Partial Order Trace Analyzer (POTA) for Distributed Programs

Alper Sen and Vijay K. Garg

TR-PDS-2003-004 May 2003

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

Partial Order Trace Analyzer (POTA) for Distributed Programs *

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}

May 16, 2003

Abstract

Checking the correctness of software is a growing challenge. In this paper, we present a prototype implementation of Partial Order Trace Analyzer (POTA), a tool for checking execution traces of message passing programs using temporal logic. So far runtime verification tools have used the total order model of an execution trace, whereas POTA uses a partial order model. The partial order model enables us to capture possibly exponential number of interleavings and, in turn, this allows us to find bugs that are not found using a total order model. However, verification in partial order model suffers from the state explosion problem – the number of possible global states in a program increases exponentially with the number of processes.

POTA employs an effective abstraction technique called *computation slicing*. A slice of a computation (execution trace) 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 mitigates the state explosion problem by reasoning only on the part of the global state space that is of interest. We implemented computing slicing algorithms for temporal logic predicates from a subset of CTL in POTA. The overall complexity of evaluating a predicate in this logic upon using computation slicing becomes polynomial in the number of processes compared to exponential without slicing.

We illustrate the effectiveness of our techniques in POTA on several test cases such as the General Inter-ORB Protocol (GIOP) [23] and the primary secondary protocol [39]. POTA also contains a translator module from execution traces to Promela [21] (input language SPIN). This module

*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

enables us to compare our results on execution traces with SPIN. In some cases, we were able to verify traces with 250 processes compared to only 10 processes using SPIN.

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.

A finite trace can be modeled in two ways. The first model imposes a partial order between events, for example Lamport's *happened-before* relation [26]. 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 [26].

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. The set of all global states reachable from the initial state $\{e_0, f_0\}$ is displayed in Figure 1(b). 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 [29, 16, 40, 33, 9, 38, 39].

In this paper, we present a prototype implementation of Partial Order Temporal Analyzer (POTA) tool for checking execution traces of distributed programs. POTA consists of an instrumentation module, a translator module that translates execution traces into Promela [21] (SPIN input language) and an an-

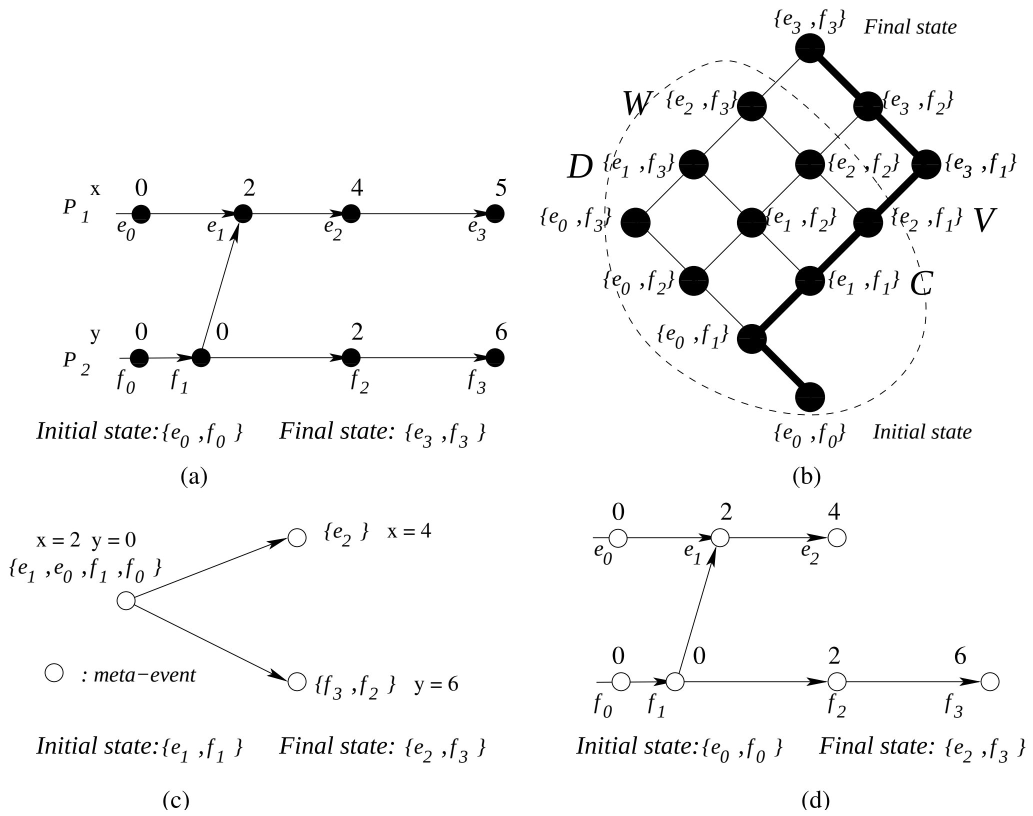

Figure 1: (a) A computation (b) its set of all reachable global states (c) its slice with respect to $(2 \leq x \leq 4) \wedge (y \neq 2)$ (d) its slice with respect to $\mathbf{EF}((2 \leq x \leq 4) \wedge (y \neq 2))$

alyzer module. The use of an effective abstraction technique called computation slicing for temporal logic verification is the most significant aspect of POTA and constitutes the analyzer module.

Computation slicing was introduced in [14, 31] 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. 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.

