Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing [*]

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

Abstract

Detecting whether a finite execution trace (or a computation) of a distributed program satisfies a given predicate, called predicate detection, is a fundamental problem in distributed systems. It finds applications in many domains such as testing, debugging, and monitoring of distributed programs. However predicate detection suffers from the state explosion problem – 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. *We present polynomial-time algorithms to compute slices with respect to temporal logic predicates from a "regular" subset of CTL, that contains temporal operators EF, EG, and AG. Furthermore, we show that these slices contain precisely those global states of the original computation that satisfy the predicate.*

Using temporal predicate slices, we give an efficient (polynomial in the number of processes) predicate detection algorithm for a subset of CTL that we call regular CTL. Regular CTL contains nested temporal predicates for which, to the best of our knowledge, there did not previously exist efficient predicate detection algorithms. Then we show that we can enlarge the subset of CTL and still obtain effective results. Our algorithm has been implemented as part of a tool for analysis of distributed programs. We illustrate the effectiveness of our techniques on several protocols achieving speedups of over three orders of magnitude in one example, compared to partial order state-space search of SPIN. Furthermore, we were able to complete the verification for 250 processes for a partial order trace.

Keywords: *specification verification of distributed systems, runtime verification, predicate detection, testing, debugging, formal methods, 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

A fundamental problem in distributed systems is that of *predicate detection* – detecting whether a finite execution trace of a distributed program satisfies a given predicate. There are applications of predicate detection in many domains such as testing, debugging, and monitoring of distributed programs. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources.

We can model a finite trace in two ways. The first model imposes a partial order between events, for example Lamport's *happened-before* relation [19]. The second model imposes a total order (interleaving) of events. We use the former approach in this paper, which is a more faithful representation of concurrency [19].

Consider an execution of a distributed program. The partial order model of the resulting execution trace is shown in Figure 1(a). In the trace, 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. Figure 1(b) contains the set of all reachable global states of the computation reachable from the initial state $\{e_0, f_0\}$. In the figure, we represent a global state as a tuple where each element is the last event that occurred on a process. Observe that $\{e_1, f_0\}$ is not a reachable global state because it depicts a situation where a message has been received from P_2 by P_1, that is e_1, but P_2 has not yet sent the message. By using a partial order representation, we are able to capture all possible interleavings of events, namely ten in total, rather than a single interleaving. One such interleaving sequence is $\{e_0, f_0\}$, $\{e_0, f_1\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_2\}$, $\{e_3, f_3\}$ as shown in Figure 1(b) with thick lines. Therefore we can obtain better coverage in terms of testing and debugging by capturing all interleavings. This coverage may translate into finding bugs that are not found using a single interleaving.

The main problem in predicate detection in the partial order model 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. A variety of strategies for ameliorating the state explosion problem, including symbolic representation of states and partial order reduction have been explored [21, 11, 33, 24, 5, 31, 32].

In this paper, we present a provably efficient predicate detection algorithm using a technique called computation slicing. *Computation slicing* was introduced in [9, 22] as an abstraction technique for analyzing *distributed computations* (finite execution traces). 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. This is in contrast to traditional slicing techniques which either work at the program level or do slicing with respect to variables. Computation 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.

With the results of this paper, we can efficiently use computation slicing for predicate detection in the subset of CTL [2] with the following three properties. First, temporal operators are $\mathbf{EF}$, $\mathbf{EG}$, and $\mathbf{AG}$. Second, atomic propositions are regular predicates, which we will define later. Third, negation operator has been pushed onto atomic propositions. We call this logic *Regular* CTL plus (RCTL+), where plus denotes that the disjunction and negation operators are included in the logic. We also consider a disjunction and negation free subset of RCTL+ and denote this by *Regular* CTL (RCTL). In RCTL+, we use the class of predicates, called *regular predicates*, that was introduced in [9]. The slice with respect to a regular predicate contains *precisely* those global states for which the predicate evaluates to true. Regular predicates widely occur in practice during verification. Some examples of regular predicates are conjunction of local predicates [8, 16] such as "all processes are in *red* state", certain channel predicates [8] such as "at most k messages are in transit from process P_i to P_j", and some relational predicates [8]. We show that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular when p is regular and present polynomial-time algorithms to compute slices with respect to these temporal predicates.

We show that the complexity of predicate detection for a predicate p in RCTL is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p. To the best of our knowledge, there did not previously exist efficient algorithms (polynomial in the number of processes) to detect predicates that 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. Furthermore, we validate with experiments that even for RCTL+ predicates our computation slicing based technique is very effective.

We also implemented our predicate detection algorithms for RCTL and RCTL+, which use computation slicing, in a prototype tool called Partial Order Trace Analyzer (POTA) [29]. We performed experiments using POTA on several protocols such as the Asynchronous Transfer Mode Ring (ATMR) [17]. ATMR protocol is an ISO standard based on a high-speed shared medium connecting a number of ac-

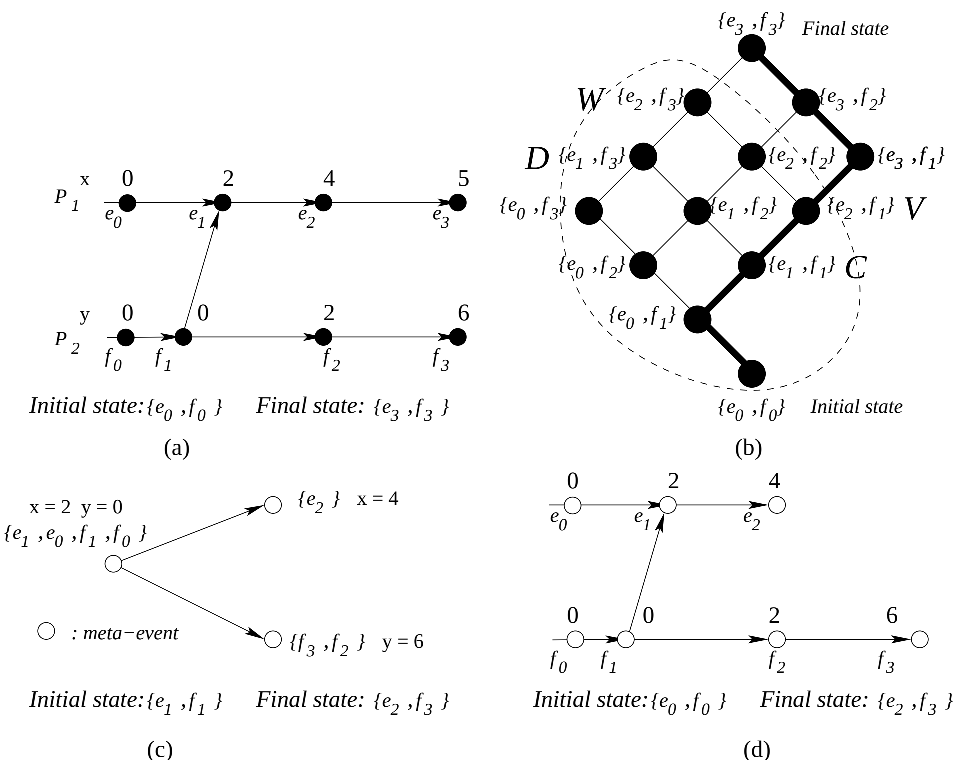


Figure 1. (a) A computation (b) its set of all reachable global states (c) its slice with respect to $(2 \leq x \leq 4) \wedge (y \neq 2)$ (d) its slice with respect to $\mathbf{EF}((2 \leq x \leq 4) \wedge (y \neq 2))$

