pair of elements has a unique meet

[*] Part of the work reported here was done while this author was at the University of Texas at Austin.

(infimum) and join (supremum). A property to be verified is said to be *meet-closed* [10] 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 [10] that the class of meet-closed properties is precisely the class for which, in each program trace, there exists a unique minimum set of events 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 of the trace 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. If the explored crucial path does not encounter an error state, then there is no error state in the trace.

In this paper, we show that every formula in a subset of CTL, which we call Crucial Event Temporal Logic (CETL), exhibits meet-closure. 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 at the level of 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 [11], which is a large repository of benchmarks for explicit-state model checkers, 101 (77%) can be expressed in CETL.

We present an explicit-state model checking algorithm for CETL formulae, and show how crucial events can be identified. While the problem of identifying a crucial event for a general CETL formula remains open, there are many cases where we *can* identify crucial events.

We have implemented our approach as an extension to the popular model checker SPIN [12]. We call our tool SPICED (Simple PROMELA Interpreter with Crucial Event Detection). We provide experimental results from a wide range of examples from the BEEM database [11], as well as from the SPIN distribution [13]. 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. In 100% of our tests, the error trails produced by SPICED were at least as short as those produced by SPIN. 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 experimental results that show that we achieve state space reduction comparable to p.o. reduction even in the absence of errors.

This paper is organized as follows. Section 2 introduces relevant concepts and notation. In Section 3, we prove the meet-closure of some CTL operators, define the logic CETL, and present the notion of crucial events. Section 4 presents the basic CTL model checking algorithm that we use as the basis for our algorithm. We enhance this algorithm to use crucial events for model checking CETL formulae within a single trace in Section 5, and then within a complete program in Section 6. In Section 7, we show how crucial events are identified. Experimental results are presented in Section 8, followed by concluding remarks in Section 9.

### 1.1 Related work

Lattice theory has previously been applied to the verification of finite execution traces of distributed programs, where it has been shown to result in significant reduction in the time and memory required for verification. A survey of these applications was presented in [14]. The concept of meet-closure of properties, and crucial events, was first presented in [10], along with an algorithm for reachability detection of meet-closed properties on finite traces. In [15], 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, all these previous applications were limited to a single *finite* trace of a program. Also, they 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.

## 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 said to be *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 vertex set is exactly the set of 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 [7, 1] $I \subseteq T \times T$ is a symmetric, antireflexive relation such that $(\alpha, \beta) \in I$ iff:

- **Enabledness:** If $\alpha \in enabled(s)$, then $\beta \in enabled(s)$ if and only if $\beta \in enabled(\alpha(s))$, and
- **Commutativity:** If $\alpha, \beta \in enabled(s)$, then $(\alpha(\beta(s)) = \beta(\alpha(s)))$.

The enabledness condition states that execution of $\alpha$ does not affect the enabledness of $\beta$, and the commutativity condition states that executing $\alpha$ and $\beta$ in either order results in the same state. We say that $\alpha, \beta$ 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 $P$ into equivalence classes called *traces* [7]. 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 $w$. 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 [7, 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 $\sigma_1$, 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 $\sigma_1$ is finite, then $\sigma_1 \sqsubseteq \sigma_3$ iff there exists $\sigma_2$ such that $\sigma_3 = \sigma_1.\sigma_2$. If $\sigma \sqsubseteq \sigma'$, then the reachable states of $\sigma$ is a subset of the reachable states of $\sigma'$. 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

It is well-known [8, 7] that a 1-1 correspondence exists between traces and partially ordered sets (posets). 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. For instance, if an event $e$ denotes the sending of a message, and $f$ the corresponding receive event, then $e \rightarrow f$. Every transition sequence of $\sigma$ is a linearization of $(E, \rightarrow)$, and conversely every linearization of this poset is a valid transition sequence of $\sigma$. We will use the notation $\sigma = (E, \rightarrow)$ for the poset corresponding to a trace $\sigma$.

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 $\sigma$ corresponds to executing the set of events in some down-set of $(E, \rightarrow)$. Conversely, every state in $\sigma$ 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.

From lattice theory [16], 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. 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 $\sigma$. 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. This view of occurrences of states as elements of a lattice was previously explored in [8, 9, 17], among others. Figure 1 illustrates the relationship between programs, traces, posets, and lattices.

We say that a formula (property) is meet-closed (correspondingly, join-closed) if, whenever any two states of a trace $\sigma$ satisfy the formula, the state corresponding to their meet (correspondingly, join) in $L(\sigma)$ also satisfies it.

