Automatic Generation of Computation Slices for Detecting Temporal Logic Predicates

Alper Sen and Vijay K. Garg

TR-PDS-2003-001 January 2003

Parallel and Distributed Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Automatic Generation of Computation Slices for Detecting Temporal Logic Predicates *

Alper Sen and Vijay K. Garg
Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX, 78712, USA
{sen,garg}@ece.utexas.edu
http://www.ece.utexas.edu/~{sen,garg}

Abstract

In this paper we present a novel approach for detecting temporal logic predicates. Predicate detection is a fundamental problem in distributed systems and finds applications in many domains such as testing, debugging, and software fault tolerance of distributed programs. However predicate detection suffers from state explosion, that is, the number of possible global states of the program increases exponentially with the number of processes.

To solve this problem, we generalize an effective abstraction technique called computation slicing*. In particular, we present algorithms for computing slices with respect to temporal logic predicates. A slice of a computation with respect to a predicate is the computation with the least number of global states that contains all global states of the original computation for which the predicate evaluates to true. The advantage of this technique is that, it ameliorates the state explosion problem by reasoning only on the part of the global state space that could potentially affect the predicate.*

We present polynomial-time algorithms to compute slices with respect to temporal logic predicates, more specifically, predicates that contain temporal operators $\mathbf{EF}$, $\mathbf{EG}$, *and* $\mathbf{AG}$. *Furthermore, we show that these slices contain precisely those global states of the original computation that satisfy the predicate. Finally, we use slices with respect to temporal logic predicates for detecting these predicates efficiently. In particular, we present a general framework for detecting predicates in a subset of the temporal logic CTL. This subset contains nested temporal predicates for which, to the best of our knowledge, there do not exist efficient (polynomial in the number of processes) predicate detection algorithms. We have implemented our technique and compared with the model checker NuSMV. Experimental results prove the viability of this new technique.*

Keywords: *predicate detection, testing, debugging, fault-tolerance, distributed programs, formal methods, computation slicing, temporal logic.*

* supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant

1 Introduction

Predicate detection is a fundamental problem in distributed systems and finds applications in many domains such as testing, debugging, and software fault tolerance of distributed programs.

The main problem in predicate detection is the *state explosion problem*—the set of possible global states of a distributed program with n individual processes can be of size exponential in n. Abstraction techniques are widely used to ameliorate this problem [4, 18, 19].

Computation slicing was introduced in [10, 16] as an abstraction technique for analyzing distributed computations. A *computation slice*, defined with respect to a global predicate, is the computation with the least number of global states that contains all global states of the original computation for which the predicate evaluates to true. Slicing can be used to throw away the *extraneous* global states of the original computation in an efficient manner, and focus on only those that are currently *relevant* for our purpose. The number of global states of the slice is much smaller than those of the computation. Therefore, in order to detect a fault, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice.

To illustrate what a computation slice looks like, consider the computation in Fig. 1(a) and suppose that we want to detect a predicate that holds in the computation if there exists a global state that satisfies the predicate $(2 \leq x \leq 4) \ \wedge \ (y \geq 2)$. In the computation, there are two processes P_1 and P_2 with integer variables x and y, respectively. The events are represented by solid circles. Process P_2 sends a message to process P_1 by executing event f_1 and process P_1 receives that message by executing event e_1. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of x immediately after executing e_1 is 2. The first event on each process initializes the state of the process and every global state contains these initial events. Without computation slicing, we are forced to examine all global states of the computation, thirteen in total, to decide whether the computation satisfies the predicate. Alternatively, we can compute a slice of the computation with respect to $(2 \leq x \leq 4) \ \wedge \ (y \geq 2)$ as shown in Fig. 1(b). This slice is modeled as a directed graph, each vertex of which corresponds to a subset of events. If a vertex is contained in a global state then all events corresponding to the vertex are contained in the global state. A vertex belongs to a global state only if all its incoming neighbors are also present in the global state. We can now restrict our search to the global states of the slice which are only four in number, namely $\{e_1, e_0, f_2, f_1, f_0\}$, $\{e_1, e_0, f_3, f_2, f_1, f_0\}$, $\{e_2, e_1, e_0, f_2, f_1, f_0\}$, $\{e_2, e_1, e_0, f_3, f_2, f_1, f_0\}$. The slice has much fewer global states than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

The concept of slicing is useful for detecting temporal logic predicates since it enables us to reason only on the part of the global state space that could potentially affect the predicate. Many specifications of distributed programs are temporal in nature because we are interested in properties related to the sequence of states during an execution rather than just the initial and final states. For example, the liveness property in dining philosophers problem, "a philosopher, whenever gets hungry, eventually gets to eat", is a temporal property. We use the temporal logic CTL [7] to write specifications of distributed programs and interpret it on a finite distributive lattice of global states.

We identify a *regular* subset of CTL that allows provably efficient predicate detection. Towards this goal, we use the class of predicates, called *regular predicates*, which was introduced in [10]. The slice with respect to a regular predicate contains *precisely* those global states for which the predicate evaluates to true. Some examples of regular predicates are monotonic channel predicates [8], conjunction of local predicates [8, 12], and some relational predicates [8]. We prove that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular predicates when p is a regular predicate. We also prove that temporal predicates $\mathbf{AF}(p)$, $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$, and $\mathbf{AU}(p, q)$, in general, are not regular predicates when p and q are regular predicates. However the set of global states that satisfy $\mathbf{AF}(p)$ is closed under union when the set of global states that satisfy p is closed

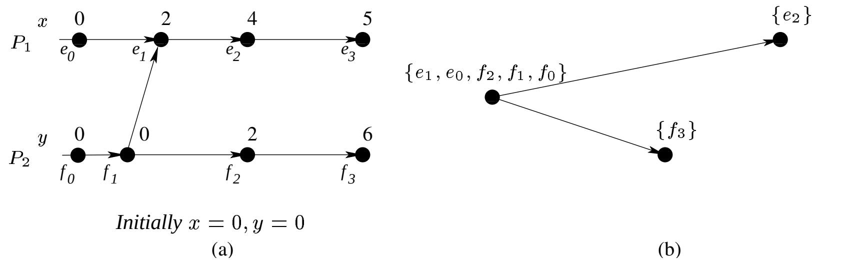

Figure 1. (a) A computation and (b) its slice with respect to $(2 \leq x \leq 4) \wedge (y \geq 2)$

under union.

We present polynomial-time algorithms to compute slices with respect to regular temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$, where p is a regular predicate. Our computation slicing algorithms have complexity $O(n|E|)$, where E is the set of events and n is the number of processes. Furthermore, every global state of slices generated by these algorithms satisfies the temporal predicate.

Finally, we show how our computation slicing algorithms can be used for temporal logic predicate detection. Specifically, we focus on a regular subset of the temporal logic CTL, where temporal operators are $\mathbf{EF}$, $\mathbf{EG}$, $\mathbf{AG}$ and atomic propositions are non-temporal regular predicates. CTL has mostly been used in the context of model checking where detection algorithms have exponential complexity in the number of processes. We present a polynomial-time predicate detection algorithm for this subset of CTL that has complexity $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in a given CTL predicate p. In this subset of CTL we can specify *nested temporal logic predicates*. To the best of our knowledge, there do not exist efficient algorithms (polynomial in the number of processes) to detect predicates which contain nested temporal logic predicates. An example of a nested predicate is $\mathbf{AG}(\mathbf{EF}(reset))$, which states that reset is possible from every state.

In summary, this paper makes the following contributions:

- We advocate the use of computation slicing approach for temporal logic predicate detection. To this end, we extend computation slicing to include algorithms for such predicates. This approach allows us to detect nested temporal logic predicates efficiently.

- We identify a regular subset of temporal logic CTL that contains predicates for which the slices contain precisely those global states that satisfy the predicate. In particular, we prove that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular predicates, whereas $\mathbf{AF}(p)$, $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$, and $\mathbf{AU}(p, q)$, in general, are not regular predicates when p and q are regular predicates.

- We present polynomial-time algorithms to compute slices with respect to temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$, where p is a regular predicate.

- As an application, we show how to use computation slicing in predicate detection for the regular subset of temporal logic CTL, which includes nested temporal logic predicates, and present polynomial-time algorithms to accomplish this.

The remainder of this paper is organized as follows. In the next section, we compare our work with related work in this area. In Section 3, we discuss the computation model that is used to represent distributed programs. A brief overview of computation slicing is given in Section 4. Temporal regular predicates and automatic generation of slices for these predicates are described in Sections 5 and 6, respectively. Section 7 discusses the application of our algorithms in predicate detection. We give experimental results in Section 8. Finally, some concluding remarks are given in Section 9.

2 Related Work

Our method differs from *model checking* [7, 13] in many aspects. First, model checking algorithms check whether a predicate is satisfied for all computations of a program, whereas we ascertain whether a predicate is satisfied for a *single* computation of a program. This is because our objective is to develop fast algorithms for fault tolerance and debugging of distributed programs where a single execution trace of the program is observed. Second, even if model checking algorithms are used on a single computation of a program with a finite global state space as in our case, the complexity of detecting a temporal predicate would be, in general, proportional to the size of the global state space which is still exponential in the number of processes (state explosion problem), whereas our algorithms have polynomial-time complexity. Finally, we focus on specific predicate classes as mentioned above, whereas model checking deals with arbitrary predicates.

The notion of a computation slice is similar to the concept of a program slice [20]. Given a program and a slicing criterion, that is, a set of variables, a program slice consists of all statements in the program that may affect the value of the variables in the set at some given point. The criterion in program slicing has also been extended to some predicate classes such as non-temporal propositions in temporal logic LTL [6]. Program slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance [14, 20]. A slice can significantly narrow the size of the program to be analyzed, thereby making the understanding of the behavior easier. We expect to have the same benefit from a computation slice for predicate detection and other problems.

Predicate detection is a widely-studied problem. There are three major approaches to solving predicate detection: global-snapshot-based approach [1], global-state-space-construction-based approach (including model checking) [7, 5], and predicate-restriction-based approach [8]. The first approach can detect only *stable* predicates (which remain true once they become true), and the second approach suffers from the state explosion problem. A variety of strategies for improving the state explosion problem such as data abstraction, symbolic and partial-order model checking have been explored [4, 18, 19]. We follow the predicate-restriction-based approach that exploits the structure of the predicate and directly uses the computation to detect if the predicate is satisfied in a global state. Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [8, 12], *disjunctive* [8], *stable* [1], *observer-independent* [2, 8], *linear* [8, 17], and *non-temporal regular* [10, 16] predicates. These predicate classes have been so far detected under some or all of the temporal operators $\mathbf{EF}$, $\mathbf{EG}$, $\mathbf{AG}$, $\mathbf{AF}$ and recently under the *until* operator of CTL [17], but not under any nesting of these operators. For example, a predicate $\mathbf{EF}(p \wedge \mathbf{EG}(q))$, where p and q are conjunctive predicates, cannot be detected using only the algorithms for conjunctive predicates. In this paper we can detect such nested temporal logic predicates.

3 Model

We assume a loosely-coupled message-passing asynchronous system without any shared memory or a global clock. A *distributed program* consists of n sequential processes denoted by $P_1, P_2, \ldots, P_n$ communicating via asynchronous messages. In this paper, we are concerned with a single *computation* (*execution*) of a distributed program. We assume that no messages are altered or spuriously introduced. We do not make any assumptions about FIFO nature of channels.

The execution of a process in a distributed computation can be viewed as a sequence of events. We call such a sequence a *local computation*. Each event is either a *send event*, a *receive event*, or an *internal event*. A send event causes a message to be sent and a receive event causes a message to be received.

We model a *distributed computation* (or simply a *computation*), denoted by $\langle E, \rightarrow \rangle$, as a directed graph with vertices as the set of events E and edges as $\rightarrow$. We use event and vertex interchangeably. We assume that the edge relation of the computation is transitive. We would like to handle distributed computations and slices in a convenient and uniform manner. Therefore, we model both using a directed graph, instead of using the traditional model of a distributed computation which is a partial order on a set of events [15].