cess nodes by channels in a ring topology. Peng et al. [25] performed experiments with this scalable protocol for different number of nodes in the ring. However, they could not complete full state space verification even for a configuration of ATMR with 3 nodes using SPIN [15] model checking tool. Instead, they used bit-state hashing approximation technique for verification upto 6 nodes. In our experiments, we could complete full state space verification of execution traces for a configuration with 250 nodes. We used the translator module in POTA that translates traces into Promela (SPIN input language) to enable comparison with SPIN on execution traces. Although, SPIN is designed for checking correctness of programs and not traces, to the best of our knowledge it is the best distributed program verification tool with effective reduction techniques that we can use for our models. Even with an execution trace input, SPIN failed to complete verification for configurations with more than 3 nodes in the case of ATMR. In all cases, our technique completes fast and uses state space efficiently.

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, whereas $\mathbf{AF}(p)$, $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$, and $\mathbf{AU}(p, q)$, in general, are not regular when p and q are regular.
- We present polynomial-time algorithms to compute slices for $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$, when p is regular.
- 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.

2. Example

To illustrate predicate detection using computation slicing, consider the computation in Figure 1(a). Let $p = (2 \leq$

$x \leq 4) \wedge (y \neq 2)$, and suppose we want to detect $\mathbf{EF}(p)$, that is, whether there exists a global state that satisfies p. 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 the slice of the computation with respect to the regular predicate $\mathbf{EF}(p)$ and use this slice for predicate detection. For this purpose, first we compute the slice with respect to the atomic proposition p as follows. Immediately after executing f_2, the value of y becomes 2 which does not satisfy $y \neq 2$. To reach a global state satisfying $y \neq 2$, f_3 has to be executed. In other words, any global state in which only f_2 has been executed but not f_3 is of no interest to us and can be ignored. The slice is shown in Figure 1(c). It is modeled by a partial order on a set of meta-events; each *meta-event* consists of one or more "primitive" events. A global state of the slice either contains all the events in a meta-event or none of them. Moreover, a meta-event "belongs" to a global state only if all its incoming neighbours are also contained in the state. The slice contains only four states C, D, V and W and has much fewer states than the computation itself – exponentially smaller in many cases – resulting in substantial savings. Using the slice in Figure 1(c), we can obtain *the* last state that satisfies p in the computation, which is denoted by W. We also know from the definition of $\mathbf{EF}(p)$ that every global state of the computation that occurs before W satisfies $\mathbf{EF}(p)$, e.g. states enclosed in the dashed ellipse in Figure 1(b). Therefore, applying this observation we can compute the slice with respect to $\mathbf{EF}(p)$ as shown in Figure 1(d), where the slice and the computation have the same consistent cuts upto W. Finally, we check whether the initial state of the computation is the same as the initial state of the slice. If the answer is yes then the predicate is satisfied, otherwise not.

Computation slicing can indeed be used to facilitate predicate detection even for a larger class of predicates than RCTL+ as illustrated by the following example. Consider a predicate p that is a conjunction of two clauses p_1 and p_2. Now, assume that p_1 is such that it belongs to RCTL+ but p_2 has no structural property that can be exploited for efficient detection, such as, $(x_1 * x_2 + x_3 > x_4)$, where x_i is an integer variable on process i. To detect p, without computation slicing, we are forced to use global-state-space-construction-based approaches, which do not take advantage of the fact that p_1 can be detected efficiently. With computation slicing, however, we can first compute the slice for p_1. If only a small fraction of global states satisfy p_1, then instead of detecting p in the computation, it is much more efficient to detect p in the slice. Therefore by spending only polynomial amount of time in computing the slice we can throw away exponential number of global states, thereby obtaining an exponential speedup overall. This also shows that our approach is orthogonal to previous reduction techniques.

3. Related Work

Predicate detection is a hard problem. Detecting even a 2-CNF predicate under $\mathbf{EF}$ modality has been shown to be NP-complete, in general [23]. Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [8, 16], *disjunctive* [8], *observer-independent* [1, 8], *linear* [8, 26], and *non-temporal regular* [9, 22] 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 under the *until* operator of CTL [26], 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 efficiently detected using only the efficient algorithms for conjunctive predicates. With the results of this paper, we can detect such nested temporal logic predicates efficiently.

The idea of using temporal logic for analyzing execution traces (also referred to as runtime verification) has recently been attracting a lot of attention. We first presented a temporal logic framework for partially ordered execution traces in [26] and gave efficient algorithms for predicates of the form $\mathbf{EG}(p)$ and $\mathbf{AG}(p)$ when p is a linear predicate. The efficiency of those algorithms depended on the fact that p was a state predicate and therefore we could efficiently evaluate the satisfiability of p at a global state. However, in this paper we present implementation of efficient algorithms even when p is a temporal predicate.

Some other examples of using temporal logic for checking execution traces are the commercial Temporal Rover tool (TR) [4], the MaC tool [18], the JPaX tool [13], and the JMPaX tool [30]. TR allows the user to specify the temporal formula in programs. These temporal formula are translated into Java code before compilation. The MaC and JPaX tools consider a totally ordered view of an execution trace and therefore can potentially miss bugs that can be deduced from the trace. LTL based verification of execution traces use automata generation [10, 7] or rewriting [14], where the verification complexity is polynomial time yet the representation model is a total order. JMPaX tool is closer to our tool POTA [29] because of the partial order trace model. We work with message passing distributed programs, whereas JMPaX considers multithreaded shared memory Java programs. However, the input to our techniques is a partial order therefore we can easily use the partial order traces generated by JMPaX and consequently extend the applicability of our technique to multithreaded Java programs. JMPaX uses a subset of temporal logic with safety where atomic propositions can be arbitrary. Whereas we use a subset of temporal logic with both safety and liveness where atomic propositions are restricted. The complexity of the predicate detection algorithm in our approach is polynomial-time whereas the complexity is exponential-time in JMPaX.

4. 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.

Traditionally, a distributed computation is modeled as a partial order on a set of events, called happened-before relation [19]. The *happened-before* relation between any two "primitive" events e and f can be formally stated as the smallest relation such that e happened-before f if and only if e occurs before f in the same process, or e is a send of a message and f is a receive of that message, or there exists an event g such that e happened-before g and g happened-before f. In this paper we relax the restriction that the order on events must be a partial order. More precisely, we use directed graphs to model distributed computations as well as slices. Directed graphs allow us to handle both of them in a uniform and convenient manner.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote the set of vertices and edges, respectively. We define a *consistent cut* (global state) on directed graphs as a subset of vertices such that if the subset contains a vertex then it contains all its incoming neighbours. Formally, C is a consistent cut of G, if $\forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) \wedge (f \in C) \Rightarrow (e \in C)$. 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 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. A distributed computation in our model can contain cycles. This is because whereas a computation in the happened-before model captures the *observable* order of execution of events, a computation in our model captures the set of possible consistent cuts. Intuitively, each strongly connected component of a computation can be viewed as a *meta-event*; a meta-event consists of one or more primitive events.