Using the results in [14, 31] and [35], we can efficiently use computation slicing for the subset of CTL [4] with the following three properties. First, temporal operators are **EF**, **EG**, and **AG** and boolean operators are conjunction and disjunction. 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 the 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 [14]. 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 [12, 22] such as "all processes are in *red* state", certain channel predicates [12] such as "at most k messages are in transit from process P_i to P_j", and some relational predicates [12].

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 $\mathbf{EF}(p)$. 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. Figure 1(b) contains the set of all reachable global states of the computation. In the figure, we represent a global state as a tuple where each element is the last event that occurred on a process. Alternatively, we can compute the slice of the computation with respect to 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). 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.

POTA implements predicate detection algorithms for RCTL and RCTL+ which use computation slicing. We show in [35], 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 exist tools that implement 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 performed experiments using POTA on several protocols. We also used the POTA translator module to enable comparison with SPIN on execution traces. In fairness, SPIN is designed for checking correctness of programs and not traces. However, to the best of our knowledge it is the best distributed program verification tool we can use for our partial order models. Some of the protocols we used for experiments are the General Inter-ORB Protocol (GIOP) [23] and the primary secondary protocol [39]. GIOP is a central feature of the Common Object Request Broker Architecture (CORBA) that aids in achieving the desired interoperability between ORBs. The CORBA specification defines a standard protocol to allow communication of object invocations between ORBs. Kamel and Leue [23] could not fully verify an abstract model of GIOP with 10 processes. Instead, they verified a simplified version of the protocol without server migration functionality. In one case, we generated execution traces of unsimplified GIOP protocol for a configuration with 250 processes. However, even with an execution trace input, SPIN failed to complete verification with more than 10 processes. We also injected faults into the protocol and analyzed the resulting execution traces. With SPIN, we used bit-state hashing approximation option to handle larger number of processes, but in this case SPIN failed to find the faults before running out of memory. However, POTA was able to find the faults easily. In all cases, our algorithms are significantly faster and space efficient than SPIN. We have measured over three orders of magnitude gain over SPIN some experiments.

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.

The remainder of this paper is organized as follows. In the next section, we compare our work with related work in this area. Section 3 presents an overview of the POTA architecture. In Section 4, we discuss the computation model that is used to represent distributed programs. A brief overview of computation slicing is given in Section 5. Section 6.1 describes the regular predicates and our logic. Section 7 describes the underlying temporal logic computation slicing algorithms. We give experimental results in Section 8. Finally, some conclusions and a description of future work are given in Section 9.

2 Related Work

The notion of a computation slice is similar to the concept of a program slice [41]. 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 atomic propositions in temporal logic LTL [8]. Millett and Teitelbaum [30] have applied program slicing to Promela. Program slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance [25, 41]. 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 hard problem. Detecting even a 2-CNF predicate under **EF** modality has been shown to be NP-complete, in general [13].

Predicate detection is a widely-studied problem. There are three major approaches to solving predicate detection: global-snapshot-based approach [2], global-state-space-construction-based approach (including model checking) [4, 5], and predicate-restriction-based approach [12]. 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. 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* [12, 22], *disjunctive* [12], *stable* [2], *observer-independent* [3, 12], *linear* [12, 36], and *non-temporal regular* [14, 31] predicates. These predicate classes have been so far detected under some or all of the temporal operators **EF**, **EG**, **AG**, **AF** and under the *until* operator of CTL [36], 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. In POTA, 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 [36] 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) [7], the MaC tool [24], the JPaX tool [19], and the JMPaX tool [37]. 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. Other LTL based verification of execution traces are based on automata generation [11] or rewriting [20], where the verification complexity is polynomial time for full LTL yet the representation model is a total order.

JMPaX tool is closer to POTA because of the partial order trace model. The differences in both approaches can be summarized as follows. POTA works with message passing distributed programs, whereas JMPaX considers multithreaded shared memory Java programs. JMPaX uses a subset of temporal logic with safety where atomic propositions can be arbitrary. Whereas POTA uses a subset of temporal logic with both safety and liveness where atomic propositions are restricted. The complexity of the predicate detection algorithm in POTA is polynomial-time whereas the complexity is exponential-time in JMPaX.

3 Overview of POTA Architecture

The overall structure of POTA architecture is shown in Figure 2. The tool consists of 3 main modules; analyzer, translator, and instrumentor.

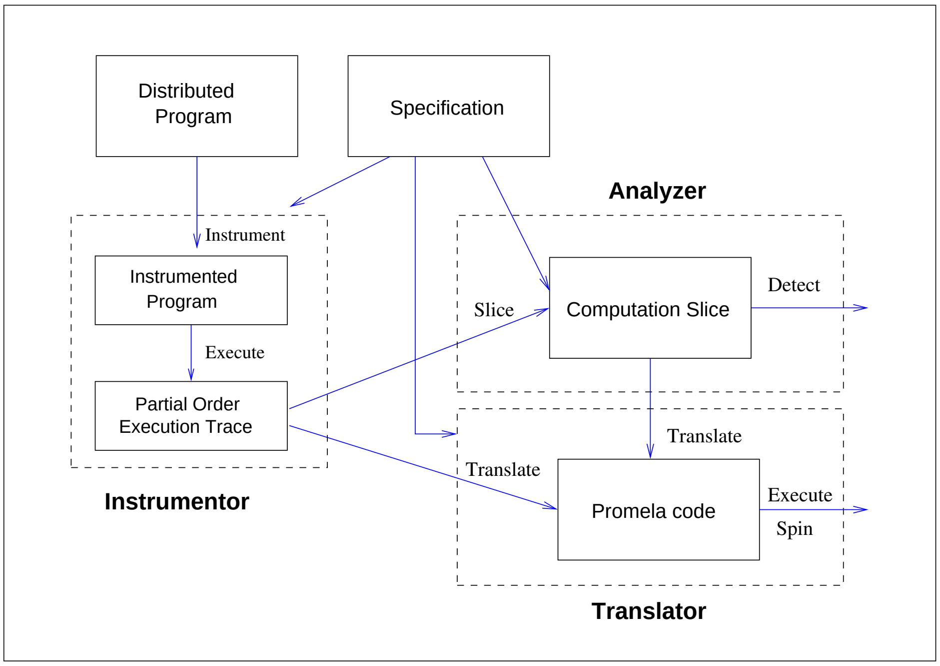

Figure 2: Overview of POTA Architecture

The *analyzer* module contains our computation slicing and predicate detection algorithms. Given an execution trace and a predicate (specification) in