Let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote the set of vertices and edges of a directed graph G. We assume that each vertex has a path to itself. We define a *consistent cut* (global state) on directed graphs as a subset of vertices of a directed graph such that if the subset contains a vertex then it contains all its incoming neighbors. Formally, C is a consistent cut of G, if $\forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) \wedge (f \in C) \Rightarrow (e \in C)$. From this definition it follows that if a strongly connected component is part of a consistent cut then all vertices in it are contained in the cut. We say that a strongly connected component is *non-trivial* if it has more than one vertex. We denote the set of consistent cuts of a directed graph G by $\mathcal{C}(G)$. Observe that the empty set $\emptyset$ and the set of vertices $\mathsf{V}(G)$ trivially belong to $\mathcal{C}(G)$. We call them *trivial* consistent cuts.

We use $\sqcup$ and $\sqcap$ to denote the join and meet operators of a lattice, L. We say that a lattice is *distributive* if meet distributes over join, i.e. $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

The set of consistent cuts of a directed graph forms a distributive lattice under the relation $\subseteq$ [16]. We denote this lattice by $L = (\mathcal{C}(G), \subseteq)$. Note that the join (resp. meet) of two consistent cuts correspond to their set union (resp. set intersection).

We assume the presence of a fictitious *global initial* and a *global final event*, denoted by $\perp$ and $\top$, respectively. The global initial event occurs before any other event on the processes and initializes the state of the processes. The global final event occurs after all other events on the processes. Any non-trivial consistent cut will contain the global initial event and not the global final event. Therefore, every consistent cut of a computation in traditional model is a non-trivial consistent cut of the computation in our model and vice versa. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. Note that the empty consistent cut, $\emptyset$, in the traditional model corresponds to $\{\perp\}$ in our model and the final consistent cut, E, in the traditional model corresponds to $E - \{\top\}$ in our model and we denote this by $\mathcal{E}$. We use uppercase letters C, D, H, V, and W to represent consistent cuts.

The predecessor and successor events of an event e on the process on which e occurs are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist. A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally, $frontier(C) \triangleq \{ e \in C \mid succ(e) \text{ exists} \Rightarrow succ(e) \notin C\}$. A consistent cut is uniquely characterized by its frontier and vice versa.

Figure 2 shows a computation and its lattice of (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $C = \{e_3, e_2, e_1, f_2, f_1, \perp\}$ is represented by $\{e_3, f_2\}$.

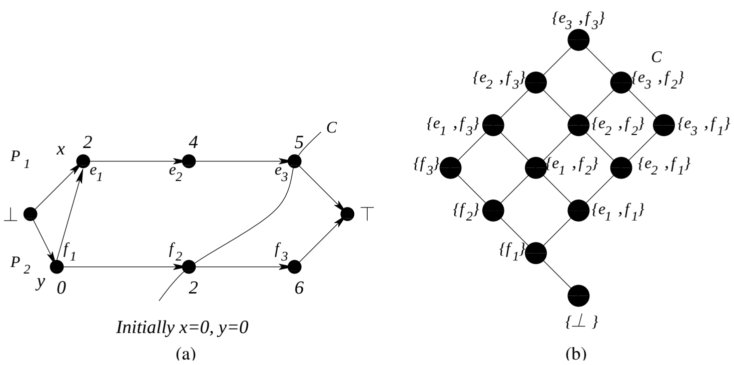

Figure 2. (a) A computation $\langle E, \rightarrow \rangle$ (b) and its lattice corresponding to $\mathcal{C}(G)$

To specify the properties of a distributed program we use the temporal logic CTL. The syntax and semantics of CTL is given in the appendix.

4 Background on Slicing

In this section, we present results from previous work on computation slicing [10, 16]. In the next section we show how to extend these results so that we can generate slices with respect to temporal logic predicates.

We define the notion of slice of a computation with respect to a predicate. A slice of a computation exists with respect to every predicate but the slice may include consistent cuts which do not satisfy the predicate.

Definition 1 (slice [16]) *A slice of a computation with respect to a predicate is the smallest computation (with minimum number of consistent cuts) that contains all consistent cuts of the original computation that satisfy the predicate.*

It is shown in [16] that the smallest computation exists and is unique for every predicate. We denote the slice of a computation $\langle E, \rightarrow \rangle$ with respect to a predicate p by $\langle E, \rightarrow \rangle_p$. Note that $\langle E, \rightarrow \rangle = \langle E, \rightarrow \rangle_{true}$. We use computation, slice, and directed-graph interchangeably.

Figure 3(a) shows a slice of the computation $\langle E, \rightarrow \rangle$ in Fig. 2(a) with respect to the conjunctive predicate $(2 \leq x \leq 4) \wedge (y \geq 2)$. Figure 3(b) shows the lattice of (non-trivial) consistent cuts of the slice, in which only the solid circles are elements of the lattice. It can be observed from the figure that this lattice is a sublattice of the one in Fig. 2(b). The slice has 4 strongly connected components $\{e_1, f_2, f_1, \bot\}$, $\{e_2\}$, $\{f_3\}$, and $\{e_3, \top\}$. The component $\{e_1, f_2, f_1, \bot\}$ corresponds to the initial consistent cut where the predicate holds. Event e_3 is in the same component as the global final event $\top$, therefore any consistent cut that includes e_3 is a trivial consistent cut. In other words, e_3 does not occur in any consistent cut that satisfies the predicate.

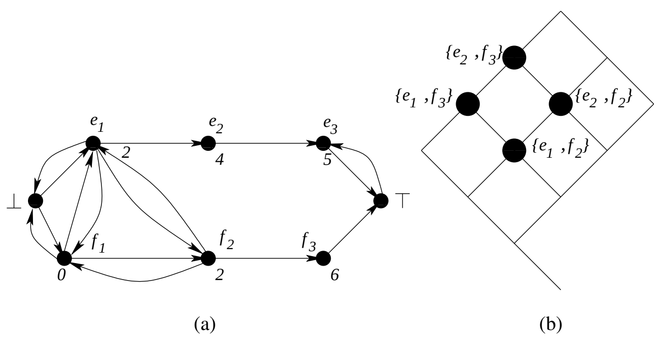

Figure 3. (a) The slice of $\langle E, \rightarrow \rangle$ in Fig. 2 with respect to the conjunctive predicate $p = (2 \leq x \leq 4) \wedge (y \geq 2)$ (without self-loops) (b) the corresponding sublattice

In our model, every slice derived from a computation $\langle E, \rightarrow \rangle$ will have the trivial consistent cuts $\emptyset$ and E among its set of consistent cuts. Consequently, a slice is *empty* if and only if it has no non-trivial consistent cuts. A slice of a computation with respect to a predicate is *lean* if and only if the slice is a computation that contains *those and only those* consistent cuts of the original computation that satisfy the predicate.

We use the following definitions of regular predicates in computation slicing.

Definition 2 (regular predicate [10]) *A predicate is regular if the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts.*

Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union will also satisfy the predicate. From the definition of a regular predicate we deduce that a regular predicate has a least satisfying cut and a greatest satisfying cut. Observe also that the class of regular predicates is closed under conjunction. Garg and Mittal [10] show that the slice of a computation with respect to a predicate is lean if and only if the predicate is regular.

Definition 3 (non-temporal regular predicate) *A predicate is a non-temporal regular predicate if it is a regular predicate and does not contain any temporal operators such as* $\mathbf{EF}$*,* $\mathbf{EG}$*,* $\mathbf{AG}$*,* $\mathbf{AF}$*,* $\mathbf{EX}$*,* $\mathbf{AX}$*,* $\mathbf{EU}$*, and* $\mathbf{AU}$*.*

Some examples of non-temporal regular predicates are monotonic channel predicates such as "there are at least k messages in transit from P_i to P_j", conjunction of local predicates, and relational predicates such as $x_1 - x_2 \leq 5$, where x_i is a monotonically non-decreasing integer variable on process i. We say that a predicate is a *temporal regular predicate* if it is a regular predicate and contains temporal operators such as $\mathbf{EF}$, $\mathbf{EG}$, $\mathbf{AG}$, $\mathbf{AF}$, $\mathbf{EX}$, $\mathbf{AX}$, $\mathbf{EU}$, and $\mathbf{AU}$. In the next section, we prove that if p is a regular predicate then $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are temporal regular predicates.

Given a computation $\langle E, \rightarrow \rangle$, a regular predicate p and an event e, let $J_p(e)$ denote the least consistent cut of $\langle E, \rightarrow \rangle$ that contains e and satisfies p. If $J_p(e)$ does not exist then it is set to the trivial consistent cut E. For example, in Fig. 3, $J_p(e_1) = J_p(\bot) = \{e_1, f_2, f_1, \bot\}$ and $J_p(e_3) = J_p(\top) = \{\bot, \top, e_0, e_1, e_2, f_0, f_1, f_2\} = E$.

The results in [10] establish that it is sufficient to know $J_p(e)$ for each event e in order to recover the slice. A directed graph with E as the set of vertices and an edge from an event e to an event f iff $J_p(e) \subseteq J_p(f)$ has the same set of consistent cuts as the slice $\langle E, \rightarrow \rangle_p$. An $O(n^2|E|)$ algorithm to compute slices for *non-temporal regular predicates* is also given in [10].

Also in [10] polynomial-time algorithms are given to compute slices with respect to boolean combination of regular predicates. Given the slices with respect to regular predicates, the complexities of computing the slice for the conjunction, disjunction of regular predicates and negation of a regular predicate are $O(n^2|E|)$, $O(n^2|E|)$, and $O(n^2|E|^2)$, respectively. Note that regular predicates are not closed under disjunction and negation operators therefore slices obtained with respect to predicates that contain these operators may not be lean.

Observe that if the initial consistent cut of a computation is the same as the initial consistent cut of the slice of the computation with respect to a regular predicate, then we say that the computation satisfies the predicate, otherwise, it does not. We can use this observation to enable computation slicing for predicate detection.

5 Temporal Regular Predicates

In this section, we study regularity of a predicate p when temporal operators $\mathbf{EF}$, $\mathbf{AG}$, and $\mathbf{EG}$ are applied to it.

Lemma 1 *If p is a regular predicate then $\mathbf{EF}(p)$, $\mathbf{AG}(p)$, and $\mathbf{EG}(p)$ are regular predicates.*

Proof Sketch: Our goal is to prove that for all consistent cuts D, H if D, H satisfy $\mathbf{EF}(p)$ (resp. $\mathbf{AG}(p)$, $\mathbf{EG}(p)$) then $(D \sqcap H)$ and $(D \sqcup H)$ satisfy $\mathbf{EF}(p)$ (resp. $\mathbf{AG}(p)$, $\mathbf{EG}(p)$).

• $\mathbf{EF}(p)$ is a regular predicate:

From the definition of $\mathbf{EF}(p)$, there exists consistent cuts D', H' which satisfy p and $D \subseteq D'$ and $H \subseteq H'$. Furthermore $D' \subseteq W$ and $H' \subseteq W$ where W is the greatest cut satisfying p. Then we have $(D \sqcap H) \subseteq W$ and $(D \sqcup H) \subseteq W$. Therefore $(D \sqcap H)$ and $(D \sqcup H)$ satisfy $\mathbf{EF}(p)$.

• $\mathbf{AG}(p)$ is a regular predicate:

From the definition of $\mathbf{AG}(p)$, for all consistent cuts D' such that $D \subseteq D' \subseteq \mathcal{E}$, D' satisfies $\mathbf{AG}(p)$. Then substituting $(D \sqcup H)$ for D' in the previous observation we have $D \subseteq (D \sqcup H) \subseteq \mathcal{E}$. Thus $(D \sqcup H)$ satisfies $\mathbf{AG}(p)$. Now we show that $(D \sqcap H)$ satisfies $\mathbf{AG}(p)$. Consider an arbitrary consistent cut J such that $(D \sqcap H) \subseteq J$. Since $D \subseteq (D \sqcup J) \subseteq \mathcal{E}$ and $H \subseteq (H \sqcup J) \subseteq \mathcal{E}$, both $(D \sqcup J)$ and $(H \sqcup J)$ satisfy p. It is easy to show that $J = (D \sqcup J) \sqcap (H \sqcup J)$ satisfies p. Therefore $(D \sqcap H)$ satisfies $\mathbf{AG}(p)$.