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 (happened-before model) is a non-trivial consistent cut of the computation in our model and vice versa. 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.

Figure 2 shows a computation and its lattice of (non-trivial) consistent cuts.

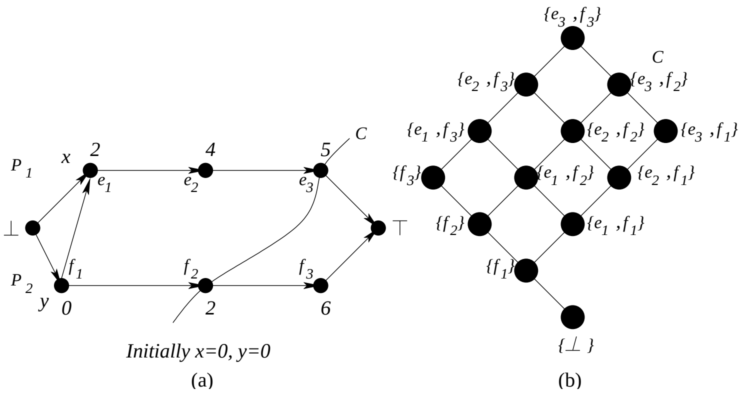


Figure 2. (a) A computation $\langle E, \rightarrow \rangle$ (b) and its lattice corresponding to $\mathcal{C}(G)$

Given a consistent cut, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. If a predicate p evaluates to true for a consistent cut C, we say that C satisfies p. We leave the predicate undefined for the trivial consistent cuts. A global predicate is $local$ if it depends on variables of a single process. We will define temporal logic predicates in Section 6.

5. Background on Slicing and Regular Predicates

The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [3]. The readers who are not familiar with earlier papers on slicing in [9, 22] are urged to read Appendix 10.2. Roughly speaking, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. More precisely,

Definition 1 (slice [22]) *A* slice *of a computation with respect to a predicate is a directed graph with the least num-*

ber of consistent cuts that contains all consistent cuts of the given computation for which the predicate evaluates to true.

We denote the slice of a computation $\langle E, \rightarrow\rangle$ with respect to a predicate p by $\mathsf{slice}(\langle E, \rightarrow\rangle, p)$. Note that $\langle E, \rightarrow\rangle = \mathsf{slice}(\langle E, \rightarrow\rangle, \mathsf{true})$. Every slice derived from the computation $\langle E, \rightarrow\rangle$ has the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts [22]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called *lean*. We next give the class of predicates for which the slice is lean.

Given a computation, the set of consistent cuts satisfying a regular predicate forms a sublattice of the set of consistent cuts of the computation [9]. Equivalently,

Definition 2 (regular predicate [9]) *A predicate is regular if given two consistent cuts that satisfy the predicate, the consistent cuts obtained by their set union and set intersection also satisfy the predicate. Formally, given a regular predicate p,*
(C satisfies p) $\wedge$ (D satisfies p) $\Rightarrow$ ($C \cap D$ satisfies p) $\wedge$ ($C \cup D$ satisfies p)

We say that a regular predicate is *non-temporal* if it does not contain temporal operators such as $\mathbf{EF}$, $\mathbf{AG}$, and $\mathbf{EG}$, otherwise it is a *temporal* regular predicate.

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 such as "P_i and P_j are in critical section", and relational predicates such as $x_1 - x_2 \leq 5$, where x_i is a monotonically non-decreasing integer variable on process i. From the definition of a regular predicate we deduce that a regular predicate has a least satisfying cut and a greatest satisfying cut. Furthermore, the class of regular predicates is closed under conjunction.

Also in [22] polynomial-time algorithms are given to compute slices with respect to boolean combination of regular predicates. Given the slices with respect to two regular predicates, the complexity of computing the slice for the conjunction and disjunction of these regular predicates is $O(n^2|E|)$. The complexity of computing the slice for the negation of a regular predicate is $O(n^2|E|^2)$. 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.

We now give a formal definition of RCTL+ which uses regular predicates as atomic propositions.

6. RCTL+ 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 $e \cap C = \emptyset$. 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 $0 \leq i < k$, $C_i \triangleright C_{i+1}$. We say that a cut D is *reachable* from a cut C if $C \subseteq D$.

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 $\lambda: \mathcal{C}(G) \rightarrow 2^{AP}$ assigns to each global state the set of predicates from AP that hold in it. In this paper we assume that atomic propositions are non-temporal regular predicates and their negations.

The formal syntax of RCTL+ is given below.

- Every predicate $ap \in AP$ is an RCTL+ formula.
- If p and q are RCTL+ formulas, then so are $p \vee q$, $p \wedge q$, $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$.

Given a finite distributive lattice $L = (\mathcal{C}(G), \subseteq)$, the formulas of RCTL+ are interpreted over the consistent cuts in $\mathcal{C}(G)$. Let p be an RCTL+ 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.
- $C \models p \wedge q$ 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 $0 \leq i \leq k$.

We define $L \models p$ if and only if $L, \{\bot\} \models p$. 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 every cut of the sequence. The formula $C \models \mathbf{EF}(p)$ intuitively means that for some consistent cut sequence $C, \ldots, \mathcal{E}$, there exists a consistent cut that satisfies p.

We define RCTL as the subset of RCTL+ where disjunction and negation operators are not allowed.

The *predicate detection* problem is to decide whether the initial consistent cut of a distributed computation satisfies a predicate.

Note that full CTL contains $\mathbf{X}$ and $\mathbf{U}$ operators, to denote *next-time* and *until* modalities. However, we are not going to explain them here due to space limitations.

7. 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. These results enable us to compute lean slices for these temporal predicates.