RCTL+, the computation slice may contain more states than the ones that satisfy the predicate. Therefore, the analyzer module uses the following strategy to decide whether the predicate is satisfied or not. Case 1, if the slice and the input trace have different initial states then the predicate is not satisfied. In this case a counterexample is generated. Case 2, if the predicate is from RCTL and the slice and the input trace have the same initial states then the predicate is satisfied. Case 3, when the predicate does not belong to RCTL (that is, it contains disjunction or negation operators) and the slice and the input trace have the same initial states then we have to take an extra step. This is because the initial state of the slice may not satisfy the predicate. Therefore, we employ the translation module and translate the slice into Promela [21] (input language of SPIN). Then we use SPIN to check the trace assuming that there are equivalent specifications in LTL.

The *translator* module takes a partial order representation of a trace and generates output in specific languages. This module serves two purposes; to enable comparison of our slicing technique with other techniques such as partial order reduction and to enable verification of predicates that do not belong to RCTL but for which we can take advantage of computation slicing. The latter purpose is served when the predicate belongs to RCTL+ as explained in Case 3 in the above paragraph or when the predicate is a conjunction of predicates where one of the conjuncts belong to RCTL+ as explained in the introduction. Since we are working with distributed programs which exhibit a lot of parallelism and independency, partial order reduction techniques can take advantage of these properties of distributed programs. The SPIN model checker contains implementation of partial order reduction techniques. Currently, translation from traces to Promela is supported. The translation mechanism is similar to the technique explained in [27] for translations from message sequence charts (MSC) to Promela.

The *instrumentation* module inserts code at the appropriate places in the program to be monitored. The instrumented program is such that it outputs the values of variables relevant to the predicate in question and keeps a vector clock that is updated for each internal, send and receive event according to the Fidge/Mattern algorithm [10, 28]. We use the vector clock to obtain a partial order representation of traces.

Upon running the instrumented program a separate log file for each process is generated. Each log file consists of a sequence of local states that a process goes through. Each local state contains the values of variables relevant to the predicate being verified and a vector clock. Log files for every process are then combined to obtain a partial order representation of the execution trace.

Instead of using a log file, if every process sends its trace to a dedicated process which combines them during runtime, we can obtain an *on-line* verification environment.

Currently, programs are manually instrumented. We conducted experiments with Java and Promela programs. For SPIN programs, we insert code into Promela programs and also made changes to the SPIN source code so that we can obtain a partial order model when we run SPIN in simulation mode with the option for generating a message sequence chart output. SPIN's MSC output is by default a total ordered execution. However, we observed from this MSC output that there are unnecessary dependencies therefore events do not need to be totally ordered such as request messages from two different processes sent to two different servers do not need to be totally ordered.

We are in the process of choosing an appropriate instrumentation technique for Java programs. The choice is between Java JDI as in [1] or byte code instrumentation as in JPaX.

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 [26]. 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 use $\mathcal{P}(G)$ to denote the set of pairs of vertices (u, v) such that there is a path from u to v in G. We assume that each vertex has a path to itself.

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. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [26] . 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.

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 3 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 $\{e_3, f_2\}$.

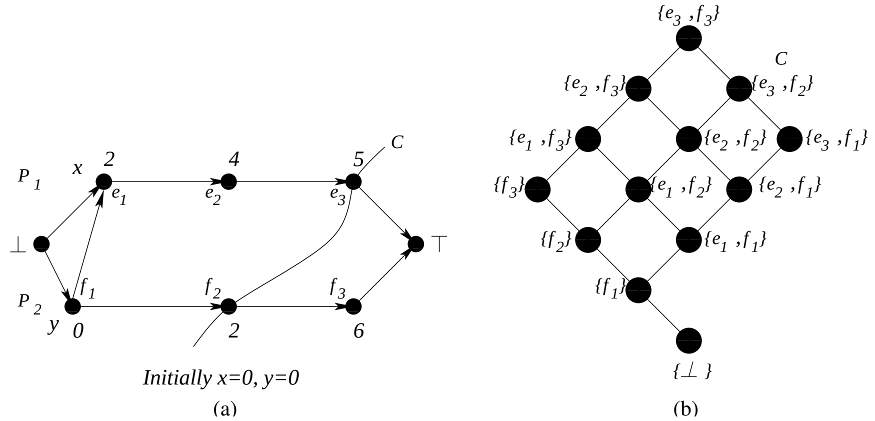

Figure 3: (a) A computation $\langle E, \rightarrow \rangle$ (b) and its lattice corresponding to $\mathcal{C}(G)$

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.

5 Background on Slicing

The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [6] which we describe next.

5.1 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 [6]. 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 [6]. 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 [6, 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 [15], 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 [6]. 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.