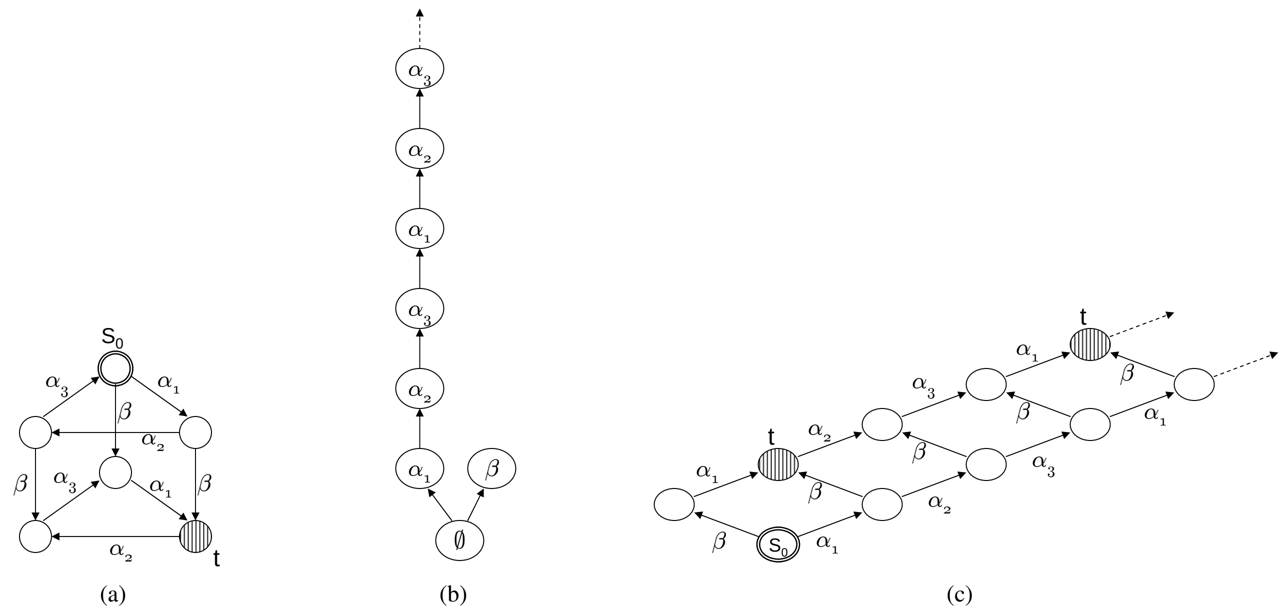


**Fig. 1.** A program $P$ with transitions $T = \{\alpha_1, \alpha_2, \alpha_3, \beta\}$. $\forall i, j : (\alpha_i, \beta) \in I$. $P$ has one maximal trace: $\sigma = [s_0, \beta(\alpha_1\alpha_2\alpha_3)^\omega]$ (a) The full state space graph. (b) The poset corresponding to $\sigma$. (c) The down-set lattice $\mathcal{L}(\sigma)$. Two occurrences of the state $t$ in $\mathcal{L}(\sigma)$ are shown.

**Definition 1. Meet-closed** *[10]: A formula $\phi$ is meet-closed iff, for every trace $\sigma$:*

$$\forall G, H \in \mathcal{L}(\sigma) : [(G \models \phi) \wedge (H \models \phi) \Rightarrow (G \cap H) \models \phi]$$

**Definition 2. Join-closed***: A formula $\phi$ is join-closed iff, for every trace $\sigma$:*

$$\forall G, H \in \mathcal{L}(\sigma) : [(G \models \phi) \wedge (H \models \phi) \Rightarrow (G \cup H) \models \phi]$$

**Definition 3. Regular** *[17]: A formula $\phi$ is regular iff it is meet- and join- closed.*

In the next section, we prove that some commonly-used CTL operators exhibit meet- and join-closure.

## 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 $\phi$ 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 $P_i$.

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

*Proof.* Let $\sigma = [s, v]$ be a trace, and $\phi$ a process-local state formula defined on the local variables of process $P_j$. Since no two transitions from $P_j$ are independent, no two transitions from $P_j$ can commute with each other. So, the events from $P_j$ must occur in the same sequence in every path of $\sigma$.

Let $v_j$ be the restriction of $v$ to events from $P_j$, *i.e.*, $v_j$ is obtained from $v$ by deleting all events from processes other than $P_j$. Let $G$ and $H$ be any two down-sets of $\sigma$ such that $G \models \phi$ and $H \models \phi$. Let $u$ and $w$ be any two transition sequences leading, respectively, from $s$ to $G$ and $s$ to $H$ in $\mathcal{L}(\sigma)$. Then, both $u_j$ and $w_j$ (derived in a similar fashion as $v_j$ from $v$) are prefixes of $v_j$. Thus, either $u_j$ is a prefix of $w_j$, or $w_j$ is a prefix of $u_j$.

WLOG, say $u_j$ is a prefix of $w_j$. Let $u'$ be any transition sequence from $s$ to $(G \cap H)$ in $\mathcal{L}(\sigma)$. Clearly, $u'_j$ can only contain events that are common to both $u_j$ and $w_j$. Now, $u_j$ is a prefix of $w_j$. Therefore, $u'_j$ contains the same events as $u_j$, and since all transitions from $P_j$ must occur in the same sequence in all paths, $u'_j = u_j$. Since the truth value of $\phi$ is determined purely by events from process $P_j$, and $G \models \phi$, we have $(G \cap H) \models \phi$.

Similarly, let $w'$ be some transition sequence from $s$ to $(G \cup H)$ in $\mathcal{L}(\sigma)$. By a similar argument as above, we can show that $w'_j = w_j$, hence $(G \cup H) \models \phi$. □

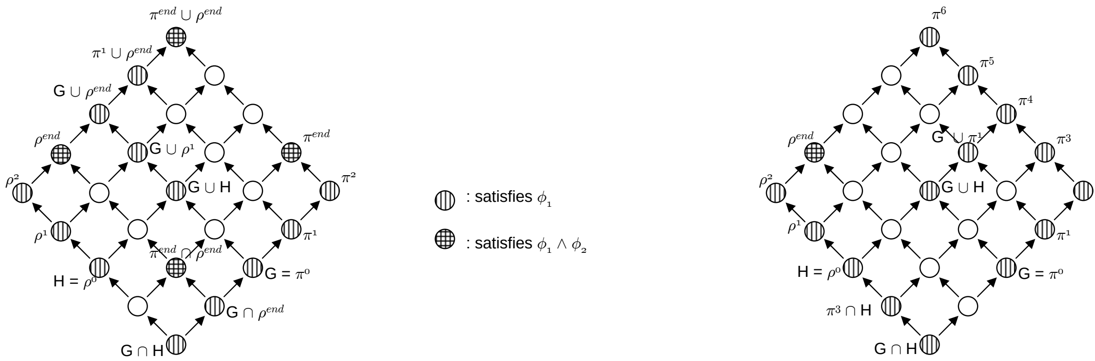
The following theorem was proved in [17], using the properties of set union and intersection.

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

On the other hand, disjunction does not preserve meet-closure [17].

Let $\pi^i$ denote the $i^{th}$ state on the path $\pi$. The following CTL operators are considered in this paper:

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



(a) (b)

**Fig. 2.** Illustrating the construction of $\lambda$ and $\nu$ in Theorem 3. (a) Case 1: $G, H \models E[\phi_1 U (\phi_1 \wedge \phi_2)]$ (b) Case 2: $G \models EG(\phi_1)$

It can be shown that the existential until operator, $E[\phi_1 U \phi_2]$, does not preserve meet-closure [18]. 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.** *The following temporal formulae are regular, if $\phi_1$ and $\phi_2$ are regular:*

- $E[\phi_2 R \phi_1]$
- $E[\phi_1 U (\phi_1 \wedge \phi_2)]$

*Proof.* We show that $E[\phi_2 R \phi_1]$ is regular, for regular $\phi_1$ and $\phi_2$. Let $G, H \in \mathcal{L}(\sigma)$.

$$\begin{array}{lcr} & (G \models E[\phi_2 R \phi_1]) \wedge (H \models E[\phi_2 R \phi_1]) & \\ \Rightarrow & (G \models \phi_1) \wedge (H \models \phi_1) & \{\text{definition of } E[\phi_2 R \phi_1]\} \\ \Rightarrow & (G \cap H) \models \phi_1 \text{ and } (G \cup H) \models \phi_1 & \{\text{meet- and join-closure of } \phi_1\} \end{array}$$

Recall that $E[\phi_2 R \phi_1] = E[\phi_1 U (\phi_1 \wedge \phi_2)] \vee EG(\phi_1)$.