• $\mathbf{EG}(p)$ is a regular predicate:

From the definition of $\mathbf{EG}(p)$, there exist consistent cut sequences $(D_0 = D), \ldots, (D_k = \mathcal{E})$ and $(H_0 = H), \ldots, (H_m = \mathcal{E})$ such that p is satisfied at all cuts of the sequences. Since p is a regular predicate, $(H_0 \sqcup D_0), (H_0 \sqcup D_1), \ldots, (H_0 \sqcup D_k)$ and $(H_0 \sqcap D_0), (H_0 \sqcap D_1), \ldots, (H_0 \sqcap D_{k-1}), (H_0), \ldots, (H_m)$ are also such sequences. Therefore $(D \sqcap H)$ and $(D \sqcup H)$ satisfy $\mathbf{EG}(p)$. ■

The set of global states that satisfy the temporal predicate $\mathbf{AF}(p)$ is closed under union (shown in the appendix) but not under intersection. We give an example in Figure 4 where consistent cuts D and H satisfy $\mathbf{AF}(p)$ but their intersection $(D \sqcap H)$ does not. This is because there exists a path starting from $(D \sqcap H)$ and ending at the final cut $\mathcal{E}$ where p never holds on the path. Similarly, we show in the appendix that $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$ may not be regular when p and q are regular predicates.

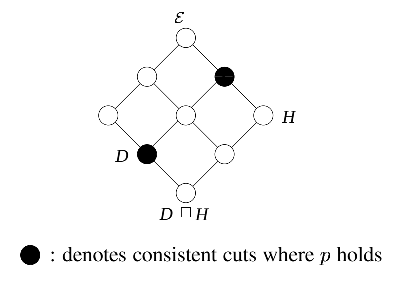

Figure 4. $\mathbf{AF}(p)$ may not be regular when p is regular

From above discussion and the previous section we can formulate a subset of CTL, called *regular CTL*, that contains only regular predicates as follows: the subset contains temporal operators $\mathbf{EF}$, $\mathbf{AG}$, $\mathbf{EG}$, the boolean operator $\wedge$ and atomic propositions are non-temporal regular predicates.

6 Algorithms for Computing Slices with respect to Temporal Predicates

In this section, we present polynomial-time algorithms to compute slices of computations with respect to temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$.

Given a slice $\langle E, \rightarrow\rangle_p$, where p is a regular predicate, applying the temporal operator $\mathbf{EF}$ on this slice corresponds to the slice of the computation $\langle E, \rightarrow\rangle$ with respect to the predicate $\mathbf{EF}(p)$. Specifically, we can compute the smallest computation that includes all consistent cuts of the original computation that satisfy $\mathbf{EF}(p)$ and denote this by a slice $\langle E, \rightarrow\rangle_{\mathbf{EF}(p)}$.

Although $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular predicates, we cannot use the polynomial-time algorithms in [10, 16] to generate slices of these temporal predicates because those algorithms are for generating slices with respect to *non-temporal* regular predicates.

Observe that $\langle E, \rightarrow\rangle_{\mathbf{EF}(p)}$, when p is a regular predicate, is a lean slice because $\mathbf{EF}(p)$ is a regular predicate (as shown in previous section), similarly for $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$ and $\langle E, \rightarrow\rangle_{\mathbf{EG}(p)}$.

6.1 Computing Slices with respect to $\mathbf{EF}(p)$

Algorithm A1 in Fig. 5 generates the slice of the original computation with respect to $\mathbf{EF}(p)$. Intuitively, the slice $\langle E, \rightarrow\rangle_{\mathbf{EF}(p)}$ is the same as the original computation up to the last consistent cut that satisfies p, call this cut W. This is because every consistent cut that is a subset of W satisfies $\mathbf{EF}(p)$. The algorithm finds W from the slice $\langle E, \rightarrow\rangle_p$ and then puts successors of events in the frontier of W into a strongly connected component with $\top$. Therefore any consistent cut that includes these successor events is a trivial consistent cut. This ensures that W is the final consistent cut of the slice $\langle E, \rightarrow\rangle_{\mathbf{EF}(p)}$. Note that W exists if the slice $\langle E, \rightarrow\rangle_p$ is non-empty since a regular predicate p has a greatest satisfying cut.

Algorithm A1

Input:	A computation $\langle E, \rightarrow\rangle$ and its slice $\langle E, \rightarrow\rangle_p$
Output:	Slice $\langle E, \rightarrow\rangle_{\mathbf{EF}(p)}$
Step 1.	Let G be $\langle E, \rightarrow\rangle$ and let W be the final cut of $\langle E, \rightarrow\rangle_p$
Step 2.	**If** W exists **then**
Step 3.	$\forall\, e \in frontier(W)$: add an edge from the vertex $\top$ to $succ(e)$ in G
Step 4.	**return** G
Step 5.	**else return** empty slice

Algorithm A2

Input:	A computation $\langle E, \rightarrow\rangle$ and its slice $\langle E, \rightarrow\rangle_p$
Output:	Slice $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$
Step 1.	Let G be $\langle E, \rightarrow\rangle_p$
Step 2.	For each pair of vertices (e, f) in G such that, (i) $\neg(e \rightarrow f)$ in $\langle E, \rightarrow\rangle$, and (ii) $(e \rightarrow f)$ in G add an edge from vertex e to the vertex $\perp$
Step 3.	**return** G

Algorithm A3

Input:	A computation $\langle E, \rightarrow\rangle$ and its slice $\langle E, \rightarrow\rangle_p$
Output:	Slice $\langle E, \rightarrow\rangle_{\mathbf{EG}(p)}$
Step 1.	Let G be $\langle E, \rightarrow\rangle_p$
Step 2.	For each pair of vertices (e, f) in G such that, (i) $\neg(e \rightarrow f)$ in $\langle E, \rightarrow\rangle$, and (ii) $(e \rightarrow f)$ and $(f \rightarrow e)$ in G add an edge from vertex e to the vertex $\perp$
Step 3.	**return** G

Figure 5. Algorithms for generating a slice with respect to $\mathbf{EF}(p)$, $\mathbf{AG}(p)$, and $\mathbf{EG}(p)$

We give an application and the proof of the correctness of the algorithm in the appendix.

Complexity of Algorithm A1: In Step 1 we can find the final cut of the slice $\langle E, \rightarrow\rangle_p$ using the strongly connected components of the slice and this has complexity $O(n|E|)$. Steps 2 to 4 take $O(n)$ time. The overall complexity is $O(n|E|)$.

6.2 Computing Slices with respect to $\mathbf{AG}(p)$

A slice of a computation, in general, has *additional edges* which do not exist in the computation itself. In fact, such edges are the reason for less number of consistent cuts in the slice compared to the original computation. For example, (f_2, e_3) in Fig. 6(a) is an additional edge. This edge exists in the slice because the consistent cut $\{e_3, e_2, e_1, f_1, \perp\}$ (as observed from Fig. 6(b)) does not satisfy p. These additional edges are of interest to us in developing algorithms for applying temporal operators $\mathbf{EG}$ and $\mathbf{AG}$ on slices.

Every consistent cut that satisfies $\mathbf{AG}(p)$ satisfies p. Therefore, the set of consistent cuts of the slice $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$ is a subset of the set of consistent cuts of the slice $\langle E, \rightarrow\rangle_p$. Consequently, we can use $\langle E, \rightarrow\rangle_p$ to obtain $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$.

Observe that consistent cuts of the slice $\langle E, \rightarrow\rangle_p$ that do not include vertex e of each additional edge (e, f) do not satisfy $\mathbf{AG}(p)$. For simplicity, let the slice $\langle E, \rightarrow\rangle_p$ have a single additional edge (e, f). For example, consistent cuts $\{\perp\}$, $\{f_1, \perp\}$, $\{e_1, f_1, \perp\}$, and $\{e_2, e_1, f_1, \perp\}$ of the slice in Fig. 6(a) do not include vertex f_2 of the additional edge (f_2, e_3). It is easy to see that these consistent cuts do not satisfy $\mathbf{AG}(p)$ and therefore cannot exist in the slice with respect to $\mathbf{AG}(p)$.

We now give a proof sketch to the claim that a consistent cut D satisfies $\mathbf{AG}(p)$ iff it includes vertex e of the additional edge (e, f) in $\langle E, \rightarrow\rangle_p$. If a consistent cut D does not include vertex e then there exists a consistent cut H that can be reached from D in the original computation such that H does not include e but includes f. In this case, it is clear that H does not satisfy p since (e, f) is an edge in the slice $\langle E, \rightarrow\rangle_p$ and every consistent cut of $\langle E, \rightarrow\rangle_p$ that includes f must include e. Therefore from the definition of $\mathbf{AG}(p)$, D does not satisfy $\mathbf{AG}(p)$. Now we prove the other direction. If a consistent cut D does not satisfy $\mathbf{AG}(p)$ then there exists a consistent cut H reachable from D such that H does not satisfy p. We know that only the consistent cuts that include f but not e do not satisfy p. Since H is reachable from D and H does not include e, we have that D also does not include e.

Algorithm A2 in Fig. 5 generates the slice of the original computation with respect to $\mathbf{AG}(p)$. In the algorithm, for any additional edge (e, f), we add an edge from the vertex e to the vertex $\perp$. This ensures that consistent cuts which do not include vertex e of any additional edge (e, f) are disallowed from the slice $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$. For example, consistent cut $\{e_1, f_1, \perp\}$ of the slice in Fig. 6(a) does not include vertex f_2 of the additional edge (f_2, e_3) in Fig. 6(a), therefore we add an edge $(f_2, \perp)$ and obtain the slice in Fig. 6(c). The cut $\{e_1, f_1, \perp\}$ cannot be a consistent cut of this new slice since it has to include vertex f_2.

Observe that at the end of Step 2, a strongly connected component that includes $\perp$ is obtained. We also know that an initial cut of a computation, if it exists, contains at least the event $\perp$. We prove in the appendix that this strongly connected component is the initial cut of $\langle E, \rightarrow\rangle_{\mathbf{AG}(p)}$ and furthermore is also the least consistent cut, V, that satisfies $\mathbf{AG}(p)$. It can be observed that $V = \{f_2, f_1, \perp\}$ in Fig. 6(c).

Note that in Algorithm A2 we obtain the output using $\langle E, \rightarrow\rangle_p$, whereas in Algorithm A1 we obtain the output using $\langle E, \rightarrow\rangle$. We present the proof of correctness of the algorithm in the appendix.

Complexity of Algorithm A2: In Step 2 we can add edges for each additional edge in the slice $\langle E, \rightarrow\rangle_p$. From [16], there are $O(n|E|)$ such edges. Therefore, the overall complexity is $O(n|E|)$.

6.3 Computing Slices with respect to $\mathbf{EG}(p)$

Similar to the $\mathbf{AG}(p)$ case, we can use $\langle E, \rightarrow\rangle_p$ to obtain the slice $\langle E, \rightarrow\rangle_{\mathbf{EG}(p)}$. Algorithm A3 in Fig. 5 generates the slice $\langle E, \rightarrow\rangle_{\mathbf{EG}(p)}$. For each additional edge (e, f) *that generates a non-trivial strongly connected component* in the slice $\langle E, \rightarrow\rangle_p$, we add an edge from the vertex e to the vertex $\perp$. This is different from the algorithm for $\mathbf{AG}(p)$ where for each additional edge (e, f) in the slice $\langle E, \rightarrow\rangle_p$, we add an edge from the vertex e to the vertex $\perp$.

