# Computation Slicing: Techniques and Theory

*Neeraj Mittal and Vijay K. Garg*

**TR-PDS-2001-004 November 2001**

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

# Computation Slicing: Techniques and Theory

Neeraj Mittal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
neerajm@cs.utexas.edu

Vijay K. Garg*
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

**Abstract**

*We generalize the notion of slice introduced in our earlier paper [6]. A slice of a distributed computation with respect to a global predicate is the smallest computation that contains all consistent cuts of the original computation that satisfy the predicate. We prove that slice exists for all global predicates. We also establish that it is, in general, NP-complete to compute the slice. Optimal algorithms to compute slices for special cases of predicates are provided. Furthermore, we present an efficient algorithm to graft two slices, that is, given two slices, either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. We give application of slicing in general and grafting in particular to global property evaluation of distributed programs. Finally, we show that the results pertaining to consistent global checkpoints [13, 17] can be derived as special cases of computation slicing.*

## 1 Introduction

Writing distributed programs is an error prone activity; it is hard to reason about them because they suffer from the combinatorial explosion problem. Testing and debugging, and

*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.

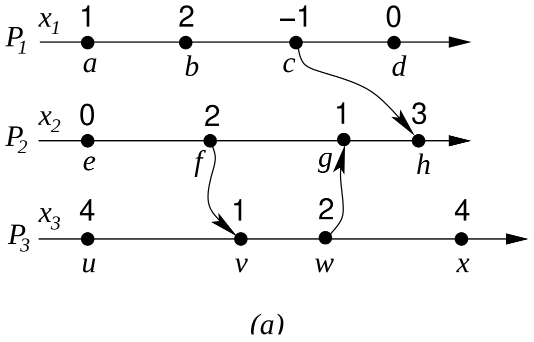


Figure 1: (a) A computation and (b) its slice with respect to $(x_1 \geq 1) \wedge (x_3 \leq 3)$.

fault tolerance are two important ways to ensure the reliability of distributed systems. Thus it becomes necessary to develop techniques that facilitate the analysis of distributed computations. For example, monitoring the execution of a distributed system, and, on detecting a fault, initiating the appropriate corrective action is an important way to tolerate software bugs. Various abstractions such as predicate detection (e.g., [1, 3, 7]) and predicate control [15, 16, 11] have been defined to carry out such analysis.

In our earlier paper [6], we propose another abstraction, called *computation slice*, which was defined as: a slice of a distributed computation with respect to a global predicate is another computation that contains *those and only those* consistent cuts (or snapshots) of the original computation that satisfy the predicate. In [6], we also introduce a class of global predicates called *regular predicates*: a global predicate is regular if and only if whenever two consistent cuts satisfy the predicate then the cuts given by their set intersection and set union also satisfy the predicate. We show that slice exists only for regular predicates and present an efficient algorithm to compute the slice. The class of regular predicates is closed under conjunction.

A limitation of the definition of slice in [6] is that slice exists only for a specific class of predicates. This prompted us to weaken the definition of slice to the *smallest* computation that contains all consistent cuts of the original computation that satisfy the predicate. In this paper, we show that slice exists for all global predicates under the new definition.

The notion of computation slice is analogous to the concept of program slice [18]. Given a program and 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. A slice could be static [18] or dynamic (for a specific program input) [9]. The notion of a slice has been also extended to distributed programs [8]. Program slicing has been shown to be useful in program debugging, testing, program understanding and software maintenance [9, 18]. A slice can significantly narrow the size of the program to be analyzed, thereby making the understanding of the program behaviour easier. We expect to reap the same benefit from a computation slice.

Computation slicing is also useful for reducing search space for GNP-complete problems such as predicate detection [3, 7, 14, 12]. Given a distributed computation and a global predicate, predicate detection requires finding a consistent cut of the computation, if it exists, that satisfies the predicate. It is a fundamental problem in distributed systems and arises in contexts such as testing and debugging, monitoring and fault tolerance of distributed programs.