- **Case 1:** Both $G$ and $H$ satisfy $E[\phi_1 U (\phi_1 \wedge \phi_2)]$.
  In the lattice $\mathcal{L}(\sigma)$, there exist finite paths $\pi$ and $\rho$, starting from $G$ and $H$ respectively, such that $\pi^{end} \models \phi_2$ and $\rho^{end} \models \phi_2$, where $\pi^{end}$ and $\rho^{end}$ are the final states on $\pi$ and $\rho$, respectively. From the meet- and join-closure of $\phi_2$, $(\pi^{end} \cap \rho^{end}) \models \phi_2$, and $(\pi^{end} \cup \rho^{end}) \models \phi_2$. We can construct a path $\lambda$ starting from $(G \cap H)$ as follows:

$$\lambda = G \cap H, G \cap \rho^1, G \cap \rho^2, ..., G \cap \rho^{end}, \pi^1 \cap \rho^{end}, \pi^2 \cap \rho^{end}, ..., \pi^{end} \cap \rho^{end}$$

From the properties of set intersection, for each $i$, $\lambda^i$ can either be the same as $\lambda^{i-1}$ or contain one additional event. Eliminating consecutive identical down-sets, we get a valid path in which for each $i$, $\lambda^i$ contains one event more than $\lambda^{i-1}$. From the meet-closure of $\phi_1$, it follows that $\lambda$ is a witness for $E[\phi_1 U(\phi_1 \wedge \phi_2)]$. Similarly, we can construct $\nu$ starting from $(G \cup H)$:

$$\nu = G \cup H, G \cup \rho^1, G \cup \rho^2, ..., G \cup \rho^{end}, \pi^1 \cup \rho^{end}, \pi^2 \cup \rho^{end}, ..., \pi^{end} \cup \rho^{end}$$

From the properties of set union, for each $i$, either $\nu^i$ can be the same as $\nu^{i-1}$, or contain one additional event. Eliminating consecutive identical down-sets, one obtains a valid path. From the join-closure of $\phi_1$, it follows that $\nu$ is a witness for $E[\phi_1 U(\phi_1 \wedge \phi_2)]$.

- **Case 2:** Either $G$ or $H$ satisfies $EG(\phi_1)$.

  WLOG, let $G \models EG(\phi_1)$. Let $\pi$ be a witness path starting from $G$, and $v$ be its corresponding transition sequence. We first show that there exists a finite $k \geq 0$ such that $H \subseteq \pi^k$. Let $s$ be the starting state of $\sigma$. Let $u$ and $w$ be transition sequences leading, respectively, from $s$ to $G$ and $s$ to $H$ in $\mathcal{L}(\sigma)$. Since $G \models EG(\phi_1)$, $u.v$ is a maximal transition sequence of $\sigma$, *i.e.*, $\sigma = [s, u.v]$. Therefore, $w \preceq u.v$. By the definition of $\preceq$, there exists a finite prefix $u'$ of $u.v$ such that $u' \equiv w'$ and $w$ is a prefix of $w'$. Let $K$ be the final state of the transition sequence $u'$. Recall that $H$ is the final state of the sequence $w$. Then, we have $H \subseteq K$. Now, $K$ can occur either before or after $G$ in the path corresponding to $u.v$. In either case, $K \subseteq \pi^k$ for some finite $k \geq 0$.

  We use the above property to construct a path $\lambda$ starting from $(G \cap H)$:

  $$\lambda = G \cap H, \pi^1 \cap H, \pi^2 \cap H, ...., (\pi^k \cap H = H)$$

  Eliminating consecutive identical down-sets, $\lambda$ becomes a valid path. Since $\pi$ is a witness for $G \models EG(\phi_1)$, every state along $\pi$ satisfies $\phi_1$. Also, $H \models \phi_1$. Thus, by the meet-closure of $\phi_1$, every state on $\rho$ satisfies $\phi_1$. Let $\rho$ be the witness path for $E[\phi_2 R \phi_1]$ starting from $H$. Then, the required witness path for $E[\phi_2 R \phi_1]$ from $(G \cap H)$ is given by $\lambda.\rho$.

  To demonstrate join-closure, we construct the following path $\nu$ starting from $(G \cup H)$:

  $$\nu = G \cup H, \pi^1 \cup H, \pi^2 \cup H, ....$$

  Removing consecutive identical down-sets, $\nu$ becomes a valid path. From the join-closure of $\phi_1$, it follows that $\nu$ is a witness path for $(G \cup H) \models EG(\phi_1)$.

The proof that $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ is regular is the same as Case 1 above. □

As $EF(\phi_1) = E[true\, U(true \wedge \phi_1)]$, and $EG(\phi_1) = E[false\ R\ \phi_1]$, we have[3]:

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

**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 $\phi_1$ and $\phi_2$ are CETL formulae, so are $(\phi_1 \wedge \phi_2)$, $E[\phi_2 R \phi_1]$, and $E[\phi_1 U(\phi_1 \wedge \phi_2)]$.*

**Definition 5.** *Let $\phi$ be a CETL formula. The set $sub(\phi)$ of subformulae of $\phi$ is defined as follows:*

- *If $\phi$ is a process-local state formula, or $true$ or $false$, then $sub(\phi) = \{\phi\}$.*
- *If $\phi$ is $\phi_1 \wedge \phi_2$, $E[\phi_2 R \phi_1]$ or $E[\phi_1 U(\phi_1 \wedge \phi_2)]$, then $sub(\phi) = \{\phi\} \cup sub(\phi_1) \cup sub(\phi_2)$.*

The length of a CETL formula $\phi$ is equal to the cardinality of $sub(\phi)$. We now explore the relation between meet-closure and *crucial events*.