5.2 Computation Slice

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 [32]) *A* slice *of a computation with respect to a predicate is a directed graph with the least number 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 $\langle E, \rightarrow \rangle = \mathsf{slice}(\langle E, \rightarrow \rangle, \mathsf{true})$. It was proven in [32] 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. 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 [32]. 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.

6 Regular Predicates

Given a computation, the set of consistent cuts satisfying a regular predicate forms a sublattice of the set of consistent cuts of the computation [14]. Equivalently,

Definition 2 (regular predicate [32]) *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 **EF**, **AG**, and **EG**, otherwise it is a *temporal* regular predicate. In [14] polynomial-time algorithms are given for computing slices for non-temporal regular predicates. In [35], we showed that $\mathbf{EF}(p)$, $\mathbf{AG}(p)$, and $\mathbf{EG}(p)$ are temporal regular predicates when p is regular and gave polynomial-time algorithms to compute these slices, which we will briefly explain in the next section.

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 [31] 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.

6.1 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 $\lambda \colon \mathcal{C}(G) \rightarrow 2^{AP}$ 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 $L, \{\perp\} \models p$. 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.

7 Algorithms for Computing Slices for Temporal Predicates

Our distributed program analysis tool POTA uses computation slicing for predicate detection. Mittal and Garg [31] also used computation slicing for efficient detection of predicates of the form $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, $\mathbf{AG}(p)$ for *non-temporal* regular p. However, their predicate detection algorithm is based on computing slices for non-temporal regular predicates. Therefore, it cannot be used for detecting *nested* temporal predicates such as $\mathbf{AG}(p)$ when p is a temporal predicate like $p = \mathbf{EF}(q)$. In this section, we explain our slicing algorithms from [35] for *temporal* regular predicates to enable *efficient* predicate detection for RCTL+ which also includes nested temporal predicates.

The slice of a computation with respect to a *temporal* predicate is the smallest computation that contains all consistent cuts of the given computation for which the predicate holds. We proved in [35] that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular for regular p. Therefore, the slices for these temporal predicates are lean.

The input to each algorithm in this section is a computation $\langle E, \rightarrow \rangle$ and its slice with respect to a regular predicate p, that is, $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$. The output of each algorithm is an application of a temporal operator on the slice. For example, in order to generate a slice with respect to $\mathbf{AG}(\mathbf{EF}(p))$, where p is a non-temporal regular predicate, we can use the slicing algorithms explained in this section as follows: First, we compute the slice for p using the algorithms in [14, 31] for non-temporal regular predicates. Then, we give this slice and the computation to the $\mathbf{EF}$ slicing algorithm to obtain $\mathsf{slice}(\langle E, \rightarrow \rangle, \mathbf{EF}(p))$. Finally, the output of $\mathbf{EF}$ slicing algorithm and the computation is given as an input to $\mathbf{AG}$ slicing algorithm to obtain $\mathsf{slice}(\langle E, \rightarrow \rangle, \mathbf{AG}(\mathbf{EF}(p)))$.

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. For example, consider Figure 7(a) that displays the slice of the computation in Figure 3 with respect to $\neg((x = 5) \wedge (y = 2))$.

The only consistent cut in the computation that does not satisfy the predicate is $\{e_3, e_2, e_1, f_1, \bot\}$. By adding the edge (f_2, e_3), we disallow this consistent cut from the slice. Below, for computing slices for $\mathbf{EF}(p)$, we will show which edges we add to the computation. 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.

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 W is the final cut of the slice. To ensure that all cuts which cannot reach W do not belong to the slice, we add edges from $\top$ to the successors of events in the frontier of W. 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 3(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).

Now we describe the $\mathbf{AG}(p)$ slicing algorithm in Figure 5. We explained above that to obtain the slice for $\mathbf{AG}(p)$ we will add edges to the slice for p and eliminate consistent cuts that do not belong to slice for $\mathbf{AG}(p)$. Now we show which edges we should add. We claim that consistent cuts of the $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ that do not include vertex e of each additional edge (e, f) do not satisfy $\mathbf{AG}(p)$. For simplicity, let the $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ have a single additional edge (e, f). For example, consistent cuts $\{\bot\}$, $\{f_1, \bot\}$, $\{e_1, f_1, \bot\}$, and $\{e_2, e_1, f_1, \bot\}$ of the slice in Figure 7(a) 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 $\mathbf{AG}(p)$ and therefore we should add edges to eliminate them. We now give a proof sketch of the correctness of the algorithm for the simplified case with a single additional edge. The proof for full case can be found in [35].

Theorem 1 *Given a computation* $\langle E, \rightarrow \rangle$ *and* $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$*, 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* $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$.
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 $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ and every consistent cut of $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ 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