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 and H satisfy $\mathbf{EF}(p)$ (resp. $\mathbf{AG}(p)$, $\mathbf{EG}(p)$) then $(D \cap H)$ and $(D \cup H)$ 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' that satisfy p and $D \subseteq D'$ and $H \subseteq H'$. Furthermore we have that $D' \subseteq W$ and $H' \subseteq W$ where W is the greatest cut satisfying p. Then we have $(D \cap H) \subseteq W$ and $(D \cup H) \subseteq W$. Therefore $(D \cap H)$ and $(D \cup H)$ both 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 $(D \cup H)$ for D' in the previous observation we have $D \subseteq (D \cup H) \subseteq \mathcal{E}$. Thus $(D \cup H)$ satisfies $\mathbf{AG}(p)$. Now we show that $(D \cap H)$ satisfies $\mathbf{AG}(p)$. Consider an arbitrary consistent cut J such that $(D \cap H) \subseteq J$. Since $D \subseteq (D \cup J) \subseteq \mathcal{E}$ and $H \subseteq (H \cup J) \subseteq \mathcal{E}$, both $(D \cup J)$ and $(H \cup J)$ satisfy p. Since p is regular, it is easy to show that $J = (D \cup J) \cap (H \cup J)$ satisfies p. Therefore $(D \cap H)$ 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, $(H_0 \cup D_0), (H_0 \cup D_1), \ldots, (H_0 \cup D_k)$ and $(H_0 \cap D_0), (H_0 \cap D_1), \ldots, (H_0 \cap D_{k-1}), (H_0), \ldots, (H_m)$ are also such sequences. Therefore $(D \cap H)$ and $(D \cup H)$ satisfy $\mathbf{EG}(p)$. ■

However, the regularity does not follow in the case of $\mathbf{AF}(p)$, $\mathbf{EX}(p)$, $\mathbf{AX}(p)$, $\mathbf{EU}(p, q)$ and $\mathbf{AU}(p, q)$. We now give an example in Figure 3 where consistent cuts D and H satisfy $\mathbf{AF}(p)$ but their intersection $(D \cap H)$ does not. This is because there exists a path starting from $(D \cap H)$ and ending at the final cut $\mathcal{E}$ where p never holds on the path. We present examples for the other temporal predicates in the extended version of the paper [27].

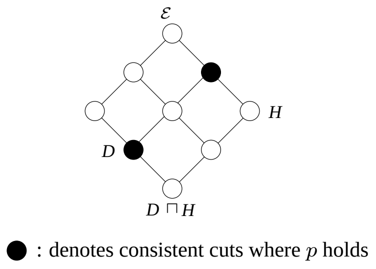


Figure 3. $\mathbf{AF}(p)$ may not be regular

8. Computation Slices for Temporal Predicates

In this section, we describe computation slicing algorithms for *temporal* regular predicates to enable *efficient* predicate detection for RCTL+. Earlier, Mittal and Garg [9, 22] presented computation slicing algorithms for *non-temporal* regular predicates, which they use to detect predicates such as $\mathbf{EF}(p)$, $\mathbf{EG}(p)$ and $\mathbf{AG}(p)$. We present algorithms for *temporal* regular predicates, which we use to detect *nested* temporal predicates such as $\mathbf{EG}(p \wedge \mathbf{EF}(q))$.

Since the consistent cuts of the slice of a computation is a subset of consistent cuts of the computation, the slice can be obtained by adding edges to the computation. In other words, the slice contains *additional edges* that do not exist in the computation. These additional edges may generate strongly connected components in the slice. For example, consider Figure 6(a) that displays the slice of the computation in Figure 2 with respect to $\neg((x = 5) \wedge (y = 2))$. The only consistent cut in the computation that does not satisfy the predicate is $\{e_3, f_1\}$. By adding the edge (f_2, e_3), we disallow this consistent cut from the slice. Similarly, since the consistent cuts of the slice for $\mathbf{AG}(p)$ is a subset of consistent cuts of the slice for p, the slice for $\mathbf{AG}(p)$ can be obtained by adding edges to the slice for p. Below, we will show which edges we should add to a computation (resp. to the slice for p) for computing slices for $\mathbf{EF}(p)$ (resp. for computing slices for $\mathbf{AG}(p)$).

8.1. Slicing Algorithms

Now we explain Algorithm A1 in Figure 5 for generating the slice of a computation with respect to $\mathbf{EF}(p)$. From the definition of $\mathbf{EF}(p)$, all consistent cuts of the computation that can reach the greatest consistent cut that satisfy p, say W, will also satisfy $\mathbf{EF}(p)$ and furthermore these are the only cuts that satisfy $\mathbf{EF}(p)$. We can find the cut W using $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ when it is nonempty. We construct the slice for $\mathbf{EF}(p)$ from the computation so that the slice has the same consistent cuts as the computation upto the final cut of the slice W. To ensure that all cuts that cannot reach W do not belong to the slice, we add edges from $\top$ to the successors of events in the frontier of W in the computation. Adding an edge from $\top$ to an event makes any cut that contains the event trivial. Figure 4 shows the application of Algorithm A1. Given the slice of the computation in Figure 2(a) for some predicate p as shown in Figure 4(a), first we compute the final cut of the slice for p, that is, $\{e_2, f_3\}$. Then, on the computation, we add an edge from $\top$ to the successor of e_2, that is e_3. The successor of f_3 does not exist so we do not add any other edges. The resulting slice for $\mathbf{EF}(p)$ is displayed in Figure 4(c).

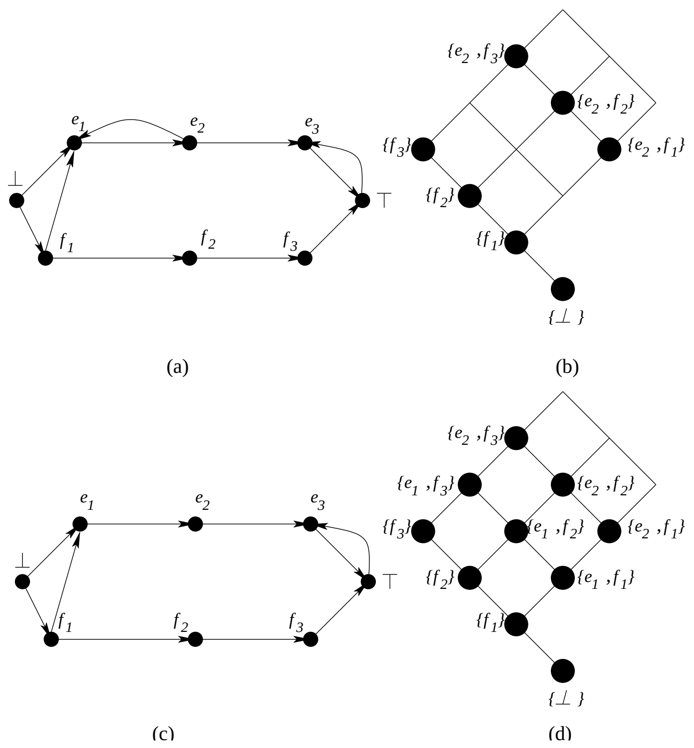


Figure 4. (a) A slice of $\langle E, \rightarrow\rangle$ in Fig. 2 (b) the corresponding sublattice (c) The application of the temporal operator $\mathbf{EF}$ on the slice in (a) (d) the corresponding sublattice

In Step 1 we can find the final cut of slice$(\langle E, \rightarrow\rangle, p)$ using the strongly connected components in $O(n|E|)$. Steps 2 to 4 take $O(n)$ time. The overall complexity is $O(n|E|)$.