[3] $true$ and $false$ are trivially meet- and join-closed.

### 3.1 Crucial events

Let $G$ be any down-set of a trace $\sigma = (E, \rightarrow)$. Let $\phi$ be some meet-closed formula, and $G \not\models \phi$. Let $\mathcal{G}$ be the set of all $\phi$-satisfying states that are reachable from $G$ in $\sigma$. That is:

$$\mathcal{G} = \{H \in \mathcal{L}(\sigma) | G \subseteq H \wedge H \models \phi\}$$

Note that $\mathcal{G}$ can be an infinite set. Let $\mathcal{H}$ consist of all the elements of $\mathcal{G}$ that are minimal under $\subseteq$. That is:

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

$\mathcal{H}$ is necessarily finite for finite-state programs. We now define:

$$K = \bigcap_{H \in \mathcal{H}} H$$

By the meet-closure of $\phi$, $K \models \phi$. Also, $G \subseteq K$. That is, $K$ is the unique and well-defined $\phi$-satisfying state that is reachable from $G$ by executing the fewest events. In other words, $K \setminus G$ gives us the minimum set of events that must be executed along any path starting from $G$, in order to reach a $\phi$-satisfying state from $G$ in $\sigma$. The events in $K \setminus G$ are called **crucial events** [10].

**Definition 6. Crucial event:** *In a trace $\sigma$, an event $e$ is said to be crucial from a state $G$ with respect to a meet-closed formula $\phi$, denoted $e \in crucial(G, \phi, \sigma)$ iff:*

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

In simple terms, a crucial event is one whose execution is necessary in order to reach a $\phi$-satisfying state from $G$ in $\sigma$.

A transition sequence starting from $G$ and comprising exactly of the events in $crucial(G, \phi, \sigma)$ gives us the shortest path from $G$ to a $\phi$-satisfying state in $\sigma$. 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. A crucial path is of particular interest in model checking, because it gives us the shortest witness path to a $\phi$-satisfying state. 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 $\phi$ 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 $\phi$-satisfying state in the full state space graph. The state $s$ can have multiple occurrences in $\sigma$ (for example, in Figure 1(c), the state $t$ occurs multiple times in $\sigma_2$). Let $G'$ be another down-set of $\sigma$ 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 $\sigma$ has the same set of crucial events w.r.t. $\phi$. Based on this observation, we define $crucial(s, \phi, \sigma) \overset{def}{\equiv} crucial(G, \phi, \sigma)$, where $G$ is any down-set of $\sigma$ 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 $\phi$ is an open problem. However, we can identify a *subset* of $crucial(s, \phi, \sigma)$ in most cases, as we shall see in Section 7.

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 two lemmas are used in the proofs presented in Sections 5.1 and 5.2, and are from [19].

**Lemma 1.** *[19] Given a trace $\sigma$, and states $C, D, F \in \mathcal{L}(\sigma)$, if $C \triangleright F$ and $D \subseteq F$, then $(C \cap D) \trianglerighteq D$.*

*Proof.* From the definition of $\triangleright$, $\exists e \notin C : C \cup \{e\} = F$. If $e \notin D$, then $D \subseteq C$, so $(C \cap D) = D$. If $e \in D$, then $(C \cap D) = D \setminus \{e\}$. That is, $(C \cap D) \triangleright D$. □

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

*Proof.* Let $C = F \cup \{e\}$. If $e \in D$, then $C \subseteq D$, so $(C \cup D) = D$. If $e \notin D$, then $C \cup D = D \cup F \cup \{e\}$. Since $F \subseteq D$, $(C \cup D) = D \cup \{e\}$, which implies $D \triangleright (C \cup D)$. □

We now show how the concepts presented so far can be used to prune the state space while model checking CETL formulae. We will start by presenting a "baseline" model checking algorithm in Section 4, then enhance it with a reduction technique that exploits lattice theory in Sections 5 and 6.

```
Procedure check_CETL(s, $\phi$)
    pre : $info(s, \phi)$ is $true$, $false$ or $\star$.
    post: $info(s, \phi) \neq \star$.
1   begin
2       if $info(s, \phi) \neq \star$ then return
3       if $\phi$ is a process-local state formula then
4           if $s \models \phi$ then $info(s, \phi) := true$
5           else $info(s, \phi) := false$
6       endif
7       if $\phi$ is $(\phi_1 \wedge \phi_2)$ then
8           $check\_CETL(s, \phi_1)$
9           if $info(s, \phi_1) = false$ then $info(s, \phi) := false$
10          else
11              $check\_CETL(s, \phi_2)$
12              $info(s, \phi) := info(s, \phi_2)$
13          endif
14      endif
15      if $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$ then
16          new stack($stk$)  /* Create a new stack, with stack id $stk$ */
17          $push(s, stk)$
18          $check\_EU\_ER(s, \phi, stk)$
19          $pop(stk)$
20      endif
21  end
```

## 4 Model checking CETL

Our baseline algorithm is a local model checking algorithm based on ALMC [20]. As in ALMC, we use a function $info()$, which uses a hash table to implement the function $info : S \times sub(\phi) \mapsto \{true, false, \star\}$.

$info()$ keeps track of all the subformulae evaluated so far at any state $s$. If $info(s, \phi_1) = \star$, then we do not yet know whether $s \models \phi_1$. If $\phi_1$ has already been evaluated at $s$, then $info(s, \phi_1) = true$ if $s \models \phi_1$, and $false$ otherwise. Of course, initially $info(s, \phi_1) = \star$ for all state-subformula pairs.

Procedure $check\_CETL()$ is the main routine of our model checking algorithm, and is self-explanatory for process-local state formulae (lines 3-6) and conjunctions (lines 7-14). For temporal subformulae, we offload the work to Procedure $check\_EU\_ER()$, passing it a clean stack to use for its state space search (lines 15-20). Procedure $check\_EU\_ER(s, \phi)$ performs a depth-first search starting from state $s$, with the stack $stk$ maintaining the current search path. The depth first search only explores the events returned by Procedure $ample(s, \phi)$ (line 15) from each state $s$. We call the set of events returned by Procedure $ample()$ an "ample set", which is a term borrowed from Peled's p.o. reduction technique [1]. In the non-reduced (baseline) case, $ample(s, \phi)$ is equal to $enabled(s)$.