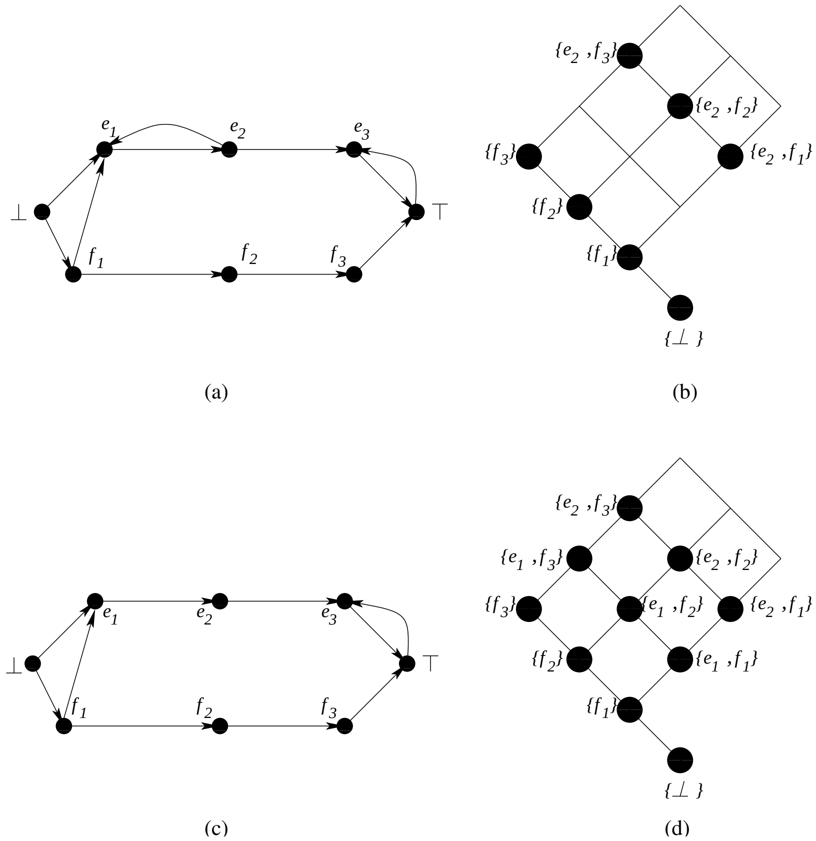

Figure 4: (a) A slice of $\langle E, \to \rangle$ in Fig. 3 (b) the corresponding sublattice (c) The application of the temporal operator $\mathbf{EF}$ on the slice in (a) (d) the corresponding sublattice

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. ∎

In Algorithm A2, for any additional edge (e, f), we add an edge from vertex e to vertex $\perp$. This ensures that consistent cuts of the computation that do not include vertex e of any additional edge (e, f) are disallowed from the slice, whereas the rest still belong to $\mathsf{slice}(\langle E, \to \rangle, \mathbf{AG}(p))$. For example, consistent cut $\{e_1, f_1, \perp\}$ of the slice in Figure 7(a) does not include vertex f_2 of the additional edge (f_2, e_3) in Figure 7(a), therefore we add an edge $(f_2, \perp)$ and obtain the slice in Figure 7(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.

The algorithm for $\mathbf{EG}(p)$ slicing displayed in Figure 6 is similar to the $\mathbf{AG}(p)$ slicing algorithm and will not be explained here. The complexity of the temporal

Algorithm A1

Input: A computation $\langle E, \rightarrow\rangle$ and slice$(\langle E, \rightarrow\rangle, p)$
Output: slice$(\langle E, \rightarrow\rangle, \mathbf{EF}(p))$
Step 1. Let G be $\langle E, \rightarrow\rangle$ and let W be the final cut of slice$(\langle E, \rightarrow\rangle, p)$
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 slice$(\langle E, \rightarrow\rangle, p)$
Output: slice$(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$
Step 1. Let G be slice$(\langle E, \rightarrow\rangle, p)$
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

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

Algorithm A3

Input: A computation $\langle E, \rightarrow\rangle$ and slice$(\langle E, \rightarrow\rangle, p)$
Output: slice$(\langle E, \rightarrow\rangle, \mathbf{EG}(p))$
Step 1. Let G be slice$(\langle E, \rightarrow\rangle, p)$
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 6: Algorithm for generating a slice with respect to $\mathbf{EG}(p)$

slicing algorithms is $O(n|E|)$ [35].

7.1 Complexity of RCTL Predicate Detection

Given a predicate in RCTL we can compute the slice for the predicate recursively from inside-out by applying the appropriate temporal or boolean operator on the slices. It is then easy to see if the predicate is satisfied by just checking whether the initial state of the computation and the slice are the same. The complexity of predicate detection is dominated by the complexity of computing the slice with respect to a non-temporal regular predicates, which has $O(n^2|E|)$ complexity [14, 31]. Therefore, the overall complexity of predicate detection