Observe that consistent cuts of the slice $\langle E, \rightarrow\rangle_p$ that do not include vertices of such a non-trivial strongly connected component do not satisfy $\mathbf{EG}(p)$. For simplicity, let the slice $\langle E, \rightarrow\rangle_p$ have a single non-trivial strongly connected

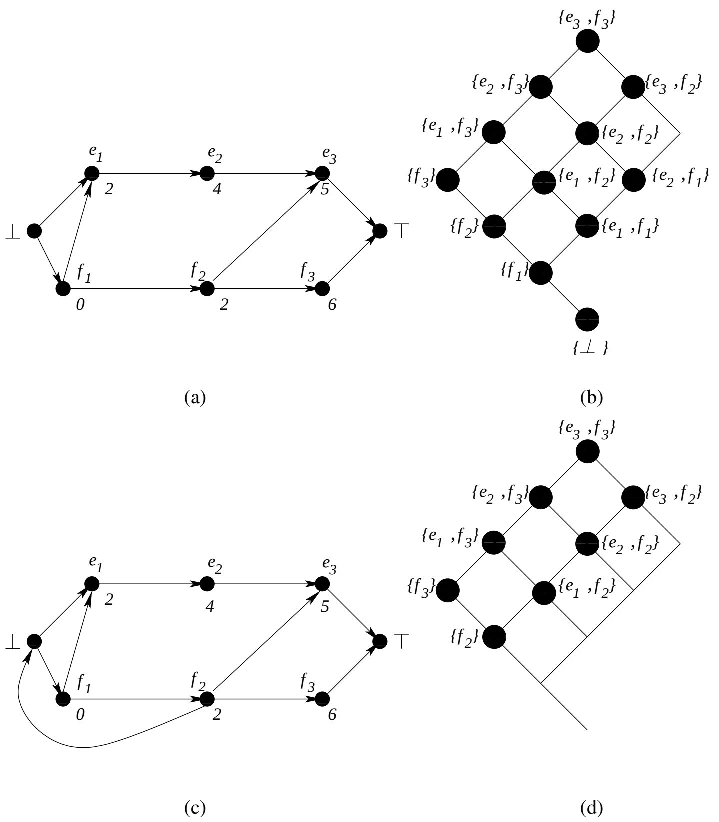

Figure 6. (a) The slice of $\langle E, \rightarrow \rangle$ in Fig. 2 with respect to $(x \neq 5) \vee (y \neq 2)$ (b) the corresponding sublattice (c) The slice of $\langle E, \rightarrow \rangle$ in Fig. 2 with respect to $\mathbf{AG}$ $((x \neq 5) \vee (y \neq 2))$ (d) the corresponding sublattice

component with vertices e and f that is generated by an additional edge (e, f). In this case two types of consistent cuts cannot belong to the slice $\langle E, \rightarrow \rangle_p$ and therefore do not satisfy p: consistent cuts that include f but not e or consistent cuts that include e but not f.

We claim that a consistent cut satisfies $\mathbf{EG}(p)$ iff it includes both vertices e and f. Assume that a consistent cut D does not include vertex e or vertex f or both vertex e and f. Then for all paths starting from D there exists a consistent cut H that can be reached from D in the original computation such that H does not include e but includes f or H does not include f but includes e. It is clear that H does not satisfy p. Therefore from the definition of $\mathbf{EG}(p)$, we have that D does not satisfy $\mathbf{EG}(p)$. Now the other direction. If a consistent cut D does not satisfy $\mathbf{EG}(p)$ then for all paths starting from D there exists a consistent cut H reachable from D such that H does not satisfy p. We know that only the consistent cuts that do not include f but include e or that do not include e but include f do not satisfy p. Since H is reachable from D, we have that D does not include e or f or both e and f.

In the algorithm, by adding an edge from e to $\perp$, we make sure that all events in such strongly connected components exist in all consistent cuts of the output slice.

If we apply the $\mathbf{EG}(p)$ algorithm to the slice in Fig. 6(a) the output is going to be the same as Fig. 6(a) because the additional edge (f_2, e_3) does not generate a non-trivial strongly connected component. The complexity of Algorithm A3 is the same as Algorithm A2. We present an application and the proof of correctness of the algorithm in the appendix.

Next we present a framework for detecting CTL predicates that uses slicing algorithms of this section.

7 Application of Slices in Predicate Detection

Using temporal logic CTL we can specify *safety* and *liveness* properties of a program. Examples of safety properties are invariants and mutual exclusion, $\mathbf{AG}(\neg(critical_1 \wedge critical_2))$, where $critical_i$ denotes the critical section for each process i. Examples of liveness properties are response properties, $\mathbf{AG}(invoke \Rightarrow \mathbf{EF}(response))$, which means that "every invocation has a possible response" and resettability properties such as $\mathbf{AG}(\mathbf{EF}(reset))$.

Employing the results of the previous sections, we can obtain lean slices for the subset of CTL we call regular CTL. Regular CTL includes temporal operators $\mathbf{EF}$, $\mathbf{AG}$, $\mathbf{EG}$, the boolean operator $\wedge$ and atomic propositions are non-temporal regular predicates. Observe that slices with respect to regular CTL predicates contain precisely those consistent cuts that satisfy the predicate.

A general algorithm to detect a predicate in regular CTL such as $\mathbf{AG}(p \wedge \mathbf{EF}(q))$ is as follows: First compute the slice with respect to $\mathbf{AG}(p \wedge \mathbf{EF}(q))$. We can do this recursively from inside to out. In other words apply the appropriate temporal or boolean operator on the slices until we obtain the slice with respect to $\mathbf{AG}(p \wedge \mathbf{EF}(q))$. Then check if the initial cut of the original computation is the same as the initial cut of the slice. If the answer is yes then the predicate is satisfied, otherwise not. The complexity of this general algorithm is dominated by the complexity of computing the slice with respect to a non-temporal regular predicate such as p and q in the above example. This has $O(n^2|E|)$ complexity. Therefore, the overall complexity of predicate detection for regular CTL is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p.

In the regular subset of CTL we can specify *nested temporal logic predicates*. To the best of our knowledge, there do not exist efficient algorithms (polynomial in the number of processes) to detect predicates which contain nested temporal logic predicates.

We can expand the subset of CTL we considered in the above general predicate detection algorithm with boolean negation and disjunction operators. However, a predicate in this new subset of CTL may not be regular. Consequently, the slice may not be a lean slice and we may need to use a global-state-space-construction based approach (such as model checking) to detect the predicate. [1] In this case, the complexity of predicate detection may not be polynomial-time. However, we believe that the state space of the slice is much smaller than the original computation's state space therefore enabling us to ignore many global states. Since the complexity of generating a slice increases polynomially as the size of the computation increases, we believe that our technique, even in the expanded subset of CTL, is an effective strategy for ameliorating state explosion problem.

Following the above general algorithm, now we present an example in Fig. 7 for detecting $\mathbf{AG}(p \wedge \mathbf{EF}(q))$ on the computation given in Fig. 2 when $p = (y \geq 6)$ and $q = (2 \leq x \leq 4) \wedge (y \geq 2)$. First, we generate slices with respect to non-temporal regular predicates p and q. Next, slices of the computation with respect to predicates $\mathbf{EF}(q)$, $p \wedge \mathbf{EF}(q)$ and $\mathbf{AG}(p \wedge \mathbf{EF}(q))$ are obtained. Finally we observe that the initial cut of the original computation, which is $\{\perp\}$, is different from the initial cut of the slice with respect to $\mathbf{AG}(\neg p \vee \mathbf{EF}(q))$, which is empty. Therefore the predicate is not satisfiable.

[1] see the appendix for a discussion of slice categories

8 Experimental Results

We implemented our algorithms in Java. In order to evaluate the effectiveness of our algorithms, we compare our approach with symbolic model checking using NuSMV [3]. For a fair comparison, we disabled counter example generation in NuSMV since our algorithms currently do not generate counter examples. All experiments were performed on a 1.4 Ghz Pentium 4 machine with 768MB RAM and running Linux. We restricted the memory usage to 256MB, but did not set a time limit. No pre-computed variable ordering files were used in the experiments. The three performance metrics we measured are running time (T in seconds), number of BDD nodes (BDD), and memory usage (M in megabytes).

We consider 3 message passing distributed programs *distributed dining philosophers*, *distributed mutual exclusion*, and *primary-secondary* for experiments. We check for complements of mutual exclusion (safety) and starvation freedom (liveness) properties for the first two programs. The complement of the safety property for the first program is the disjunction of $\mathbf{EF}(eat_i \wedge eat_j)$ when $i, j \in 0 \ldots (n-1)$, $i \neq j$,$j = (i+1) \bmod n$,$j = (i-1+n) \bmod n$. The complement of the liveness property for the first program is the disjunction of $\mathbf{EF}(hungry_i \wedge \mathbf{EG}(\neg eat_i))$ when $i \in 0 \ldots n-1$. The properties in the case of the second program are disjunction of $\mathbf{EF}(inCS_i \wedge inCS_j)$ when $i, j \in 0 \ldots (n-1)$, $i \neq j$, and disjunction of $\mathbf{EF}(try_i \wedge \mathbf{EG}(\neg inCS_i))$ when $i \in 0 \ldots n-1$. Note that upon using the distributivity property of $\mathbf{EF}$ over disjunction, we can check each $\mathbf{EF}$ clause separately.

The last program concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The property requires that there is a pair of processes P_i and P_j such that (1) P_i is acting as a primary and correctly thinks that P_j

is its secondary, and (2) P_j is acting as a secondary and correctly thinks that P_i is its primary. Both the primary and secondary may choose new processes as their successor at any time. The complement of the safety property is $\mathbf{EF} \bigwedge(\neg isPrimary_i \vee \neg isSecondary_j \vee (secondary_i \neq P_j) \vee (primary_j \neq P_i))$ when $i, j \in 0 \ldots (n-1)$, $i \neq j$. Note that this predicate contains disjunction operators.

We manually instrumented the first two protocols to generate computations. Furthermore, we use the simulator in Java by Stoller, Unnikrishnan and Liu to generate computations for primary-secondary program. Further details of the programs can be found in [19, 11]. We implemented a translator in Java from computations to SMV to facilitate symbolic model checking.

We consider two scenarios: fault-free and faulty. The programs always generate fault-free computations in terms of the specifications we consider. A faulty computation is generated by randomly injecting faults into a fault-free computation. Note that in the fault-free scenario, we know before-hand that the computation does not contain a faulty global state. We cannot, however, assume the availability of such knowledge in general. Thus it is important to study the behavior of our algorithms in the fault-free scenario as well. For all examples the computations are obtained by running the programs for 10 seconds. The measurements are averaged over 2 computations.

Our results are shown in Tables 1, 2, 3, and 4. With slicing for fault-free computations, the slice is always empty. The memory usage is under 10 MB and the time usage is under 40 seconds. On the other hand, with symbolic model checking, for the few cases when we do not run out of memory (denoted by symbol '*'), the time usage is under 40 minutes and the number of BDD nodes reaches 50 million. Even on injecting 10 faults, the slice remains quite small. The memory usage is under 5MB and the time usage is under 80 seconds except for 2 cases in the primary secondary example where the slicing algorithm did not find a result under 8 hours (denoted by symbol '-'). On the other hand, with symbolic model checking, there are 11 cases where the model checker did not find a result under 8 hours. In cases where the model checker found a result, the time usage is under 30 minutes and the number of BDD nodes reaches 35 million. The predicate in the primary secondary example contains disjunction operators, therefore we first compute the slice with respect to this predicate and then utilize symbolic model checking. However, the state space and the time usage is still much smaller than the symbolic model checking by itself. Note that since our computations are finite executions of the programs, liveness properties may fail. However, our experiments show that slicing is an effective technique for finding bugs.

9 Conclusion

We have presented an approach for detecting temporal logic predicates based on computation slicing. We identified a regular subset of temporal logic CTL that allows provably efficient predicate detection. To this end, we proved that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular predicates when p is a regular predicate and presented polynomial-time algorithms to compute *lean* slices for these temporal predicates. Application of our algorithms for detecting safety and liveness properties of distributed programs was presented. Since with computation slicing the state space is restricted to the part that may potentially affect the predicate, we believe that this is an effective technique to ameliorate state explosion.

References

[1] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[2] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan 1995.

[3] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

Table 1. Distributed Dining Philosophers, Safety Property

	No Faults				10 Faults			
	Slicing		SMV		Slicing		SMV	
n	M	T	BDD	T	M	T	BDD	T
3	0.12	0.08	1,087,341	2,234	0.08	0.16	53,130	1.75
4	0.15	0.09	12,521,837	2,375	0.17	0.22	189,307	5.23
5	0.15	0.14	*		0.19	0.2	345,115	8.98
6	0.26	0.17	*		0.16	0.23	580,974	16.5
7	0.11	0.19	*		0.1	0.32	811,451	27.05
8	0.33	0.29	*		0.34	0.56	1,249,407	48.26
9	0.32	0.36	*		0.35	0.45	1,642,428	61.24
10	0.46	0.39	*		0.47	0.51	2,476,792	111.28
11	0.52	0.55	*		0.34	0.57	3,298,728	697.27
20	1.35	2.29	*		0.67	4.28	-	

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1512–1542, Sept. 1994.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

[6] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proc. Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[7] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.