Algorithm A2 in Figure 5 generates the slice for $\mathbf{AG}(p)$. We explained above that to obtain the slice for $\mathbf{AG}(p)$ we will add edges to the slice for p and therefore eliminate consistent cuts that do not belong to the slice for $\mathbf{AG}(p)$. We claim that consistent cuts of slice for 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). Consider consistent cuts $\{\perp\}$, $\{f_1\}$, $\{e_1, f_1\}$, and $\{e_2, f_1\}$ of the slice in Figure 6(a) that do not include vertex f_2 of the additional edge (f_2, e_3). It is easy to see that these four consistent cuts do not satisfy $\mathbf{AG}(p)$. This is because there exists a consistent cut $\{e_3, f_1\}$ in the computation that does not satisfy p, yet which is reachable from these four consistent cuts. We now give a proof sketch of the correctness of the algorithm for the simplified case with a single additional edge. Due to space limitations, the formal proof of correctness of the algorithms in this section are given in the extended version of the paper in [27].

Algorithm A1

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

Algorithm A2

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

Algorithm A3

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

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

Theorem 2 *Given a computation $\langle E, \rightarrow\rangle$ and* slice$(\langle E, \rightarrow\rangle, p)$*, a consistent cut D in $\langle E, \rightarrow\rangle$ satisfies $\mathbf{AG}(p)$ iff it includes vertex e of the additional edge (e, f) in* slice$(\langle E, \rightarrow\rangle, p)$.
Proof Sketch:

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 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 slice$(\langle E, \rightarrow\rangle, p)$ 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. ■

For example, the slice in Figure 6(a) has an additional edge (f_2, e_3) so we add the edge $(f_2, \perp)$ and obtain the slice for $\mathbf{AG}(p)$ as in Figure 6(c).

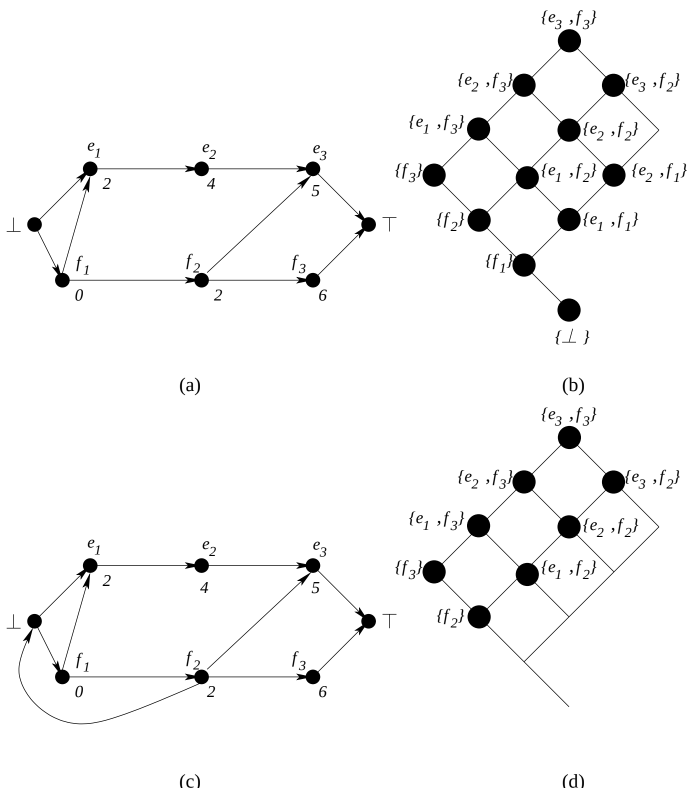


Figure 6. (a) The slice of $\langle E, \rightarrow\rangle$ in Fig. 2 with respect to $\neg((x = 5) \wedge (y = 0))$ (b) the corresponding sublattice (c) The slice of $\langle E, \rightarrow\rangle$ in Fig. 2 with respect to $\mathbf{AG}\ \neg((x = 5) \wedge (y = 0))$ (d) the corresponding sublattice

In Step 2 we can add edges for each additional edge in slice$(\langle E, \rightarrow\rangle, p)$. From [22], there are $O(n|E|)$ such edges when the skeletal representation of a slice is used. Therefore, the overall complexity is $O(n|E|)$.

The algorithm for $\mathbf{EG}(p)$ slicing displayed in Figure 5 is similar to the $\mathbf{AG}(p)$ slicing algorithm. However in this case, for each additional edge (e, f) *that generates a non-trivial strongly connected component* in slice$(\langle E, \rightarrow\rangle, p)$, we add an edge from the vertex e to the vertex $\perp$. Intuitively, due to such a strongly connected component on all paths from the initial to the final state in the computation there exists a cut that does not satisfy p.

8.2. Predicate Detection using Slicing

Figure 7 displays our predicate detection algorithm that uses slicing algorithms developed in this section. The complexity of predicate detection for RCTL is dominated by the complexity of computing the slice with respect to a non-temporal regular predicates, which has $O(n^2|E|)$ complexity [9, 22]. Therefore, the overall complexity of predicate detection for RCTL is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p.

When the predicate does not belong to RCTL (that is, it contains disjunction or negation operators) the slice may not be lean. In this case, we may have to take an extra step. This is because the initial state of the slice may not satisfy the predicate. Therefore, we employ the translator module of POTA and translate the slice into Promela [15] . Then we use SPIN to check the trace assuming that there are equivalent specifications in LTL. Using SPIN may lead to exponential-time complexity for RCTL+. However, the slice is in general much smaller than the computation and therefore we still have efficient verification, which we validate with experiments in the next section.

```
Input:   A computation $\langle E, \rightarrow\rangle$ and an RCTL+ predicate $p$
Output:  Predicate is satisfied or not
1.       Recursively process $p$ from inside to outside
         while applying temporal and boolean operators
         to compute slices
2.       If initial($\langle E, \rightarrow\rangle$) $\neq$ initial(slice($\langle E, \rightarrow\rangle, p$)) then
3.           return false and counterexample
         else
4.         if $p$ does not contain $\neg$ or $\vee$ then
5.           return true
6.         else run SPIN on the translated slice($\langle E, \rightarrow\rangle, p$)
```

Figure 7. Predicate Detection using Slicing

9. Experimental Results

We implemented our algorithms using Java in a prototype tool called Partial Order Temporal Analyzer (POTA)

[29]. POTA consists of analyzer, translator and instrumentor modules. We have implemented slicing based predicate detection algorithms in the analyzer module. Predicate detection algorithms from our previous research has not been implemented yet. In order to evaluate the effectiveness of POTA, we performed experiments with scalable protocols, comparing our computation slicing based approach with partial order reduction based approach of SPIN [15]. The translator module takes an execution trace and generates Promela code. Currently, the instrumentation is being done manually. The instrumented program is such that when run every process outputs its local state where a local state contains the values of variables relevant to the predicate in question and a vector clock that is updated for each internal, send and receive event according to the Fidge/Mattern [6, 20] algorithm. Vector clocks enable us to obtain a partial order representation of traces. All experiments were performed on a 1.4 Ghz Pentium 4 machine running Linux. We restricted the memory usage to 512MB, but did not set a time limit. The two performance metrics we measured are running time (T in seconds) and memory usage (M in megabytes). We run the programs for 20 seconds and our measurements are averaged over 20 traces for each program. Further experimental results can be obtained from POTA website [28] for protocols such as General Inter-ORB Protocol (GIOP), distributed dining philosophers and leader election.