We are interested in finding a witness for either $E[\phi_1 U(\phi_1 \wedge \phi_2)]$, or $EG(\phi_1)$ (if $\phi = E[\phi_2 R \phi_1]$). In either case, every state of the witness path must satisfy $\phi$, and also $\phi_1$. Consequently, we abandon the current search path (by backtracking) if we encounter a state $s'$ that either does not satisfy $\phi$ (line 3), or does not satisfy $\phi_1$ (lines 5-8).

The search stops with success in one of three cases: (1) a state satisfying $(\phi_1 \wedge \phi_2)$ is found (line 11), which is the final state of a witness path for $E[\phi_1 U(\phi_1 \wedge \phi_2)]$, or (2) some state $t$ satisfying $\phi$ is reached (line 3, 28-30), in which case we can transitively deduce that $s \models \phi$, or (3) if $\phi = E[\phi_2 R \phi_1]$, and a cycle is found consisting entirely of $\phi_1$-satisfying states (lines 18-24). If a witness is found, then we use the fact that *every* state on the witness path also satisfies $\phi$ to set $info(s', \phi) = true$ for every state $s'$ on the current search path (lines 28-31).

Note that $check\_EU\_ER(s, \phi)$ not only evaluates whether $s \models \phi$, but also evaluates the truth value of $\phi$ at every state visited during the search. This gives our baseline model checking algorithm an asymptotic time complexity that is linear in the length of the formula and the size of the full state space graph, similar to ALMC.

The baseline algorithm does not exploit any lattice-theoretic properties. We now show how we can use meet-closure to select only a *subset* of the enabled events at each state as our ample set. We start with the narrower problem of model-checking a CETL formula in a single trace of a program, then extend this approach to model checking the entire program.

```
Procedure check_EU_ER($s$, $\phi$, $stk$)
begin
    /* $\phi = E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$ */
    if $info(s, \phi) \neq \star$ then return
    $check\_CETL(s, \phi_1)$
    if $info(s, \phi_1) = false$ then
        $info(s, \phi) := false$
        return
    endif
    $check\_CETL(s, \phi_2)$
    if $info(s, \phi_2) = true$ then
        $info(s, \phi) := true$ /* $s \models (\phi_1 \wedge \phi_2)$ */
        return
    endif
    /* $s \models (\phi_1 \wedge \neg\phi_2)$ */
    $working\_set := ample(s, \phi)$
    for each $\alpha \in working\_set$ do
        $t := \alpha(s)$
        if $on\_stack(t, stk)$ then
            /* Found a cycle */
            if $\phi$ is $E[\phi_2 R \phi_1]$ then
                $info(s, \phi) := true$  /* Cycle demonstrates $EG(\phi_1)$ */
                return
            endif
        else
            $push(s, stk)$
            $check\_EU\_ER(t, \phi)$
            $pop(stk)$
            if $info(t, \phi) = true$ then
                $info(s, \phi) := true$  /* $(s \models \phi_1) \wedge (t \models \phi) \Rightarrow (s \models \phi)$ */
                return
            endif
        endif
    endfor
    $info(s, \phi) := false$ /* No successors satisfy $\phi$, backtrack */
    return
end
```

# 5 Model checking CETL formulae within a single trace

The following example highlights the basic principle of our approach. Let $\sigma = [s, v]$ be some trace, and $\phi$ a meet-closed formula. Suppose we are interested in finding out whether $s \models EF(\phi)$. If $s \models \phi$, then we are done. If $s \not\models \phi$, then there exists a crucial path for $\phi$ in $\sigma$, starting from $s$. Starting from $s$, we simply need to choose an ample set that consists of a single crucial event at each state in order to build this crucial path. This approach results in state space reduction by choosing a singleton ample set, and the crucial path built is the shortest witness for $s \models EF(\phi)$. In this section, we show how crucial paths can act as witnesses for the temporal operators in CETL.

## 5.1 $E[\phi_1 U(\phi_1 \wedge \phi_2)]$

Let $G_0$ be some down-set of $\sigma$ that satisfies $E[\phi_1 U(\phi_1 \wedge \phi_2)]$. Let $\pi$ 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 $\sigma$ that are reachable from $G_0$, are minimal under $\subseteq$[4], 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, $G \models (\phi_1 \wedge \phi_2)$.

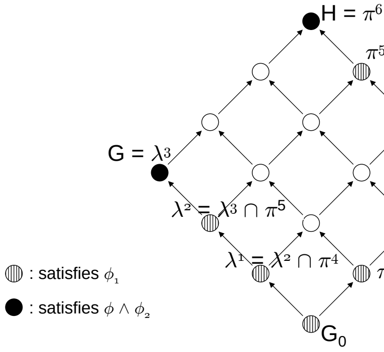

H = $\pi^6$
$\pi^5$
$\pi^4$
$\pi^3$
G = $\lambda^3$
$\lambda^2 = \lambda^3 \cap \pi^5$
$\pi^2$
$\lambda^1 = \lambda^2 \cap \pi^4$
$\pi^1$
G$_0$
▥ : satisfies $\phi_1$
● : satisfies $\phi \wedge \phi_2$

**Fig. 3.** Example illustrating the construction of Theorem 5

**Theorem 5.** *There exists a path from $G_0$ to $G$ such that every state along the path satisfies $\phi_1$.*

*Proof.* We will construct a path $\lambda$ from $G_0$ to $G$, consisting entirely of $\phi_1$-satisfying states. We construct this path backwards, starting from $\lambda^k = G$, towards $\lambda^0 = G_0$. Figure 3 illustrates this construction through an example.

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 $\lambda$ 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:

$$\lambda^i \not\subseteq \pi^{j-1} \tag{3}$$

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 violates (3). 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 $\phi_1$, $G' \models \phi_1$. □

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 $\phi_1$, 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.** *We can construct the path of Theorem 5 as follows. Starting from $G_0$, at each state $H$ we execute a single enabled event $\alpha$ that satisfies the following two conditions:*

- *$\alpha \in crucial(H, \phi_2, \sigma)$, and*
- *$H \cup \{\alpha\} \models \phi_1$.*

*Proof.* Let $G$ be as in (2). From Theorem 5, there exists *some* path $\lambda$ 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 $\phi_1$-satisfying successor.