[8] V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

[9] V. K. Garg and N. Mittal. On detecting global predicates in distributed computations. In *Proc. of the 15^{th} International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, 2001.

[10] V. K. Garg and N. Mittal. On slicing a distributed computation. In *Proc. of the 15^{th} International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, 2001.

[11] S. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[12] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient detection of conjuntions of local predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, 1998.

[13] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the Fifth International Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, New York, 1982. Springer-Verlag.

[14] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *Proc. of the 3rd International Workshop on Automated and Algorithmic Debugging*, pages 43–58, 1997.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] N. Mittal and V. K. Garg. Computation slicing: Techniques and theory. In *In Proc. of the 15^{th} International Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, 2001.

[17] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, 2002.

[18] S. D. Stoller and Y. Liu. Efficient symbolic detection of global properties in distributed systems. In *10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1447 of *LNCS*, pages 357–368, 1998.

Table 2. Distributed Dining Philosophers, Liveness Property

	No Faults				10 Faults			
	Slicing		SMV		Slicing		SMV	
n	M	T	BDD	T	M	T	BDD	T
3	0.04	0.05	823,458	3.4	0.04	0.07	52,569	1.97
4	0.06	0.06	12,351,676	63.17	0.09	0.08	155,349	3.92
5	0.06	0.08	49,546,780	831	0.08	0.09	269,098	5.79
6	0.11	0.06	*		0.12	0.12	432,167	9.55
7	0.11	0.09	*		0.14	0.13	610,683	13.14
8	0.14	0.08	*		0.17	0.14	917,908	20.15
9	0.14	0.08	*		0.19	0.13	1,194,589	24.2
10	0.19	0.09	*		0.23	0.19	1,773,104	37.28
11	0.24	0.12	*		0.31	0.23	2,357,908	47.89
20	0.53	0.54	*		0.48	1.22	13,404,324	16

[19] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *12th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 264–279, 2000.

[20] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

Table 3. Distributed Mutual Exclusion, Safety Property

	No Faults				10 Faults			
	Slicing		SMV		Slicing		SMV	
n	M	T	BDD	T	M	T	BDD	T
3	0.1	0.08	1,541,106	18.9	0.09	0.16	69,267	3.9
4	0.18	0.1	28,569,207	834.45	0.17	0.27	248,380	14.38
5	0.12	0.14	*		0.17	0.25	658,792	41
6	0.41	0.2	*		0.42	0.69	1,371,455	135
7	0.4	0.38	*		0.64	1.2	3,369,348	631.73
8	0.41	1.01	*		0.78	1.96	-	
9	0.66	1.75	*		0.82	3.17	-	
10	0.73	1.82	*		0.75	5.77	-	
11	0.9	2.73	*		1.34	8.85	-	
20	2.51	18.98	*		4.11	35.38	-	

Table 4. Distributed Mutual Exclusion, Liveness Property

	No Faults				10 Faults			
	Slicing		SMV		Slicing		SMV	
n	M	T	BDD	T	M	T	BDD	T
3	0.04	0.06	1,512,394	5.49	0.04	0.12	63,348	2.76
4	0.08	0.07	42,398,313	358.78	0.08	0.14	286,582	9.5
5	0.13	0.11	*		0.15	0.14	909,744	28.16
6	0.18	0.16	*		0.22	0.2	2,271,050	71.25
7	0.24	0.4	*		0.31	1.2	5,895,164	193.42
8	0.29	0.75	*		0.47	1.44	11,606,639	419.05
9	0.49	1.06	*		0.4	2	16,514,367	674
10	0.52	1.32	*		0.54	5.38	24,285,637	1,313
11	0.58	2.5	*		0.54	7.16	33,948,011	301.27
20	1.57	35.95	*		1.79	72.42	564,036,268	11,327

Table 5. Primary Secondary, Safety Property

	No Faults				10 Faults				
	Slicing		SMV		Slicing			SMV	
n	M	T	BDD	T	BDD	M	T	BDD	T
3	0.09	0.06	399,434	120	7,902	0.22	1.63	27,314	1.06
4	0.32	0.09	5,458,687	2,145	14,173	0.09	0.92	103,900	1.58
5	0.44	0.11	*		18,833	0.12	2.67	217,961	341
6	0.15	0.19	*		31,494	0.14	10.58	49,784	426
7	0.4	0.19	*		29,256	0.06	33.66	89,871	1,837
8	0.79	0.38	*		25,278	0.3	4.14	-	
9	1.76	0.5	*		32,125	0.11	5.73	-	
10	0.53	0.53	*		43,534	0.4	11.6	-	
11	3.02	0.84	*		-			-	
20	7.74	6.39	*		-			-	

Appendix

9.1 CTL Syntax and Semantics

We define *successor* of a cut by a relation $\triangleright \subseteq \mathcal{C}(G) \times \mathcal{C}(G)$ such that $C \triangleright D$ if and only if $D = C \cup e$, where e is the set of vertices in some strongly connected component in $\langle E, \rightarrow \rangle$ and $e \notin C$. We denote the reflexive closure of this relation by $\trianglerighteq$. A *consistent cut sequence* $C_0, C_1, \ldots, C_k$ of $(\mathcal{C}(G), \subseteq)$ satisfies that for each $0 \leq i < k$, $C_i \triangleright C_{i+1}$.

Propositional temporal logics use a finite set of atomic propositions AP, each one of which represents some property of the global state. A labeling function $\lambda\colon \mathcal{C}(G) \rightarrow 2^{AP}$ assigns to each global state the set of predicates from AP that hold in it.

The formal syntax of a subset of CTL is given below.

- Every predicate $ap \in AP$ is a CTL formula.
- If p and q are CTL formulas, then so are $\neg p$, $p \wedge q$, $\mathbf{EF}(p)$, $\mathbf{AF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$.

The symbols $\wedge$ and $\neg$ have their usual meanings. There are two path quantifiers: $\mathbf{A}$ denotes *for all consistent cut sequences* and $\mathbf{E}$ denotes *for some consistent cut sequence*. There are two linear temporal operators: $\mathbf{G}$ is the *always* operator and $\mathbf{F}$ is the *eventually* operator.

Given a finite distributive lattice $L = (\mathcal{C}(G), \subseteq)$, the formulas of CTL are interpreted over the consistent cuts in $\mathcal{C}(G)$.

Let p be a CTL formula and C be a consistent cut in $\mathcal{C}(G)$. Then, the satisfaction relation, $L, C \models p$ means that predicate p holds at consistent cut C in lattice $L = (\mathcal{C}(G), \subseteq)$ and is defined inductively below. We denote $C \models p$ as a short form for $L, C \models p$, when L is clear from the context.

- $C \models ap$ iff $ap \in \lambda(C)$ for an atomic proposition ap.
- $C \models \neg p$ iff $C \not\models p$.
- $C \models p \wedge q$ iff $C \models p$ and $C \models q$.
- $C \models p \vee q$ iff either $C \models p$ or $C \models q$.
- $C \models \mathbf{EG}(p)$ iff for some consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \triangleright C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for all $0 \leq i \leq k$.
- $C \models \mathbf{AG}(p)$ iff for all consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \triangleright C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for all $0 \leq i \leq k$.
- $C \models \mathbf{EF}(p)$ iff for some consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \triangleright C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for some $0 \leq i \leq k$.
- $C \models \mathbf{AF}(p)$ iff for all consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \triangleright C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for some $0 \leq i \leq k$.

We define $L \models p$ if and only if $L, \{\bot\} \models p$. [2] We can use following abbreviations in writing CTL formulas:

- $\mathbf{EG}(p) \equiv \neg\mathbf{AF}(\neg p)$, $controllable : p$.
- $\mathbf{AG}(p) \equiv \neg\mathbf{EF}(\neg p)$, $invariant : p$.
- $\mathbf{EF}(p) \equiv \neg\mathbf{AG}(\neg p)$, $possibly : p$.
- $\mathbf{AF}(p) \equiv \neg\mathbf{EG}(\neg p)$, $definitely : p$.

The formula $C \models \mathbf{AG}(p)$ (resp. $C \models \mathbf{EG}(p)$) intuitively means that for all consistent cut sequences (resp. for some consistent cut sequence) $C, \ldots, \mathcal{E}$, p holds at ev-

[2] Note that a distributed program is modeled by a set of partial order sets (directed graphs, computations). In that case, a distributed program P satisfies a CTL formula p if and only if $L \models p$ for each L in P.

ery cut of the sequence. The formula $C \models \mathbf{AF}(p)$ (resp. $C \models \mathbf{EF}(p)$) intuitively means that for all consistent cut sequences (resp. for some consistent cut sequence) $C, \ldots, \mathcal{E}$, there exists a consistent cut that satisfies p. Figure 8 illustrates four important CTL operators on a distributive lattice.

In this paper we restrict atomic propositions to the class of non-temporal regular predicates.

The *predicate detection* problem is to decide whether the initial consistent cut of a distributed computation satisfies a predicate. Since we interpret CTL on a distributive lattice, we can use the properties of the structure (distributive lattice) itself which ultimately aids in obtaining efficient detection algorithms.

Note that, full CTL contains two more linear temporal operators: $\mathbf{X}$ is the *next-time* operator and $\mathbf{U}$ is the *until* operator. Intuitively, we say that $\mathbf{EX}(p)$ (resp. $\mathbf{AX}(p)$) holds at a global state G if and only if for some path (resp. for all paths) starting from G and ending at the final state, the predicate p holds at the successor global state of G on the path. We say that $\mathbf{EU}(p, q)$ (resp. $\mathbf{AU}(p, q)$) holds at a global state G if and only if for some path (resp. for all paths) starting from G and ending at the final state, the predicate q holds at a global state on the path and at every preceding state on the path p holds. Note that if a global state satisfies predicate q, it satisfies $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$.

9.2 Proof of $\mathbf{EF}(p)$ is regular

Lemma 2 *If p is a regular predicate then $\mathbf{EF}(p)$ is a regular predicate.*

Proof: For two arbitrary consistent cuts D and H, given that $D \models \mathbf{EF}(p)$ and $H \models \mathbf{EF}(p)$, we prove that $(D \sqcap H) \models \mathbf{EF}(p)$ and $(D \sqcup H) \models \mathbf{EF}(p)$. Since p is regular, there exists a greatest consistent cut where p holds. Let this cut be W.

1. For all consistent cuts C which satisfy p, $C \subseteq W$, since p is regular
2. $D \subseteq W \wedge H \subseteq W$, from 1 and the definition of $\mathbf{EF}(p)$
3. $(D \sqcup H) \subseteq W \wedge (D \sqcap H) \subseteq W$, from 2
4. $(D \sqcap H) \models \mathbf{EF}(p) \wedge (D \sqcup H) \models \mathbf{EF}(p)$, from the definition of $\mathbf{EF}(p)$

Note that we only use the existence of the greatest satisfying cut of predicate p in this proof. ■

9.3 Proof of $\mathbf{AG}(p)$ is regular

Lemma 3 *If p is a regular predicate then $\mathbf{AG}(p)$ is a regular predicate.*

Proof: For two arbitrary consistent cuts D and H, given that $D \models \mathbf{AG}(p)$ and $H \models \mathbf{AG}(p)$, we prove that $(D \sqcap H) \models \mathbf{AG}(p)$ and $(D \sqcup H) \models \mathbf{AG}(p)$.

1. $(D \sqcup H) \models \mathbf{AG}(p)$, from the definition of $D \models \mathbf{AG}(p)$ or $H \models \mathbf{AG}(p)$

Let J be an arbitrary consistent cut such that $(D \sqcap H) \subseteq J$.