First, we use the Distributed Mutual Exclusion protocol from [12] in Java and check the complement of the liveness property, that is, $\bigvee_i \Big(\mathbf{EF}\big(tryCS_i \wedge \mathbf{EG}(\neg inCS_i)\big)\Big)$, for each process i. Observe that the negation of a local predicate $\neg inCS_i$ is also a local predicate and furthermore it is a regular predicate.

Next, we perform experiments for the primary secondary program [32], which 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 $\mathbf{EF} \bigwedge \Big(\neg isPrimary_i \vee \neg isSecondary_j \vee (secondary_i \neq P_j) \vee (primary_j \neq P_i)\Big)$ when $i, j \in 0 \ldots (n-1), i \neq j$. Note that this predicate contains disjunction operators and the slice may not be lean.

Finally, we present experimental results for the Asynchronous Transfer Mode Ring (ATMR) protocol which was verified in [25] using SPIN.

ATMR protocol [17] is an ISO standard based on a high-speed shared medium connecting a number of access nodes by channels in a ring topology. For controlling access to this type of shared medium, the ring is first initialized with a fixed number of ATM cells continuously circulating around the channel from one node to another. Within each access node there is an access unit which performs both the physical layer convergence function and the access control function. Access to the ring is requested by the client and controlled by a combination of a window mechanism and a reset procedure. The client can issue a sending request to the access unit and receive a data cell. The window mechanism limits the number of cells a node can transmit at a time, called the "credits" of this node. The reset procedure reinitializes the window in all access units to a predefined credit value.

We conducted experiments for the following predicates used in [25].

1. Once an access unit exhausts its window size credit, the credit will eventually be renewed.
   $\mathbf{EF}\Big((credit_i == 0) \wedge \mathbf{EG}(\neg(credit_i == 6))\Big)$, for all access units i, where $credit$ stands for the number of credits which is being held by an access unit and 6 is the preset maximum value.
2. A client's request will be eventually acknowledged.
   $\mathbf{EF}\Big(req_i \wedge \mathbf{EG}(\neg ack_i)\Big)$, for all clients i, where req is a cell sending request signal from a client to an access unit. If the requested cell has been sent out, the access unit will return an ack signal to the client.
3. An access unit will eventually exit $Reset$ state and enter the $Send$ state.
   $\mathbf{EF}\Big(Reset_i \wedge \mathbf{EG}(\neg Send_i)\Big)$, for all access units i.
4. An access unit will eventually exhaust its window size credit.
   $\mathbf{EF}\Big((credit_i == 6) \wedge \mathbf{EG}(\neg(credit_i == 0))\Big)$, for all access units i.

The full state space verification of ATMR by Peng et al. [25] even for a configuration with 3 nodes was not completed due to state explosion. To enable verification for larger number of processes, they used an approximation technique in SPIN called *bit-state hashing* where two bits of memory are used to store a reachable state. With bit-state hashing, they could verify upto 6 nodes on a 2GB memory machine with less than 98 percent coverage.

We generated execution traces for upto 250 nodes and completed full state space verification of these traces. Whereas, SPIN failed to complete full state space verification for more than 3 nodes even when the input were traces rather than the protocol. Similarly, we verified execution traces with 100 processes for mutual exclusion protocol and for 40 processes for primary secondary protocol, whereas SPIN failed to complete for more than 5 processes and 10 processes, respectively.

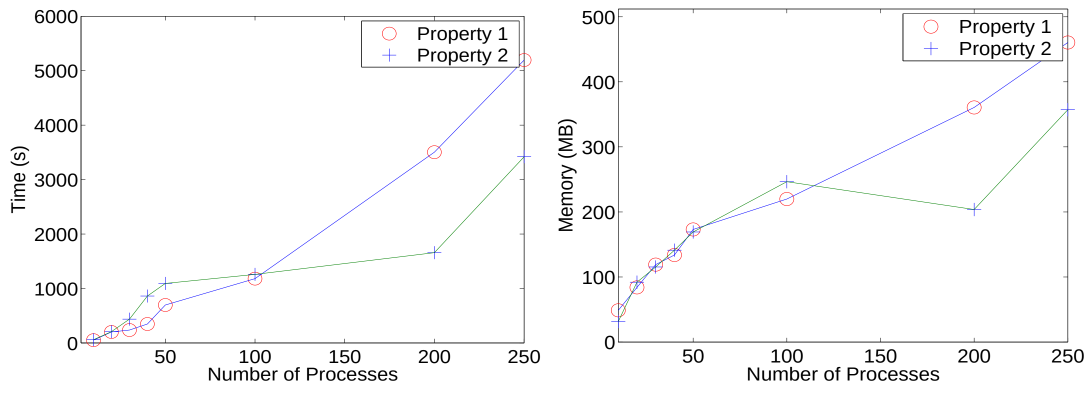


Figure 8. ATMR verifi cation results, SPIN runs out of memory for > 3 processes

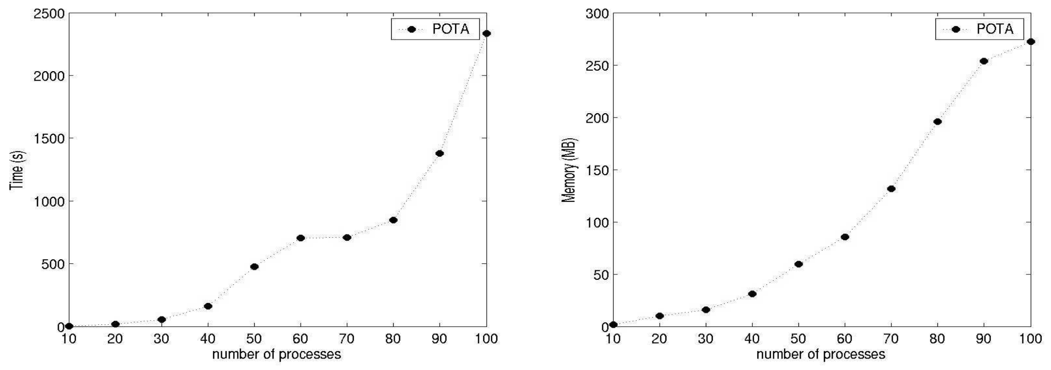


Figure 9. Mutual Exclusion verifi cation results: SPIN runs out of memory for > 6 processes

In Figure 8, we present experimental results for the first two properties of ATMR. Our results for mutual exclusion and primary secondary protocols are displayed in Figure 9 and Figure 10, respectively. We use logscale for time and memory in Figure 10, which shows that even in the case of predicates with disjunction operator slicing can reduce the state space substantially.

The experimental work proves that for large problem sizes, computation slicing is an effective technique.

References

[1] 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.

[2] 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 *LNCS*, Yorktown Heights, New York, May 1981.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[4] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Verifi cation*, volume 1885 of *LNCS*, pages 323–330, 2000.

[5] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994.