Clearly, if every event along our path is crucial for $\phi_2$, then our path will lead to $G$. It remains to be shown 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 $G$.

[4] This ensures that $\mathcal{J}$ is finite

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:

$$\lambda^{i+1} \not\subseteq H \tag{4}$$

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 violates (4). 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 $\phi_1$, $\lambda^{i+1} \cup H \models \phi_1$. Hence, $J = \lambda^{i+1} \cup H$ is the required successor for $H$. □

### 5.2 $E[\phi_2 R \phi_1]$

Recall that $E[\phi_2 R\phi_1] \overset{def}{\equiv} E[\phi_1 U(\phi_1 \wedge \phi_2)] \vee EG(\phi_1)$. 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 $\sigma$. We can construct a witness path as follows. Starting from $G_0$, at each state $H$, we execute a single enabled event $\alpha$ 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 $\phi_1$-satisfying successor state. The proof for this is exactly the same as shown in Theorem 6. □

**Procedure** `ample_trace` ($s$, $\phi$)

```
1  begin
2      /* $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R\phi_1]$ */
3      $working\_set := find\_crucial(s, \phi_2, \sigma)$
4      for each $\alpha \in working\_set$ do
5          $t := \alpha(s)$
6          $check\_CETL(t, \phi_1)$
7          if $info(t, \phi_1) = true$ then return $\{\alpha\}$
8      endfor
9      return $enabled(s)$
10 end
```

Procedure $ample\_trace()$ constructs an ample set in accordance with Theorems 6 and 7. We assume the existence of a black-box function $find\_crucial(s, \phi_2, \sigma)$, which returns a (possibly empty) subset of $enabled(s) \cap crucial(s, \phi_2, \sigma)$. The implementation of this function is deferred till Section 7. In lines 4-8, we try to find some $\alpha$ such that $\alpha \in crucial(s, \phi_2, \sigma)$ and $\alpha(s) \models \phi_1$. If such an event is found, then it satisfies the requirements of both Theorems 6 and 7, so our ample set is a singleton consisting of this event. If such an event is not found, then we explore all enabled events (line 9). The following theorem is straightforward.

**Theorem 8.** *Procedure $ample\_trace()$ returns an ample set that is sufficient for model-checking CETL formulae in a single trace of a program.*

We now extend our approach beyond a single trace, to model checking a program.

## 6 Model checking CETL formulae in a program

For model checking CETL in a single trace, we simply needed to explore a single crucial path for the formula through the trace. We achieved this by exploring a crucial successor event at each state during our depth first search. To model check CETL in a program, we need to explore a crucial path in each maximal trace of the program. That is, at each state $s$ encountered during DFS, our ample set must contain a crucial event for every trace that starts from $s$.

Let $ample(s, \phi)$ denote the ample set at state $s$, for the CETL formula $\phi$. In [1], it was shown that if $ample(s, \phi)$ satisfies the following condition (C1), then it generates a successor in 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 9.** *[1] If $ample(s, \phi)$ satisfies condition (C1), then for every maximal trace $\sigma$ starting from $s$, there exists some $\alpha \in ample(s, \phi)$ such that $[s, \alpha] \sqsubseteq \sigma$.*

If $ample(s, \phi)$ satisfies (C1), then it contains a successor event for each trace starting from $s$. A single event $\alpha \in ample(s, \phi)$ can be a successor in *multiple* traces starting from $s$. For example, executing $\alpha$ may enable $\beta$ and $\gamma$, where $(\beta, \gamma) \in D$. Thus, $\alpha$ is a successor in both $[s, \alpha\beta]$ and $[s, \alpha\gamma]$. In order to construct a crucial path per maximal trace, $\alpha$ must be crucial in every trace in which it is a successor:

**Definition 7. Universally crucial event:** *An event $\alpha$ is said to be universally crucial from a state $s$ for a meet-closed formula $\phi_2$, denoted $\alpha \in ucrucial(s, \phi_2)$, iff for every trace $\sigma$ such that $[s, \alpha] \sqsubseteq \sigma$, $\alpha \in crucial(s, \phi_2, \sigma)$.*

Recall that Procedure $check\_EU\_ER(s, \phi, stk)$ calls Procedure $ample(s, \phi)$, passing it a formula $\phi$ of the form $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$. Procedure $ample(s, \phi)$ tries to construct an ample set that satisfies the following three conditions: (1) If $\alpha \in ample(s, \phi)$, then $\alpha \in ucrucial(s, \phi_2)$ (line 3), (2) If $\alpha \in ample(s, \phi)$, then $\alpha(s) \models \phi_1$ (lines 4-8), and (3) $ample(s, \phi)$ satisfies condition (C1) (line 9). If any of these conditions is violated, then $ample(s, \phi) = enabled(s)$ (lines 7, 10). We discuss the implementation of $find\_ucrucial()$ in the next section. For now, it suffices to say that $find\_ucrucial(s, \phi_2)$ returns a (possibly empty) subset of $enabled(s) \cap ucrucial(s, \phi_2)$. The function $satisfies\_C1()$ is the same as that used in the implementation of p.o. reduction in SPIN [12].

**Procedure** `ample(`$s$, $\phi$`)`

```
1  begin
2      /* $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$ */
3      $candidate := find\_ucrucial(s, \phi_2)$
4      for each $\alpha \in candidate$ do
5          $t := \alpha(s)$
6          $check\_CETL(t, \phi_1)$
7          if $info(t, \phi_1) = false$ then return $enabled(s)$
8      endfor
9      if ($candidate = \emptyset$) or ($\neg satisfies\_C1(candidate)$) then return $enabled(s)$
10     else return $candidate$
11 end
```

**Theorem 10.** *Procedure $ample()$ returns an ample set that is sufficient for model-checking CETL in a program.*

*Proof.* It is straightforward to see that if $check\_EU\_ER(s, \phi, stk)$ finds a witness path, then $s \models \phi$. We show the other direction. Assume $s \models \phi$, where $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$. Then, either $s \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$, or $s \models EG(\phi_1)$.