2. $(J \sqcup D) \models p \wedge (J \sqcup H) \models p$, from the definitions of $D \models \mathbf{AG}(p)$ and $H \models \mathbf{AG}(p)$
3. $((J \sqcup D) \sqcap (J \sqcup H)) \models p$, since p is regular
4. $(J \sqcup (D \sqcap H)) \models p$, from 3 and distributivity of the lattice
5. $J \models p$, from 4 and $(D \sqcap H) \subseteq J$
6. $(D \sqcap H) \models \mathbf{AG}(p)$, from 5 and J is an arbitrary cut such that $(D \sqcap H) \subseteq J$
7. $(D \sqcap H) \models \mathbf{AG}(p) \wedge (D \sqcup H) \models \mathbf{AG}(p)$, from 1 and 6

■

9.4 Proof of $\mathbf{EG}(p)$ is regular

Lemma 4 *If p is a regular predicate then $\mathbf{EG}(p)$ is a regular predicate.*

Proof: For two arbitrary consistent cuts D and H, given that $D \models \mathbf{EG}(p)$ and $H \models \mathbf{EG}(p)$, we prove that $(D \sqcap H) \models \mathbf{EG}(p)$ and $(D \sqcup H) \models \mathbf{EG}(p)$. Consider the case when $D \nsubseteq H$ and $H \nsubseteq D$, otherwise the theorem would hold trivially.

We apply the definition of $\mathbf{EG}(p)$ to obtain 1 and 2 below.

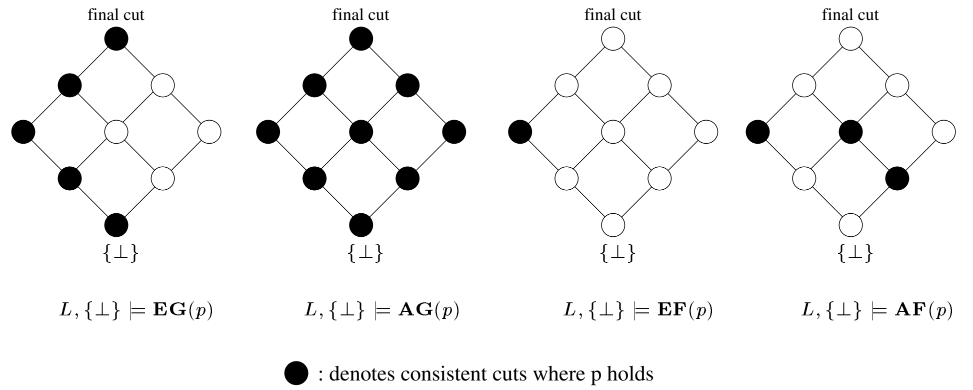

Figure 8. Basic CTL operators

1. $D \models \mathbf{EG}(p)$ iff for some consistent cut sequence $D_0, \ldots, D_k$ such that (i) $D_0 = D$, (ii) $D_k = \mathcal{E}$, (iii) $D_j \triangleright D_{j+1}$ for $0 \leq j < k$, we have (iv) $D_j \models p$ for all $0 \leq j \leq k$. Interesting case is when $k \geq 1$, otherwise the theorem is trivially true because when $k = 0$, $D = \mathcal{E}$ and $((D \sqcap H) = H) \models \mathbf{EG}(p)$ and $((D \sqcup H) = D) \models \mathbf{EG}(p)$.
2. $H \models \mathbf{EG}(p)$ iff for some consistent cut sequence $H_0, \ldots, H_m$ such that (i) $H_0 = H$, (ii) $H_m = \mathcal{E}$, (iii) $H_i \triangleright H_{i+1}$ for $0 \leq i < m$, we have (iv) $H_i \models p$ for all $0 \leq i \leq m$. Similar to 1, the interesting case is when $m \geq 1$.

We will first prove that $(H \sqcup D) \models \mathbf{EG}(p)$.

3. $\forall j : 0 \leq j \leq k : (H_0 \sqcup D_j) \models p$, since $D_j \models p$ for all $0 \leq j \leq k$ from 1, $H_0 \models p$ from 2, and p is regular
4. $\forall j : 0 \leq j < k : (H_0 \sqcup D_j) \trianglerighteq (H_0 \sqcup D_{j+1})$. Because, $D_{j+1} = D_j \cup e$, where e is the set of events in some strongly connected component in $\langle E, \rightarrow \rangle$. If $e \subseteq H_0$ then $(H_0 \sqcup D_j) = (H_0 \sqcup D_{j+1})$. If $e \nsubseteq H_0$ then $(H_0 \sqcup D_j) \triangleright (H_0 \sqcup D_{j+1})$ since $D_j \triangleright D_{j+1}$ for $0 \leq j < k$ from 1
5. There exists a consistent cut sequence $(H_0 \sqcup D_0), (H_0 \sqcup D_1), \ldots, (H_0 \sqcup D_k)$ such that (i) $(H_0 \sqcup D_0) = (H \sqcup D)$, (ii) $(H_0 \sqcup D_k) = \mathcal{E}$, (iii) $(H_0 \sqcup D_j) \trianglerighteq (H_0 \sqcup D_{j+1})$ for $0 \leq j < k$, and we have (iv) $(H_0 \sqcup D_j) \models p$ for all $0 \leq j \leq k$, from 3 and 4
6. $(H \sqcup D) \models \mathbf{EG}(p)$, from 5 and the definition of $\mathbf{EG}(p)$

Observe that, the sequence in 5 may have stuttering consistent cuts, but it is straightforward to obtain a non-stuttering sequence from this so that the definition of $\mathbf{EG}(p)$ is still applicable. Next we prove that $(H \sqcap D) \models \mathbf{EG}(p)$.

7. $\forall j : 0 \leq j \leq k : (H_0 \sqcap D_j) \models p$, from 1, 2, and p is regular
8. $\forall j : 0 \leq j < k : (H_0 \sqcap D_j) \trianglerighteq (H_0 \sqcap D_{j+1})$, similar to 4
9. There exists a consistent cut sequence $(H_0 \sqcap D_0), (H_0 \sqcap D_1), \ldots, (H_0 \sqcap D_{k-1}), (H_0), \ldots, (H_m)$ such that (i) $(H_0 \sqcap D_0) = (H \sqcap D)$ and $H_0 = (H_0 \sqcap D_k)$, (ii) $H_m = \mathcal{E}$, (iii) $(H_0 \sqcap D_j) \trianglerighteq (H_0 \sqcap D_{j+1})$ for $0 \leq j < k$, $H_i \triangleright H_{i+1}$ for $0 \leq i < m$, and we have (iv) $(H_0 \sqcap D_j) \models p$ for all $0 \leq j \leq k$ and $H_i \models p$ for all $0 \leq i \leq m$, from 7 and 8. Figure 9 shows the finite prefix in 9.
10. $(H \sqcap D) \models \mathbf{EG}(p)$, from 9 and the definition of $\mathbf{EG}(p)$
11. $(H \sqcap D) \models \mathbf{EG}(p) \wedge (H \sqcup D) \models \mathbf{EG}(p)$, from 6 and 10 ■

9.5 Proof of $\mathbf{AF}(p)$ is closed under union

Lemma 5 *If the set of consistent cuts that satisfy p is closed under union then the set of consistent cuts that satisfy* $\mathbf{AF}(p)$ *is closed under union.*

Proof: For two arbitrary consistent cuts D and H, given that $D \models \mathbf{AF}(p)$ and $H \models \mathbf{AF}(p)$, we prove that $(D \sqcup H) \models \mathbf{AF}(p)$.

We apply the definition of $\mathbf{AF}(p)$ in 1 and 2 below.

1. $D \models \mathbf{AF}(p)$ iff for all consistent cut sequences $D_0, \ldots, D_l$ such that (i) $D_0 = D$, (ii) $D_l = \mathcal{E}$, (iii) $D_i \triangleright D_{i+1}$ for $0 \leq i < l$, we have (iv) $D_k \models p$ for some $0 \leq k \leq l$.
2. $H \models \mathbf{AF}(p)$ iff for all consistent cut sequences $H_0, \ldots, H_r$ such that (i) $H_0 = H$, (ii) $H_r = \mathcal{E}$, (iii) $H_i \triangleright H_{i+1}$ for $0 \leq i < r$, we have (iv) $H_m \models p$ for some $0 \leq m \leq r$.

Our goal is to show that for each sequence $(D \sqcup H), \ldots, \mathcal{E}$, there exists a consistent cut in the sequence that satisfies p. We can extend each of these sequences backwards to include D or H.

For each sequence $(D \sqcup H), \ldots, \mathcal{E}$, consider the two extended versions of the sequence, one that includes D and the other that includes H. There are three cases:

Case 1: $(D \sqcup H) \subseteq D_k$, then D_k is the consistent cut that satisfies p in the sequence.

Case 2: $(D \sqcup H) \subseteq H_m$, then H_m is the consistent cut that satisfies p in the sequence.

Case 3: $D_k \subseteq (D \sqcup H)$ and $H_m \subseteq (D \sqcup H)$,

3. $(D_k \sqcup H_m) \models p$, because p is closed under union
4. $D \subseteq D_k \wedge H \subseteq H_m$, from the definition of D_k and H_m
5. $(D \sqcup H) \subseteq (D_k \sqcup H_m)$, from 4
6. $D_k \subseteq (D \sqcup H) \wedge H_m \subseteq (D \sqcup H)$, case 3
7. $(D_k \sqcup H_m) \subseteq (D \sqcup H)$, from 6
8. $(D \sqcup H) = (D_k \sqcup H_m)$, from 5 and 7
9. $(D \sqcup H) \models p$, from 3

Then we conclude that $(D \sqcup H)$ is the consistent cut that satisfies p in the sequence.

■

9.6 $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$ may not be regular when p and q are regular

We give an example in Figure 10(a) where consistent cuts D and H satisfy $\mathbf{EX}(p)$ and $\mathbf{AX}(p)$ but their intersection $(D \sqcap H)$ does not. Similarly, we give an example in Figure 10(b) where consistent cuts V and W satisfy $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$ but their intersection $(V \sqcap W)$ does not.

9.7 Proof of Algorithm A1

Theorem 6 *Algorithm A1 outputs* $\langle E, \rightarrow \rangle_{\mathbf{EF}(p)}$.

Proof: We will prove that a consistent cut $D \in \mathcal{C}(G)$ satisfies $\mathbf{EF}(p)$ if and only if D is in the computation output by Algorithm A1. Let W be the greatest cut of $\langle E, \rightarrow \rangle$ where p holds, which is the final cut of $\langle E, \rightarrow \rangle_p$. This is well-defined since p is regular.

Observe that the output of the algorithm is a computation which is the same as the original computation except for the final consistent cut, which is W instead of $\mathcal{E}$. From this observation we can deduce that the lattice of consistent cuts corresponding to the output of Algorithm A1 is the same as the lattice of consistent cuts of the original computation up to W.

$(\Rightarrow)$: Assume that $\mathbf{EF}(p)$ holds at a consistent cut $D \in \mathcal{C}(G)$. We prove that D is in the computation output by Algorithm A1.

1. $D \models \mathbf{EF}(p)$ iff $\exists J : J \in \mathcal{C}(G) : D \subseteq J \wedge J \models p$, from the definition of $\mathbf{EF}(p)$
2. $J \subseteq W$, since W is the greatest cut where p holds
3. $D \subseteq W$, from 1 and 2
4. D is in the computation output by A1, from 3 and the above observation

$(\Leftarrow)$: Assume that D is in the computation output by Algorithm A1. We prove that $D \models \mathbf{EF}(p)$.

5. D is in the computation output by A1 implies that $D \subseteq W$, since W is the final cut of the output
6. $D \subseteq W$ implies that $D \models \mathbf{EF}(p)$, from $W \models p$ and the definition of $\mathbf{EF}(p)$ ■

Figure 11 shows the application of Algorithm A1. Given a slice of the computation in Fig. 2(a) as in Fig. 11(a), the application of the temporal operator $\mathbf{EF}$ on the slice is given in Fig. 11(c). The final consistent cut of the slice is $\{e_2, e_1, f_3, f_2, f_1, \bot\}$.

9.8 Proof of Algorithm A2

Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A2. Using the following two lemmas, we first prove that for all consistent cuts D of the original computation, $V \subseteq D$ if and only if $D \models \mathbf{AG}(p)$. Next, we use these lemmas to prove the correctness of the algorithm.