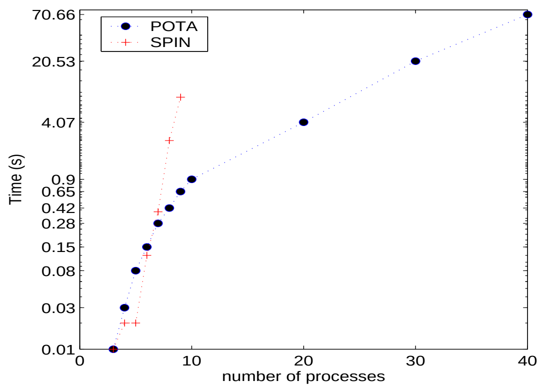
[6] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, Aug. 1991.

[7] B. Finkbeiner and H. B. Sipma. Checking Finite Traces using Alternating Automata. In *Runtime Verifi cation 2001*, volume 55 of *ENTCS*, pages 44–60, July 2001.

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

[9] 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.

[10] D. Giannakopoulou and K. Havelund. Automata-Based Verifi cation of Temporal Properties on Running Programs. In *Proc. of Int. Conference on Automated Software Engineering (ASE'01)*, pages 412– 417, San Diego, California, 2001.



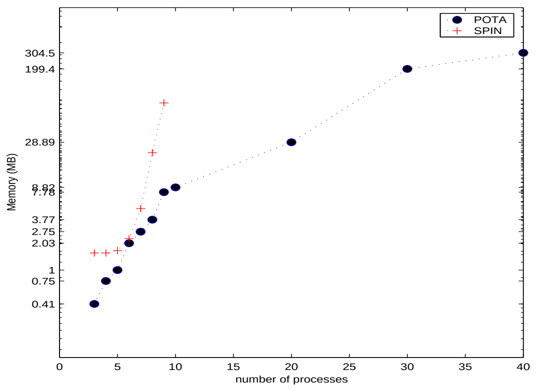


Figure 10. Primary Secondary verifi cation results: SPIN runs out of memory for > 10 processes

[11] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proc. of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

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

[13] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Runtime Verifi cation 2001*, volume 55 of *ENTCS*, 2001.

[14] K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Proc. of Int. Conference on Automated Software Engineering (ASE'01)*, pages 135–143, San Diego, California, 2001.

[15] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[16] M. Hurfi n, M. Mizuno, M. Raynal, and M. Singhal. Effi cient detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, 1998.

[17] ISO. Specifi cation of the Asynchronous Transfer Mode Ring (ATMR) Protocol. January 1993.

[18] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Runtime Verifi cation 2001*, volume 55 of *ENTCS*, 2001.

[19] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[21] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[22] 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.

[23] N. Mittal and V. K. Garg. 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.

[24] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *5th Int'l. Conference on Computer-Aided Verifi cation (CAV)*, pages 409–423. Springer, Berlin, Heidelberg, 1993.

[25] H. Pendex, S. Tahar, and F. Khendek. Comparison of SPIN and VIS for Protocol Verifi cation. *Software Tools for Technology Transfer*, 4(2):234–245, 2003.

[26] 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.

[27] A. Sen and V. K. Garg. Automatic Generation of Slices for Temporal Logic Predicate Detection. Technical Report TR-PDS-2003-001, PDSL, ECE Dept. Univ. of Texas at Austin, 2003. Available at http://maple.ece.utexas.edu/.

[28] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA). http://maple.ece.utexas.edu/˜sen/POTA.html, 2003.

[29] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Runtime Verifi cation 2003*, volume 89 of *ENTCS*, 2003.

[30] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. TR UIUCDCS-R-2003-2334, Univ. of Illinois at Urbana Champaign, Apr. 2003.

[31] S. D. Stoller and Y. Liu. Effi cient Symbolic Detection of Global Properties in Distributed Systems. In *10th Int'l. Conference on Computer-Aided Verifi cation (CAV)*, volume 1447 of *LNCS*, pages 357–368, 1998.

[32] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Effi cient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *12th Int'l. Conference on Computer-Aided Verifi cation (CAV)*, volume 1855 of *LNCS*, pages 264–279, 2000.

[33] A. Valmari. A Stubborn Attack On State Explosion. In *2nd Int'l. Conference on Computer-Aided Verifi cation (CAV)*, volume 531 of *LNCS*, pages 156–165, Berlin, Germany, 1990.

10 Appendix

10.1 Illustration of Concepts in Section 4

Example 1 *Consider the computation depicted in Figure 11(a). It has three processes, namely p_1, p_2 and p_3. The events e_1, f_1 and g_1 are the initial events, and the events e_4, f_4 and g_4 are the final events of the computation. The cut $A = \{e_1, e_2, e_3, e_4, f_1, g_1\}$ is not consistent because $g_4 \rightarrow e_4$ and $e_4 \in A$ but $g_4 \notin A$. On the other hand, the cut $\{e_1, e_2, f_1, f_2, g_1\}$ is a consistent cut. The events e_1, f_1 and g_1 belong to the same strongly connected component or meta-event. Processes p_1, p_2 and p_3 host integer variables x, y and z, respectively. The predicate $x \leq 1$ is a local predicate whereas the predicate $x + y \leq z$ is not. The consistent cut $\{e_1, f_1, g_1\}$ satisfies the predicate $x + y \leq z$ but the consistent cut $\{e_1, e_2, f_1, f_2, g_1\}$ does not.*

10.2 Birkhoff's Theorem

We first describe some concepts needed to understand the theorem. Given a lattice, its meet (infimum) and join (supremum) operators are denoted by $\sqcap$ and $\sqcup$, respectively. A lattice is *distributive* if meet distributes over join [3]. We call an element of a lattice *join-irreducible* if it cannot be expressed as join of two distinct elements (of the lattice), both different from itself [3]. Let L be a lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible elements. In case L is a distributive lattice, it satisfies an important property. Specifically, every element in L can be expressed as join of some subset of elements in $\mathcal{JI}(L)$ and vice versa [3, Birkhoff's Theorem]. In other words, $\mathcal{JI}(L)$ completely characterizes L. This is significant because $|\mathcal{JI}(L)|$ is generally much smaller—exponentially in many cases—than $|L|$. Hence if some computation on L can instead be performed on $\mathcal{JI}(L)$, we obtain a significant computational advantage.

Consider a computation $\langle E, \rightarrow \rangle$ and let $\mathcal{C}(E)$ denote the set of its consistent cuts. In [9], it was shown that $\mathcal{C}(E)$ forms a distributive lattice under the relation $\subseteq$; its join and meet operators correspond to set union ($\cup$) and set intersection ($\cap$), respectively. Furthermore, no additional structural property is satisfied by $\mathcal{C}(E)$. The set of join-irreducible elements of $\mathcal{C}(E)$ is isomorphic to the set of strongly connected components of $\langle E, \rightarrow \rangle$.