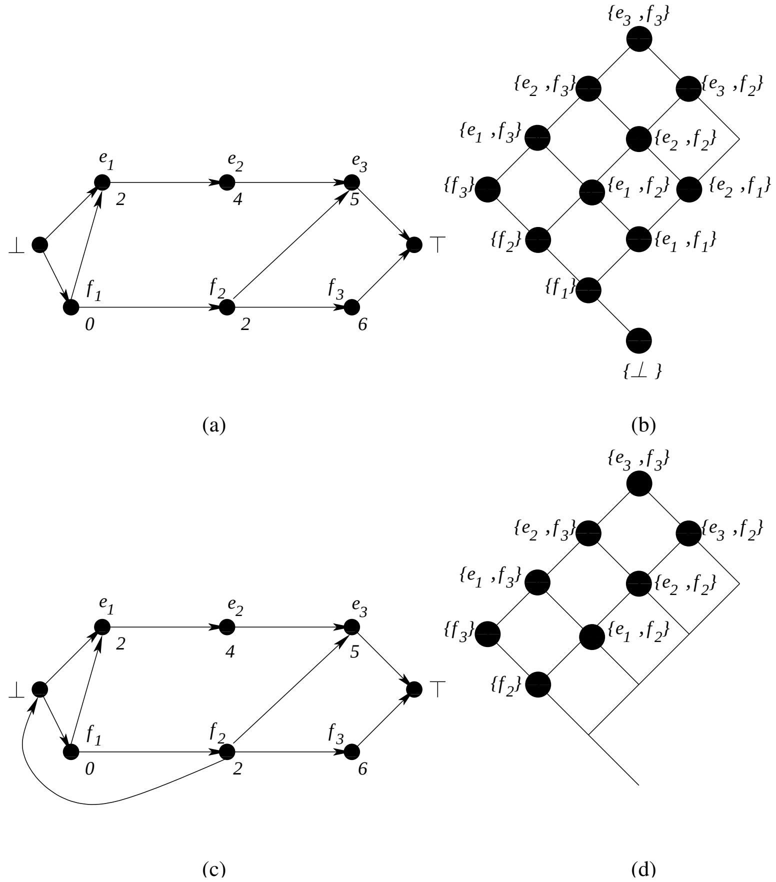

Figure 7: (a) The slice of $\langle E, \rightarrow \rangle$ in Fig. 3 with respect to $\neg((x = 5) \wedge (y = 0))$ (b) the corresponding sublattice (c) The slice of $\langle E, \rightarrow \rangle$ in Fig. 3 with respect to $\mathbf{AG}\ \neg((x = 5) \wedge (y = 2))$ (d) the corresponding sublattice

for RCTL is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p.

The predicate detection in RCTL+ has worst case exponential-time complexity. However, the slice is in general much smaller than the computation which we validate with experiments of the next section.

8 Experimental Results

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 [21]. 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 use the symbol $*$ to denote that the verification was not complemented due to running out of memory. Due to space limitations we present some of the experimental results in this section. The other results can be found in [34].

We consider the following message passing distributed programs *distributed dining philosophers*, *primary-secondary*, and *GIOP* protocols.

8.1 Distributed Dining Philosophers (ddph)

We use the Java protocol from [18] for this exercise and check the following properties. The complement of the safety property, that is, $\bigvee_{i,j\in 0\ldots(n-1)}\Big(\mathbf{EF}(eat_i \wedge eat_j)\Big)$ where i and j denote philosophers next to each other. The complement of the liveness property, that is, $\bigvee_{i\in 0\ldots(n-1)}\Big(\mathbf{EF}\big(hungry_i \wedge \mathbf{EG}(\neg eat_i)\big)\Big)$, for each philosopher i. Observe that the negation of a local predicate $\neg eat_i$ is also a local predicate and furthermore it is a regular predicate. Finally, we check the property $\mathbf{AG}(\mathbf{EF}(eat_i))$ which denotes that eating is possible from every state. Table 1 displays our results for the liveness property.

Table 1: Distributed Dining Philosophers, Liveness Property

		3	4	5	6	7	8	9	10	20	30	40	100	250
POTA	T	0.14	0.17	0.19	0.23	0.22	0.4	0.46	0.49	3.54	10.37	18.16	137.26	965.3
	M	0.18	0.29	0.36	0.42	0.5	0.64	0.78	0.92	1.57	4.53	6.84	33.45	96.01
SPIN	T	0.1	1.16	15.67	144.75	*								
	M	1.67	4.13	35.46	223.26	*								

8.2 Primary Secondary

The primary secondary program [39] 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. However, Table 2 shows that even in this case slicing can reduce the state space substantially.

Table 2: Primary Secondary, Safety Property

		3	4	5	6	7	8	9	10	20	30	40
POTA	T	0.14	0.21	0.28	0.6	0.97	1.03	1.34	2.46	7.88	33.29	133.37
	M	0.44	0.82	1.38	0.38	5.7	3.14	5.08	13.73	64.74	178.35	239.2
SPIN	T	0.01	0.01	0.02	0.07	0.38	0.45	1.96	*			
	M	1.57	1.57	1.67	2.29	5.56	6.48	23.48	*			

8.3 GIOP

In this section, we present experimental results for the General Inter-ORB Protocol (GIOP) which was verified in [23] using SPIN.

The Common Object Request Broker Architecture (CORBA) [17] describes the architecture of a middleware platform that supports the implementation of applications in distributed and heterogeneous environments. The CORBA standard is issued by OMG.

The ORB is the key component of the CORBA programming model. An ORB is responsible for transferring operations from Clients to Servers. This requires the ORB to locate a Server implementation (and possibly activate it), transmit the operation and its parameters, and finally return the results back to the Client.

The General Inter-ORB Protocol (GIOP) is the abstract protocol which is used for communications between CORBA ORBs. It specifies the transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport Protocol. GIOP is designed to be simple and easy to implement, while still allowing for reasonable scalability and performance. In order to allow server objects to move between different ORBs and have messages forwarded to them wherever they are, GIOP supports server migration.

Figure 8 displays the high level view of the Promela model of the GIOP system as depicted in [23]. The protocol consists of User, Client, Transport, Agent and Server processes. Here, we conduct experiments for 4 of the 8 LTL predicates used in [23] (properties (iv) and (v) are considered as one).

1. After sending a *URequest* message a User should eventually receive the corresponding *UReply* message.
   $\mathbf{EF}\Big(\mathit{URequestSent}_i \wedge \mathbf{EG}(\neg \mathit{UReplyReceived}_i)\Big)$, for all users i.

2. After sending an *SRequest* the GIOP-Agent should eventually receive a corresponding *SReply*.
   $\mathbf{EF}\Big(\mathit{SRequestSent}_i \wedge \mathbf{EG}(\neg \mathit{SReplyReceived}_i)\Big)$, for all agents i.

3. Requests sent by a client are responded to eventually by a reply unless they have been cancelled.
   $\mathbf{EF}\Big(\mathit{CRequestSent}_i \wedge \mathbf{EG}(\neg \mathit{CReplyReceived}_i \vee \neg \mathit{CCancelSent}_i)\Big)$, for all clients i.