Lemma 7 *Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A2. For all $D \in \mathcal{C}(G)$, if $V \subseteq D$ then $D \models \mathbf{AG}(p)$.*

Proof: Consider the case when $\mathcal{E} \models p$ in $\langle E, \rightarrow \rangle$ otherwise the slice with respect to $\mathbf{AG}(p)$ is empty. V is well-defined since either $V = \{\perp\}$ or $\perp \in V$. The latter case occurs when there exists a pair of vertices e and f that satisfies the conditions in Step 2. In other words, since $\perp \rightarrow e$ and Step 2 adds an edge $e \rightarrow \perp$, $\perp \in V$.

We know that a non-trivial consistent cut at least contains the vertex $\perp$. It follows that V is the initial cut of the output. It is straightforward to show that the output has the same edges as the original computation after initial cut V. Therefore if $V \subseteq D$ then D is in the output. Similarly, all $H \supseteq D$ are in the output. We also have that D and all $H \supseteq D$ satisfy p since the output is obtained from $\langle E, \rightarrow \rangle_p$. Therefore, $D \models \mathbf{AG}(p)$. ■

Observe that the above Lemma implies that $V \models \mathbf{AG}(p)$. Now we prove the other direction.

Lemma 8 *Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A2. For all $D \in \mathcal{C}(G)$, if $D \models \mathbf{AG}(p)$ then $V \subseteq D$.*

Proof: Let D_{min} be the least consistent cut that satisfies $\mathbf{AG}(p)$. D_{min} is well-defined since $\mathbf{AG}(p)$ is regular. We know that if $D \models \mathbf{AG}(p)$ then $D_{min} \subseteq D$. Therefore, it is sufficient to prove that $V \subseteq D_{min}$.

For this purpose it remains to show that for all $e \in E$, if $e \in V$ then $e \in D_{min}$. Note that V contains exactly those vertices that have a path to the vertex $\perp$ in the output. We distinguish between two kinds of edges in the output:

(i) the edge that was added because it was in the slice $\langle E, \rightarrow \rangle_p$.

(ii) the edge we added from a vertex e to the vertex $\perp$ because there exists a vertex f such that the pair e and f satisfies the required conditions in Step 2 of Algorithm A2.

Since e is in V, there is a path from e to the vertex $\perp$ in the output. We will prove the lemma by induction on the length of the path.

<u>Base Case:</u> If the path is of length 0 then e is the vertex $\perp$. Now, assume that if an event has a path of length $k \geq 0$ to the vertex $\perp$ then it belongs to D_{min}.

<u>Induction Step:</u> We will prove the assertion for $k+1$. Consider the next vertex in the path, say g. By induction hypothesis, g is in D_{min}. The edge from e to g can be of type (i) or (ii) defined above.

If the edge is of type (i) then since g is in D_{min} and $e \rightarrow g$ then e is in D_{min}.

If the edge is of type (ii) then g is the vertex $\perp$ and there exists a vertex f such that the pair e and f satisfies the required conditions in Step 2. We now prove by contradiction that $e \in D_{min}$. Assume $e \notin D_{min}$. If $f \in D_{min}$ then we are done because D_{min} cannot belong to the slice since $e \rightarrow f$ in the slice therefore e must be included in D_{min} as well. If $f \notin D_{min}$ then consider a path in the computation $\langle E, \rightarrow \rangle$ from D_{min} to the final consistent cut in which f is executed before e (it is always possible to find such a path because $\neg(e \rightarrow f)$ in the computation). Consider the consistent cut immediately after the execution of f, say D. Again D cannot not belong to the slice because $e \rightarrow f$ in the slice therefore e must be included in D as well. Thus D does not satisfy p. It follows that there exists a consistent cut on the path from D_{min} to the final cut that does not satisfy p. Hence D_{min} cannot satisfy $\mathbf{AG}(p)$, this leads to a contradiction. Therefore $e \in D_{min}$. ■

From Lemmas above it is easy to see that V is the least consistent cut that satisfies $\mathbf{AG}(p)$. Now we prove the correctness of the algorithm.

Theorem 9 *Algorithm A2 outputs $\langle E, \rightarrow \rangle_{\mathbf{AG}(p)}$.*

Proof: We will prove that a consistent cut $D \in \mathcal{C}(G)$ satisfies $\mathbf{AG}(p)$ if and only if D is in the computation output by Algorithm A2. Observe that the output of the algorithm is a computation which is the same as the original computation except for the edges that generate the initial consistent cut, which is V. From this observation we deduce that every consistent cut of the original computation after V is included in the output.

$(\Rightarrow)$: Assume that $\mathbf{AG}(p)$ holds at a consistent cut $D \in \mathcal{C}(G)$. From Lemma 8 we know that $V \subseteq D$. The observation above says that every consistent cut of the original computation after V is included in the output and since $V \subseteq D$, D is in the computation output by A2.

$(\Leftarrow)$: Assume that D is in the computation output by Algorithm A2. This implies that $V \subseteq D$ since V is the initial cut of the computation output by A2. From Lemma 7, $V \subseteq D$ implies $D \models \mathbf{AG}(p)$.

■

9.9 Proof of Algorithm A3

The proof of correctness of Algorithm A3 is similar to the proof of Algorithm A2.

Lemma 10 *Let D_{min} be the least consistent cut that satisfies $\mathbf{EG}(p)$. Consider any pair of events e and f that satisfies the conditions in Step 2 of Algorithm A3. Then, $e \in D_{min}$.*

Proof: Assume $e \notin D_{min}$. If $f \in D_{min}$ we are done because D_{min} cannot belong to the slice and thus does not satisfy p. There are two types of paths in the computation $\langle E, \rightarrow \rangle$ from D_{min} to the final consistent cut: First type of paths is ones in which f is executed before e (it is always possible to find such a path because $\neg(e \rightarrow f)$ in the computation). Consider the consistent cut immediately after the execution of f, say D. Again D cannot not belong to the slice since $e \rightarrow f$ in the slice and therefore e must be included in D as well, and thus D does not satisfy p. Second type of paths is ones in which e is executed before f (it is possible to find such a path if $\neg(f \rightarrow e)$ in the computation, otherwise $(f \rightarrow e)$ therefore such a path does not exist, and we are done). Consider the consistent cut immediately after the execution of e, say H. Again H cannot not belong to the slice since $f \rightarrow e$ in the slice and thus does not satisfy p. It follows that there exists a consistent that on all paths from D_{min} to the final cut that does not satisfy p and therefore D_{min} cannot satisfy $\mathbf{EG}(p)$, contradiction. ■

Lemma 11 *Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A3. For all consistent cuts D in $\langle E, \rightarrow \rangle_p$, if $V \subseteq D$ then $D \models \mathbf{EG}(p)$.*

Proof: Consider the case when $\mathcal{E} \models p$ in $\langle E, \rightarrow \rangle$ otherwise the slice of $\mathbf{EG}(p)$ is empty. The computation $\langle E, \rightarrow \rangle_p$ can have additional edges which did not exist in $\langle E, \rightarrow \rangle$. Step 2 of Algorithm A3 puts all the vertices from which there are additional edges that completes a strongly connected component into a strongly connected component, V, that includes vertex $\perp$. We know that a non-trivial consistent cut at least contains the vertex $\perp$. It follows that V is the initial cut of the output.

After D in the output, there may exist vertices e and f such that $(e \rightarrow f)$ and $\neg(f \rightarrow e)$ in $\langle E, \rightarrow \rangle_p$, and $\neg(e \rightarrow f)$ in $\langle E, \rightarrow \rangle$. If such edges do not exist then the output has the same edges as the original computation after initial cut V and the proof is similar to Lemma 7.

Since there do not exist non-trivial strongly connected components with additional edges in the output after initial cut V, every sequence of the output after V is a sequence of the original computation. Furthermore, each consistent cut on such sequences satisfy p since the output is obtained from $\langle E, \rightarrow \rangle_p$. Therefore there exists a sequence starting from D and ending at $\mathcal{E}$ that satisfies $\mathbf{EG}(p)$. ■

Lemma 12 *Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A3 and D_{min} be the least consistent cut that satisfies $\mathbf{EG}(p)$. For all $e \in E$, if $e \in V$ then $e \in D_{min}$.*

Proof: Note that V contains exactly those vertices that have a path to the vertex $\perp$ in the output. We distinguish between two kinds of edges in the output:

(i) the edge that was added because it was in the slice $\langle E, \rightarrow \rangle_p$.

(ii) the edge we added from a vertex e to the event with $\perp$ because there exists an event f such that the pair e and f satisfies the required conditions in Step 2 of Algorithm A3.

Since e is in V there is a path from e to the event with $\perp$ in the $\langle E, \rightarrow \rangle_p$. We will prove the lemma by induction on the length of the path.

Base Case: If the path is of length 0 then e is the vertex with $\perp$ and Lemma 10 is applicable. Now, assume that if an event has a path of length $k \geq 0$ to the event with $\perp$ then it belongs to D_{min}.

Induction Step: We will prove the assertion for $k+1$. Consider the next vertex in the path, say g. By induction hypothesis, g is in D_{min}. The edge from e to g can be of type (i) or (ii) above. If it is of type (i) then e is in D_{min} since g is in D_{min} and $e \rightarrow g$. If it is of type (ii) then g is the vertex $\perp$ and there exists a vertex f such that the pair e and f satisfies the required conditions in Step 2 of Algorithm A3. Lemma 10 is applicable in this case and e is in D_{min}. ■

Lemma 13 *Let V be the strongly connected component that contains $\perp$ which is obtained after Step 2 of Algorithm A3 and D_{min} be the least consistent cut that satisfies $\mathbf{EG}(p)$. V is equivalent to D_{min}.*

Proof: D_{min} is well defined since $\mathbf{EG}(p)$ is regular from Lemma 4. Lemma 12 implies that $V \subseteq D_{min}$. Lemma 11 shows that $V \models \mathbf{EG}(p)$. Since D_{min} is the least consistent cut that satisfies $\mathbf{EG}(p)$, $D_{min} \subseteq V$. Therefore, V is equivalent to D_{min}. ■

Theorem 14 *Algorithm A3 outputs $\langle E, \rightarrow \rangle_{\mathbf{EG}(p)}$.*

Proof: We will prove that a consistent cut $D \in \mathcal{C}(G)$ satisfies $\mathbf{EG}(p)$ if and only if D is in the computation output by Algorithm A3. From Lemma 13, after Step 2 of Algorithm A3 the least consistent cut V that satisfies $\mathbf{EG}(p)$ in the original computation is generated. Observe that, the output of the algorithm is a computation which is the same as the original computation except for the initial consistent cut, which is V. From this observation we can deduce that every consistent cut of the original computation after V is included in the output.

$(\Rightarrow)$: Assuming that $\mathbf{EG}(p)$ holds at a consistent cut $D \in \mathcal{C}(G)$, we prove that D is in the computation output by Algorithm A3.

1. $V \subseteq D$, from Lemma 13, V is the least satisfying cut of $\mathbf{EG}(p)$

2. D is in the computation output by A3, from 1 and above observations

$(\Leftarrow)$: Assuming that D is in the computation output by Algorithm A3, we prove that $D \models \mathbf{EG}(p)$.

4. D is in the computation output by A3 implies that $V \subseteq D$, since V is the initial cut of the computation output by A3

5. $V \subseteq D$ implies that $D \models \mathbf{EG}(p)$, from Lemma 11 ■

Figure 12 shows the application of Algorithm A3. Given a slice of the computation in Fig. 2(a) as in Fig. 12(a), the application of the temporal operator $\mathbf{EG}$ on the slice is given in Fig. 12(c). There are two additional edges, (e_2, e_1) and (f_3, e_3), in the slice $\langle E, \rightarrow \rangle_p$ that do not exist in the original computation. Only one of these edges, (e_2, e_1), generates a non-trivial strongly connected component in the slice. Therefore an edge from e_2 to $\perp$ is added.