Now, consider a subset $\mathcal{D} \subseteq \mathcal{C}(E)$. We say that $\mathcal{D}$ forms a *sublattice* of $\mathcal{C}(E)$ if $\mathcal{D}$ is closed under set union and set intersection. That is, given two consistent cuts from $\mathcal{D}$, the consistent cuts obtained by their set union and set intersection also belong to $\mathcal{D}$. It can be proved that any sublattice of a distributive lattice is also a distributive lattice [3]. Thus if $\mathcal{D}$ is a sublattice of $\mathcal{C}(E)$, then, using Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$ completely characterizes $\mathcal{D}$. This forms the basis for the notion of computation slice.

Example 2 *Consider the computation shown in Figure 11(a). The (distributive) lattice spanned by its set of consistent cuts is shown in Figure 11(b). In the figure, each consistent cut is labeled with the number of events that have to be executed on each process to reach the cut. The join-irreducible elements of the lattice have been drawn with thick boundaries. (They have exactly one incoming edge.) The lattice has eight join-irreducible elements which is same as the number of strongly connected components of the computation. It can be verified that every consistent cut of the computation can be obtained as the join of some subset of these eight join-irreducible elements and vice versa. For instance, the consistent cut R (in Figure 11(b)) can be expressed as the join of the consistent cuts U and V.*

10.3 Establishing the Existence and Uniqueness of Computation Slice

It was proven in [22] that the slice exists and is uniquely defined for all predicates. The main idea behind the proof is as follows. Consider the computation $\langle E, \rightarrow \rangle$ and a predicate p. Let $\mathcal{C}(E)$ denote the set of consistent cuts of $\langle E, \rightarrow \rangle$ and further, let $\mathcal{C}_p(E) \subseteq \mathcal{C}(E)$ be the subset of those consistent cuts that satisfy p. We show that there exists a unique subset $\mathcal{D} \subseteq \mathcal{C}(E)$ satisfying the following conditions. First, $\mathcal{D}$ contains $\mathcal{C}_p(E)$, that is, $\mathcal{C}_p(E) \subseteq \mathcal{D}$. Second, $\mathcal{D}$ forms a sublattice of $\mathcal{C}(E)$. Last, among all sublattices that fulfill the first two conditions, $\mathcal{D}$ is the *smallest* one. From Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$, the set of join-irreducible elements of $\mathcal{D}$, completely characterizes $\mathcal{D}$. We call the poset (partially ordered set) induced on the consistent cuts of $\mathcal{JI}(\mathcal{D})$ by the relation $\subseteq$ as the slice of $\langle E, \rightarrow \rangle$ with respect to p. Each join-irreducible element gives rise to a meta-event. Alternatively, the slice can also be represented by a directed graph drawn on the set of events such that the set of consistent cuts of the graph is exactly $\mathcal{D}$. Such a graph can be obtained by simply forming a strongly connected component out of each meta-event. Whereas the *poset representation* of a slice is better for presentation purposes, the *graph representation* is more suited for slicing algorithms.

Example 3 *Consider the lattice of consistent cuts depicted in Figure 11(b). The consistent cuts that satisfy the predicate $x + y - z \leq 1$ have been shaded in the figure. Figure 11(c) depicts the smallest sublattice that contains these consistent cuts. The consistent cuts P and Q do not satisfy the predicate but have been included to complete the sublattice. The join-irreducible elements of the sublattice have been drawn with thick boundaries. There are, in total, seven join-irreducible elements, namely T, U, V, W, X, Y and Z. Figure 11(d) portrays the partial order induced on*

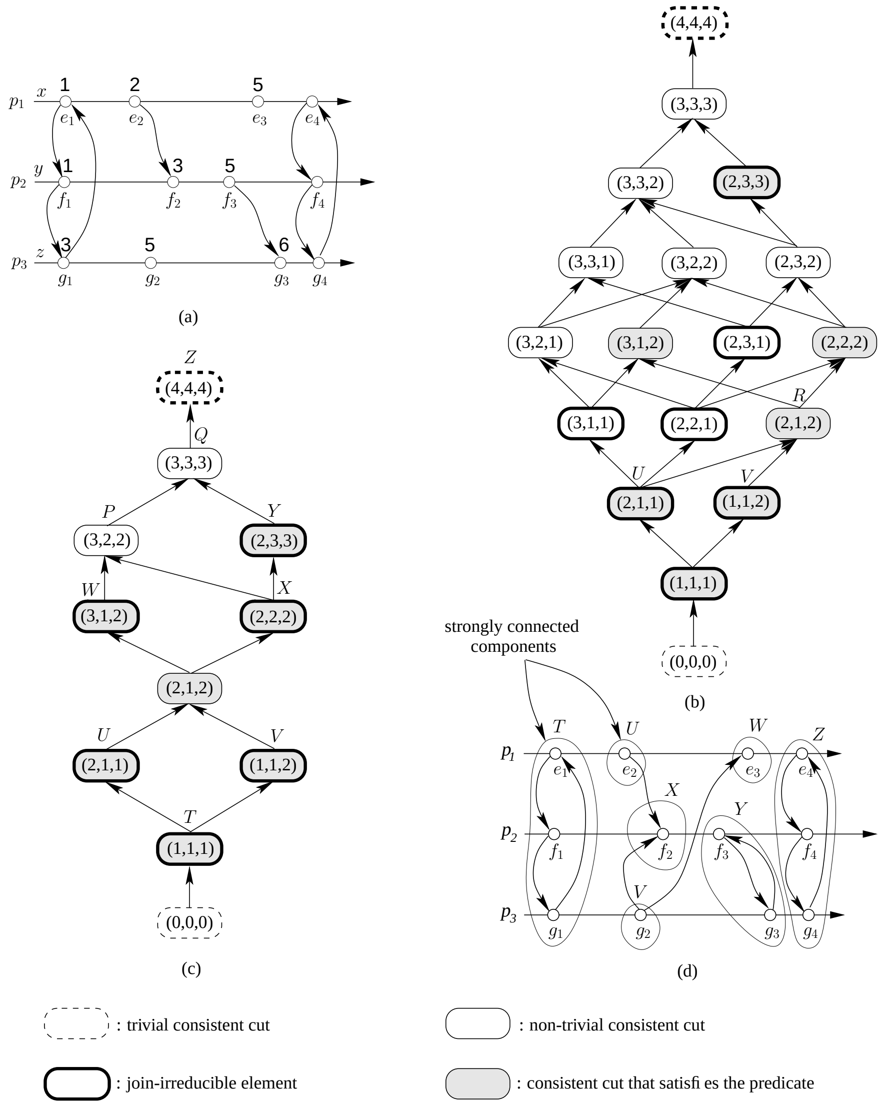

(a)
(b)
(c)
(d)
strongly connected components
: trivial consistent cut
: non-trivial consistent cut
: join-irreducible element
: consistent cut that satisfies the predicate

Figure 11. (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leq 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

the set $\mathcal{J} = \{T, U, V, W, X, Y, Z\}$. There is a one-to-one correspondence between the set of join-irreducible elements and the set of strongly connected components of the graph shown in Figure 11(d). It can be verified that every consistent cut in the sublattice can be expressed as join of some subset of $\mathcal{J}$ and, furthermore, the join of every subset of $\mathcal{J}$ is a consistent cut of the sublattice.