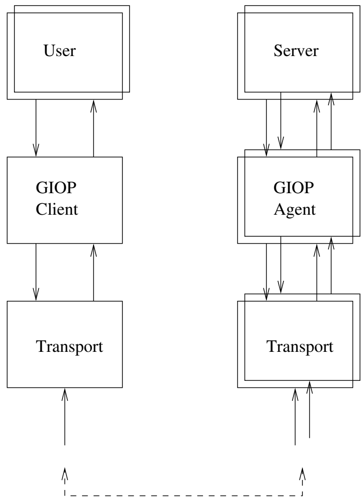

Figure 8: GIOP model

4. If the user received no exception, its request was performed exactly once. $\mathbf{AG}\Big(\neg NoException_i \vee \Big(\bigvee_k \bigwedge_j Server_j Processed_i = m\Big)\Big)$, where $m = 1$ if $k = j$ and $m = 0$ otherwise, for all users i and for all servers j, k.

5. If the user received exception, its request was performed at most once. $\mathbf{AG}\Big(\neg SystemException_i \vee \Big(\bigvee_k \bigwedge_j Server_j Processed_i = m\Big)$ $\vee \Big(\bigwedge_l Server_l Processed_i = 0\Big)\Big)$, where $m = 1$ if $k = j$ and $m = 0$ otherwise, for all users i and for all servers j, k, l.

The full verification (not approximate) of GIOP by Kamel and Leue [23] even for the configuration in Figure 8 with 10 processes was not completed due to state explosion. They could verify a simplified version of protocol without server migration with 10 processes. 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. SPIN displays a state coverage number (hash-factor) at the end of a verification with bit-state hashing. With bit-state hashing, they could verify the unsimplified version of protocol with 20 processes with 1.5 hash-factor, which means that the coverage was less than one percent since best coverage is obtained when the hash-factor is greater than 100.

We generated execution traces for a variety of GIOP architectures where we duplicated the User and Server blocks. In one case, we generated execution traces from unsimplified version of GIOP protocol where the total number of processes was increased to 250 and completed full verification of these traces. In Tables 3 and 4, we present our experimental results for the GIOP models with server migration.

Table 3: GIOP Property (iii)

		10	20	30	60	170	250
POTA	T	0.13	0.57	0.42	2.77	28.12	45.2
	M	0.5	1.71	3.19	26.62	92	256.3
SPIN	T	362.61	*				
	M	320.03	*				

Table 4: GIOP Property (v)

		10	20	30	60	170
POTA	T	0.12	5.34	19.06	134.68	174.33
	M	0.48	4.48	24.62	158.11	257.69
SPIN	T	319.21	*			
	M	305.49	*			

8.4 Discussion

In all execution trace verifications, SPIN could verify up to only 7 processes in ddph, and 10 in primary secondary and GIOP protocols, even when DCOLLAPSE and DMA compilation options were used. Observe that since the memory we use is larger than the one used in [23], the verification of the unsimplified GIOP with 10 processes is now possible in SPIN. We obtain more than three orders of magnitude speed up and state space reduction compared to partial order reduction with SPIN as shown in GIOP experiments. Using our slicing based technique we could verify upto 250 processes in some cases. We also injected faults into the traces and compared results. Even with bit-state hashing enabled verification in SPIN, the faults could not be found because the state spaces were too large and the coverage was low. However, the faults were easily found by POTA.

9 Conclusion

We have presented a partial order execution trace analysis tool (POTA) that implements our computation slicing algorithms. For problem sizes that preclude exhaustive program verification or exhaustive runtime verification, POTA proves to be an effective tool. Our technique is orthogonal to other reduction techniques, that is, one can always use POTA to reduce the state space as long as we can exploit the specification for computation slicing.

As a next step, we would like to apply our theory to multi-threaded Java programs. Furthermore, we are also interested in the research of selection of execution traces.

Acknowledgements: We would like to acknowledge Neeraj Mittal for his contribution in the implementation of our tool. We also thank Gerard J. Holzmann for discussion on SPIN.

References

[1] M. Brorkens and M. Moller. Dynamic event generation for runtime checking using the JDI. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

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

[3] 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.

[4] 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.

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

[7] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN Model Checking and Verification*, volume 1885 of *LNCS*, pages 323–330, 2000.

[8] 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.

[9] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994. Also appeared in *Proc. TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 613–628. Springer-Verlag, 1993.

[10] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[11] B. Finkbeiner and H. B. Sipma. Checking finite traces using alternating automata. In Klaus Havelund and Grigore Rosu, editors, *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 44–60. Elsevier, July 2001.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] Object Management Group. The Common Object Request Broker: Architecture and Specification. August 1997.

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

[19] K. Havelund and G. Rosu. Monitoring java programs with Java PathExplorer. In K. Havelund and G. Rosu, editors, *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[20] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of Int. Conference on Automated Software Engineering (ASE'01)*, pages 135–143, San Diego, California, 2001.

[21] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

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

[23] M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (giop) using promela and spin. *Software Tools for Technology Transfer*, 2(4):394–409, April 2000.

[24] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. In K. Havelund and G. Rosu, editors, *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[25] 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.

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

[27] S. Leue and P.B. Ladkin. Implementing and verifying msc specifications using promela/xspin. In *Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*, volume 32 of *DIMACS Series*, 1997.

[28] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

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

[30] L. I. Millett and T. Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. *Software Tools for Technology Transfer*, 2(4):343–349, April 2000.

[31] 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.