- **Case 1:** $s \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$. Let $\sigma$ be the maximal program trace to which the witness path belongs. By Theorem 5, there exists a *crucial* witness path for $s \models \phi$ in $\sigma$. We will construct a crucial witness path using only transitions in $ample(s, \phi)$.
  Let $u$ denote the transition sequence of the witness path constructed so far, and $s'$ be the final state reached after executing $u$ from $s$. In our construction, we will maintain the invariant that every event in $u$ is in $crucial(s, \phi_2, \sigma)$, and for every state $s'$ in the path, $s' \models \phi_1$. By Theorem 6, these two invariants ensure that at each state $s'$ along the constructed path, there exists some $\alpha \in enabled(s')$ such that $\alpha \in crucial(s', \phi_2, \sigma)$, and $\alpha(s') \models \phi_1$. We will show that $ample(s', \phi)$ contains such an event $\alpha$. Initially, $u := \epsilon$ (the empty string), and $s' := s$. Since $s \models \phi$, we know that $s \models \phi_1$.
    - **Case 1.1:** The candidate set picked in line 3 of Procedure $ample()$ does not satisfy (C1) or is empty. Then, $ample(s', \phi) = enabled(s')$ (line 9). As discussed in the previous paragraph, $enabled(s')$ must contain an event $\alpha$ that satisfies the two conditions of Theorem 6, so we set $u := u.\alpha$, and continue construction.
    - **Case 1.2:** The candidate set picked in line 3 is non-empty and satisfies (C1). We can express $\sigma$ as the concatenation $[s, u].\sigma'$, for some $\sigma'$. By Theorem 9, there exists some $\alpha \in ample(s', \phi)$ such that $[s', \alpha] \sqsubseteq \sigma'$. That is, $[s, u.\alpha] \sqsubseteq \sigma$. Since $\alpha$ is in $ucrucial(s', \phi_2) \cap enabled(s')$ (line 3), we have $\alpha \in crucial(s', \phi_2, \sigma)$, and $\alpha(s') \models \phi_1$, thus satisfying the conditions of Theorem 6. We set $u := u.\alpha$, and continue construction.

- **Case 2:** $s \not\models E[\phi_1 U(\phi_1 \wedge \phi_2)]$ and $s \models EG(\phi_1)$. Again, let $\sigma$ be the maximal program trace containing the witness path in the full state space graph. Using arguments identical to those in Case 1, we can show that $ample(s', \phi)$ always contains an event from $\sigma$ that satisfies the conditions of Theorem 7. Thus, we can construct a witness path using the technique of Theorem 7, with only the transitions returned by Procedure $ample()$.

□

Next, we provide an implementation for the function $find\_ucrucial()$, which is used by Procedure $ample()$.

## 7 Finding universally crucial events

In this section, we identify some sufficient conditions for an event to be universally crucial. Procedure $find\_ucrucial()$ takes as input a state $s$ and a CETL formula $\phi_2$, and returns a subset of $ucrucial(s, \phi_2) \cap enabled(s)$.

First, note that $find\_ucrucial(s, \phi_2)$ is called by our model checking routine only when $s \not\models \phi_2$. This assertion can be verified by navigating the procedure call chain of our model checking algorithm. Procedure $check\_EU\_ER(s, \phi)$ calls Procedure $ample(s, \phi)$ (line 22), where $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2 R \phi_1]$. The call to $ample(s, \phi)$ is only made after verifying that $s \not\models \phi_2$ (line 16). Procedure $ample(s, \phi)$ then calls $find\_ucrucial(s, \phi_2)$ (line 3). The following theorem both explains Procedure $find\_ucrucial()$ and shows its correctness.

**Procedure** find_ucrucial($s$, $\phi_2$)

```
input  : State $s$ and CETL formula $\phi_2$, where $s \not\models \phi_2$.
output: A subset of $ucrucial(s, \phi_2) \cap enabled(s)$.
1  begin
2      if $\phi_2$ is a process-local state formula on process $P_i$ then
3          return $enabled(s) \cap T_i$  /* $T_i$ is the set of transitions of $P_i$ */
4      endif
5      if $\phi_2$ is $(\psi_1 \wedge \psi_2)$ then
6          $check\_CETL(s, \psi_1)$
7          if $info(s, \psi_1) = false$ then  return $find\_ucrucial(s, \psi_1)$
8          else return $find\_ucrucial(s, \psi_2)$
9      endif
10     if $\phi_2$ is $E[\psi_1 U(\psi_1 \wedge \psi_2)]$ or $E[\psi_2 R \psi_1]$ then
11         $check\_CETL(s, \psi_1)$
12         if $info(s, \psi_1) = false$ then return $find\_ucrucial(s, \psi_1)$
13         else
14             if $\neg\psi_1$ is meet-closed then return $find\_ucrucial(s, \neg\psi_1)$
15             else return $\emptyset$
16         endif
17     endif
18 end
```

**Theorem 11.** *Procedure $find\_ucrucial(s, \phi_2)$ returns a subset of $ucrucial(s, \phi_2) \cap enabled(s)$.*

*Proof.* We show that Procedure $find\_ucrucial()$ returns only enabled, universally crucial events for each formula type.

- **(Lines 2-4):** $\phi_2$ is a process-local state formula on process $P_i$.
  Only transitions from $T_i$ can change the truth value of $\phi_2$. Since $s \not\models \phi_2$, in order to reach a $\phi_2$-satisfying state from $s$, we must execute some transition from $T_i \cap enabled(s)$. Now, for any $\alpha, \beta \in T_i \cap enabled(s)$, $(\alpha, \beta) \in D$. Further, execution of $\alpha$ disables $\beta$ and vice-versa. Therefore, each $\alpha \in T_i \cap enabled(s)$ is a crucial event in any trace that subsumes $[s, \alpha]$. Thus, $T_i \cap enabled(s) \subseteq ucrucial(s, \phi_2) \cap enabled(s)$.
- **(Lines 5-9):** $\phi_2 = \psi_1 \wedge \psi_2$.
  This case is straightforward. If $s \not\models \psi_1$, then clearly we first need to get to a state that satisfies $\psi_1$. Similarly for $\psi_2$. Therefore, if $s \not\models \psi_1$ then $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$, else $ucrucial(s, \psi_2) \subseteq ucrucial(s, \phi_2)$.