9.10 Relationship of Slice Categories

When the predicate contains disjunction and negation operators the slices can be approximate slices. An *approximate* slice may not necessarily be the smallest computation that contains all consistent cuts of the original computation that satisfy the predicate. For example, suppose we wish to compute the slice of a computation with respect to a predicate $\mathbf{EF}((p_1 \vee p_2) \wedge (p_3 \vee p_4))$, where p_i is a regular predicate. The slice for this predicate is approximate because after applying disjunction operator on the slices, we obtain a slice for $\mathbf{EF}(reg(p_1 \vee p_2) \wedge reg(p_3 \vee p_4))$ instead of a slice for $\mathbf{EF}((p_1 \vee p_2) \wedge (p_3 \vee p_4))$. Similarly, a negation operator in the predicate could result in an approximate slice. This is expected since detecting even a predicate in 2-CNF form when no two clauses contain variables from the same process is NP-complete in general [9].

Now we present slice categories. Given a computation $\langle E, \rightarrow \rangle$ and a predicate p, we can obtain a slice of the computation with respect to p.

Figure 13 shows the relationship of slice categories, where

- $\mathcal{C}(G)$ corresponds to the set of all consistent cuts of the computation $\langle E, \rightarrow \rangle$.

- $\models\ p$ corresponds to the set of all consistent cuts in $\mathcal{C}(G)$ that satisfy predicate p.
- *slice* corresponds to the *smallest* sublattice of $\mathcal{C}(G)$ that contains the region $\models p$. If the region X in *slice* is empty (or equivalently, if the region $\models p$ is a sublattice of $\mathcal{C}(G)$) then the slice is a *lean slice*.
- *approximate slice* corresponds to *any* sublattice of $\mathcal{C}(G)$ that contains the region $\models p$.

For example, given four regular predicates p_1, p_2, p_3, and p_4:

- We generate lean slices for the following predicates: $p_1, p_2, p_1 \wedge p_2$, $\mathbf{EF}(p_1)$, $\mathbf{EF}(p_1 \wedge p_2)$, $\mathbf{EG}(p_2)$, etc.
- We generate slices for the following predicates: $p_1 \vee p_2$, $\neg p_1$, $\neg p_2$, $(p_1 \wedge p_2) \vee (p_3 \wedge p_4)$ etc.
- We may generate approximate slices for the following predicates: $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$, $(\neg p_1 \vee p_2)$, $\mathbf{EG}(\neg p_1 \vee p_2)$, etc.

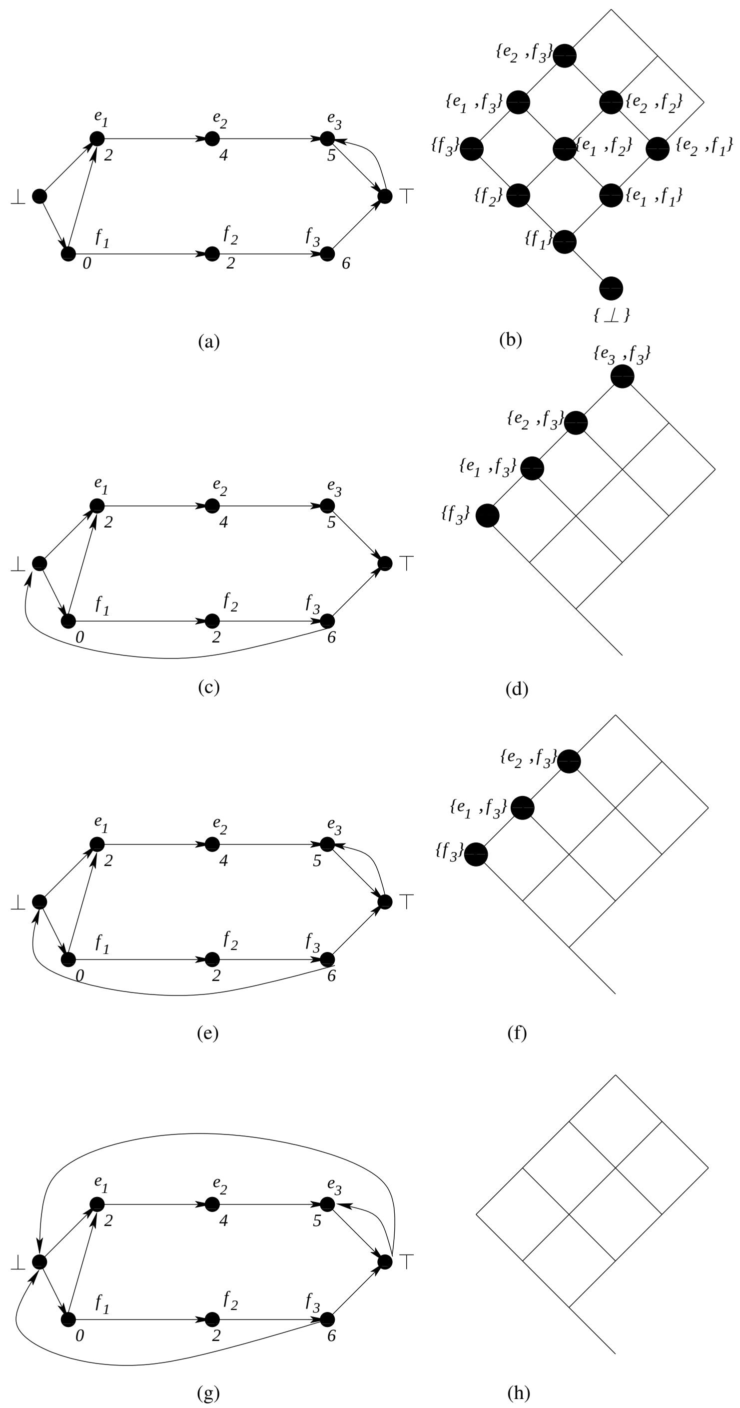

Figure 7. (a) The slice of $\langle E, \rightarrow \rangle$ in Fig. 2 with respect to $\mathrm{EF}(q)$ (b) the corresponding sublattice (c) with respect to p (d) the corresponding sublattice (e) with respect to $p \wedge \mathbf{EF}(q)$ (f) the corresponding sublattice (g) with respect to $\mathbf{AG}(p \wedge \mathbf{EF}(q))$ (h) the corresponding sublattice

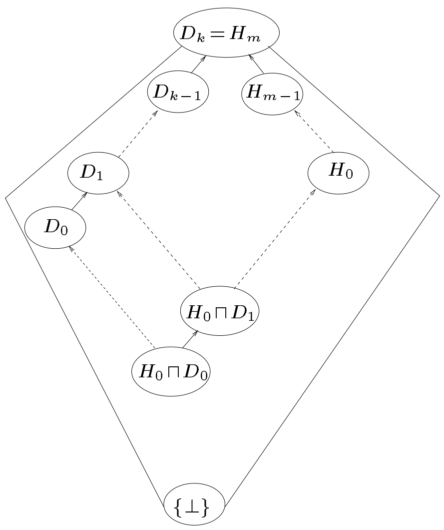

Figure 9. Lemma 4 step 9, $(H \sqcap D) \models \mathbf{EG}(p)$

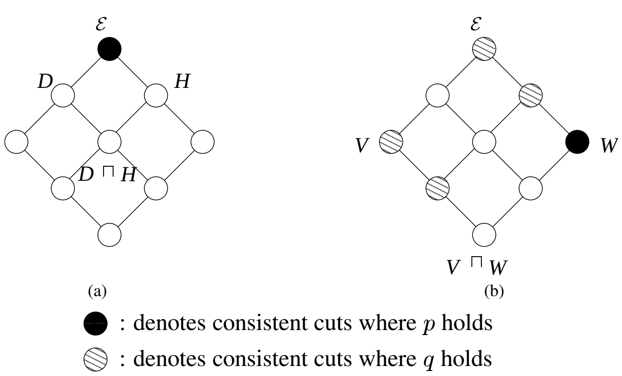

Figure 10. (a) $\mathbf{EX}(p)$ and $\mathbf{AX}(p)$ may not be regular when p is regular (b) $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$ may not be regular when p and q are regular

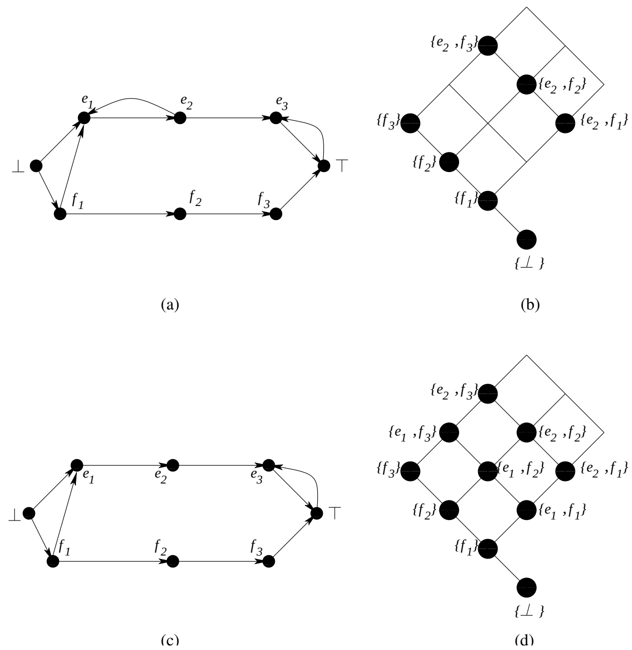

Figure 11. (a) A slice of $\langle E, \rightarrow \rangle$ in Fig. 2 (b) the corresponding sublattice (c) The application of the temporal operator $\mathbf{EF}$ on the slice in (a) (d) the corresponding sublattice

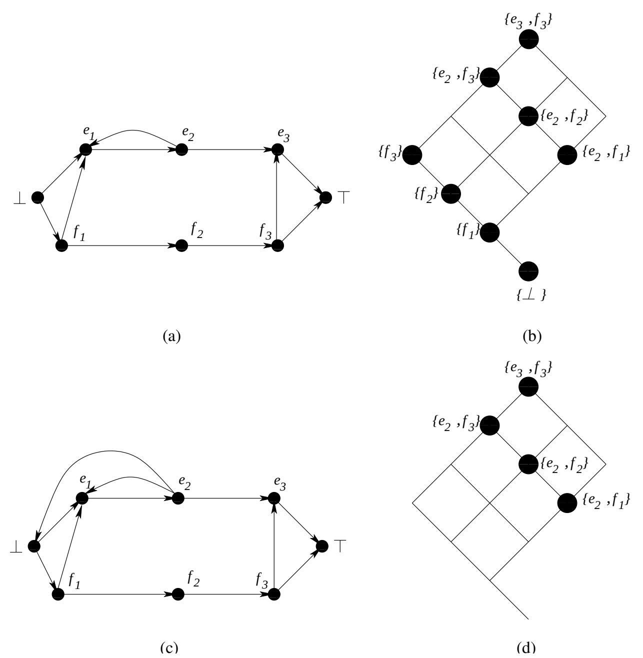

Figure 12. (a) A slice of $\langle E, \rightarrow \rangle$ in Fig. 2 (b) the corresponding sublattice (c) the application of the temporal operator EG on the slice in (a) (d) the corresponding sublattice

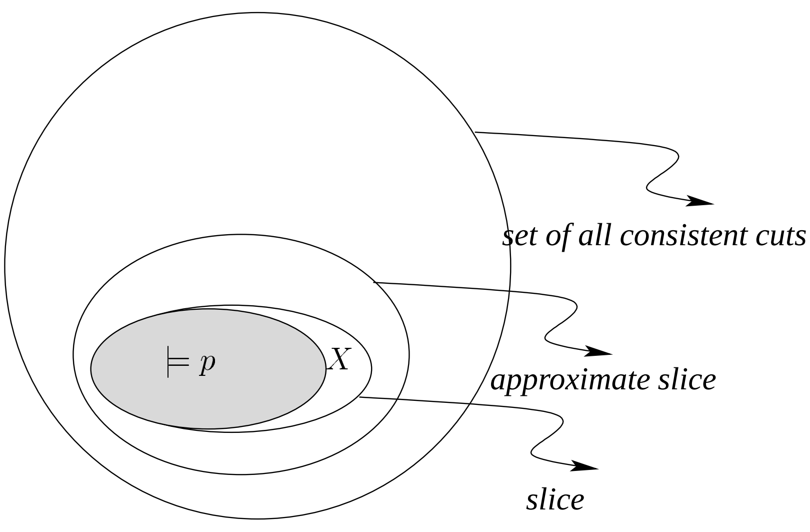

Figure 13. Relationship of slice categories