[32] 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.

[33] D. Peled. All from one, one for all: On model checking using representatives. In C. Courcoubetis, editor, *Computer Aided Verification: Proc. of the 5th International Conference CAV'93*, pages 409–423. Springer, Berlin, Heidelberg, 1993.

[34] A. Sen. http://maple.ece.utexas.edu/˜sen/.

[35] A. Sen and V. K. Garg. Automatic generation of slices for temporal logic predicate detection. Technical Report TR-PDS-2002-001, PDSL, ECE Dept. Univ. of Texas at Austin, 2002. Available at http://maple.ece.utexas.edu/.

[36] 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.

[37] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, Univ. of Illinois at Urbana Champaign, April 2003.

[38] 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.

[39] 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.

[40] A. Valmari. A stubborn attack on state explosion. In *Proc. of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 156–165, Berlin, Germany, 1991.

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

10 Appendix

10.1 Illustration of Various Concepts

10.2 Illustration of Concepts in Section 4

Example 1 *Consider the computation depicted in Figure 9(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* $g_4 \to e_4$ *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.*

10.3 Illustration of the Birkhoff's Theorem

Example 2 *Consider the computation shown in Figure 9(a). The (distributive) lattice spanned by its set of consistent cuts is shown in Figure 9(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 9(b)) can be expressed as the join of the consistent cuts* U *and* V.

10.4 Illustration of the Notion of Computation Slice

Example 3 *Consider the lattice of consistent cuts depicted in Figure 9(b). The consistent cuts that satisfy the predicate* $x + y - z \leq 1$ *have been shaded in the figure. Figure 9(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 9(d) portrays the partial order induced on 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 9(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.*

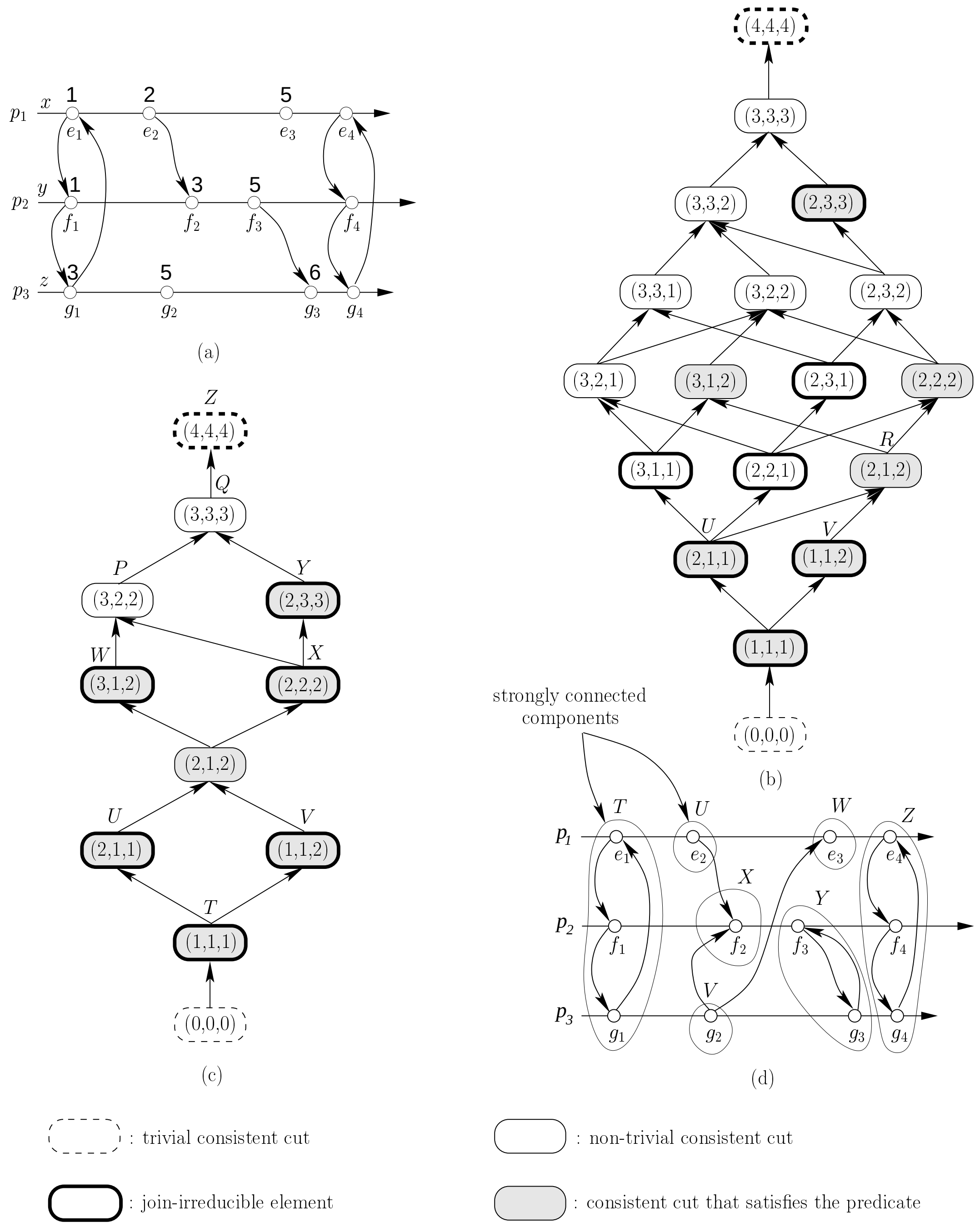

Figure 9: (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leq 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.