- **(Lines 10-17):** $\phi_2 = E[\psi_1 U(\psi_1 \wedge \psi_2)]$ or $\phi_2 = E[\psi_2 R \psi_1]$.
    - **(Line 12)**: $s \not\models \psi_1$. Clearly, any state that satisfies $\phi_2$ must satisfy $\psi_1$. Therefore, $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$.
    - **(Lines 13-16)**: $s \models \psi_1$. Let $t$ be some state reachable from $s$ such that $t \models \phi_2$. Let $w$ be a witness for $t \models \phi_2$. Since $s \not\models \phi_2$, along every path $v$ from $s$ to $t$, there must exist some state $s'$ such that $s' \not\models \psi_1$ (otherwise, $v.w$ would be a witness for $s \models \phi_2$). That is, we must first reach a state that satisfies $\neg\psi_1$ in order to reach any state that satisfies $\phi_2$. If $\neg\psi_1$ is meet-closed, then there exist crucial events for $\neg\psi_1$, so $ucrucial(s, \neg\psi_1) \subseteq ucrucial(s, \phi_2)$.

□

## 8 Experimental Results

We have implemented our approach as an extension to the SPIN model checker [12, 13], called SPICED (Simple PROMELA Interpreter with Crucial Event Detection). 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 [11]. The BEEM database is a large collection of benchmarks for explicit-state model checkers. The database includes PROMELA[5] models with errors injected into them, and lists the properties to be verified on these models. Of the 131 properties included in the BEEM database for verification, 101 (77%) can be expressed in CETL. All experiments were performed on a single-cpu 2.8 GHz Intel Pentium 4 machine with 512 MB of memory, running Red Hat Enterprise Linux WS Release 4.

Table 1 shows the results for the largest problem sizes, for each of the verified models. Table 1 only lists a subset of our results, due to space limitations. Altogether, we have performed experiments on over 80 instances of various models, with 75 of these instances containing errors. These results are available from our website: `http://maple.ece.utexas.edu/spiced`. Over all our experiments, SPICED produced error trails that were at least as short as SPIN's in 100% of the cases, were at least 10x shorter in 55% of the cases, and at least 100x shorter in 19% of the cases. For 44% of the cases, SPICED completed verification faster than SPIN, with at least a 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. Note that SPICED consistently achieves dramatic reductions in the size of the produced error trail, compared to SPIN with p.o. reduction. In many instances, this also results in a significant reduction in the number of states visited during verification, which in turn resulted in less memory consumption and faster run times.

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 [13], and have previously been used to showcase the effectiveness of p.o. reduction [21]. 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. As the results show, we achieve state space reduction comparable to p.o. reduction.

## 9 Conclusions and future work

In this paper, we have presented a model checking technique that produces short error trails, while simultaneously achieving state space reduction, for the exhaustive validation of programs. 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 is highly dependent on the ability to identify crucial events during state space exploration. We have shown how crucial events can be identified in many cases, but the problem of finding crucial events for a general CETL formula remains open. This is a direction for future research. We also intend to apply our approach to the verification of a larger, more diverse set of models to find further opportunities for improvement.

[5] PROMELA is the input language for SPIN.

| Model | Tool | Time (sec) | States | Memory (MB) | Formula | Trail length | Trail reduction factor |
|---|---|---|---|---|---|---|---|
| phils.7 | SPICED | 0.01 | 15 | 3.15 | $EF(P_0.req \wedge EG(!P_0.grant))$ | 6 | N/A |
| | 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 | N/A |
| | 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 | N/A |
| | 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 | 403.29 |
| | 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 | 382.79 |
| | 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 | 125.69 |
| | 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 | 44.33 |
| | 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 | 33.64 |
| | 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 | 18.61 |
| | 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 | 3.84 |
| | SPIN | 0.13 | 22386 | 5.5 | local assert() | 19396 | |
| needham.4 | SPICED | 0.01 | 27 | 2.72 | $EF(init0.fin \wedge resp0.fin)$ | 15 | 3.47 |
| | 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 | 3.4 |
| | 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 | 1.6 |
| | 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 | 1.38 |
| | SPIN | 0.01 | 85 | 2.62 | $\neg\Box(req0 \Rightarrow \Diamond grant0)$ | 170 | |
| frogs.3 | SPICED | 0.41 | 190318 | 16.45 | $EF(Check.done)$ | 261 | 1 |
| | SPIN | 0.38 | 190315 | 13.99 | local assert() | 261 | |

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

## References

1. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1994) 377–390
2. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Volume 1032. Springer-Verlag Inc., New York, NY, USA (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) (2004) 201–210
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, Springer-Verlag (2002) 112–127
6. Lafuente, A.L.: Symmetry reduction and heuristic search for error detection in model checking. In: Workshop on Model Checking and Artificial Intelligence. (2003)
7. 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, Springer-Verlag (1989) 285–363
8. 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, Springer-Verlag New York, Inc. (1987) 325–392
9. Mattern, F.: Virtual time and global states of distributed systems. In: Proc. of the International Workshop on Distributed Algorithms. (1989) 215–226
10. Chase, C.M., Garg, V.K.: Efficient detection of restricted classes of global predicates. In: WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms, London, UK, Springer-Verlag (1995) 303–317
11. Pelanek, R.: BEEM: BEnchmarks for Explicit Model checkers. http://anna.fi.muni.cz/models/index.html (2007)

| 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 2.** State space reduction with SPICED.

12. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
13. Holzmann, G.: On-the-fly LTL model checking with SPIN. http://spinroot.com/spin/ (2007)
14. Garg, V.K., Mittal, N., Sen, A.: Applications of lattice theory to distributed computing. ACM SIGACT Notes **34**(3) (2003) 40–61
15. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: OPODIS '03: Proceedings of the 7th International Conference on Principles of Distributed Systems, La Martinique, France (2003) 171–183
16. Davey, B., Priestley, H.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
17. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2001) 322
18. Kashyap, S., Garg, V.K.: Meet and join-closure of CTL operators. Technical Report TR-PDS-2008-001, ECE Dept, University of Texas at Austin, http://maple.ece.utexas.edu/TechReports/2008/TR-PDS-2008-001.pdf (2008)
19. Sen, A., Garg, V.K.: Detecting temporal logic predicates on the happened-before model. In: IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (2002) 76–83
20. Vergauwen, B., Lewi, J.: A linear local model checking algorithm for CTL. In: CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory, London, UK, Springer-Verlag (1993) 447–461
21. 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) (1999) 279–287