As an illustration, suppose we want to detect the predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geq 1) \wedge (x_3 \leq 3)$ in the computation shown in Figure 1(a). The computation consists of three processes $P_1$, $P_2$ and $P_3$ hosting integer variables $x_1$, $x_2$ and $x_3$, respectively. The events are represented by solid circles. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of variable $x_1$ immediately after executing the event $c$ is $-1$. The first event on each process initializes the state of the process and every consistent cut contains these initial events. Without computation slicing, we are forced to examine all consistent cuts of the computation, twenty eight in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute a slice of the computation with respect to the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3)$ as portrayed in Figure 1(b). The slice is modeled by a directed graph. Each vertex of the graph corresponds to a subset of events. If a vertex is contained in a consistent cut, the interpretation is that all events corresponding to the vertex are contained in the cut. Moreover, a vertex belongs to a consistent cut only if all its incoming neighbours are also present in the cut. We can now restrict our search to the consistent cuts of the slice which are only six in number, namely $\{a, e, f, u, v\}$, $\{a, e, f, u, v, b\}$, $\{a, e, f, u, v, w\}$, $\{a, e, f, u, v, b, w\}$, $\{a, e, f, u, v, w, g\}$ and $\{a, e, f, u, v, b, w, g\}$. The slice has much fewer consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

As another example, suppose we want to detect the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ in the computation shown in Figure 2(a). Each $x_i$ is a boolean variable. An event for which the respective variable is true has been encircled. We prove elsewhere [12] that it is, in general, NP-complete to detect such predicates. To reduce the search space, we compute slices of the computation with respect to clauses $x_1 \vee x_2$ and $x_3 \vee x_4$ as shown in Figure 2(b) and Figure 2(c), respectively. Next, we graft the two slices together to obtain a slice that contains consistent cuts common to both of them as shown in Figure 2(d). In this case, the computation has as many as seven consistent cuts whereas the slice contains only a single consistent cut, namely $\{a, b, c, d, e, g\}$.

We also show that the results pertaining to consistent global checkpoints [13, 17] can be derived as special cases of computation slicing. In particular, we furnish an alternate characterization of the condition under which individual local checkpoints can be combined with others to form a consistent global checkpoint (consistency theorem by Netzer and Xu [13]): a set of local checkpoints can belong to the same consistent global snapshot if and only if the local checkpoints in the set are mutually consistent (including with itself) in the slice.

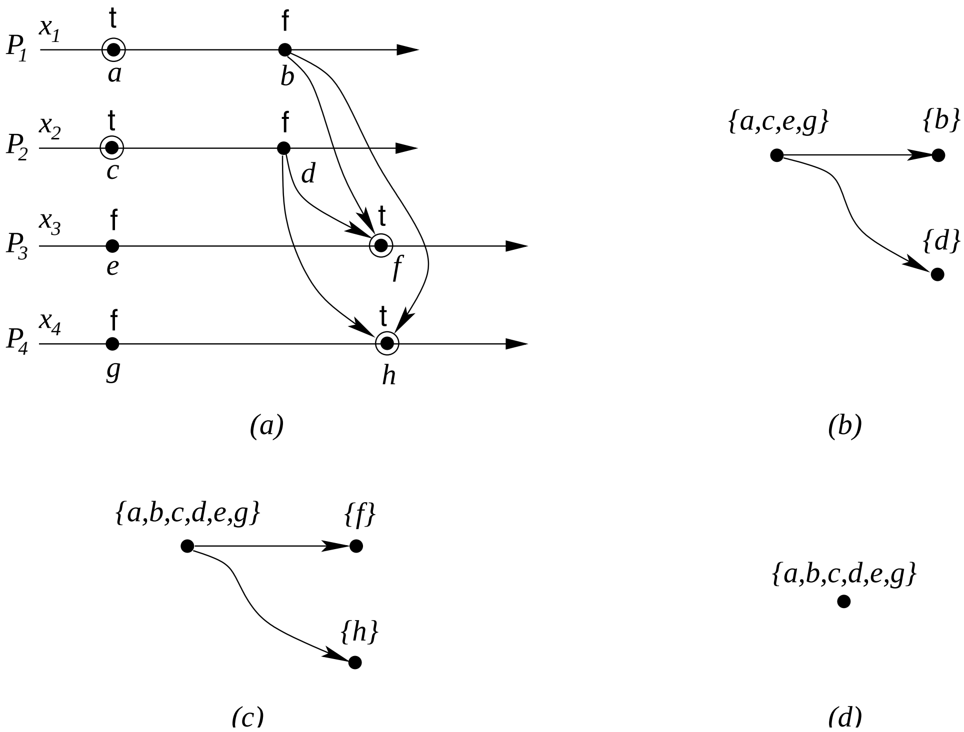


Figure 2: (a) A computation, (b) its slice with respect to $x_1 \vee x_2$, (c) its slice with respect to $x_3 \vee x_4$ and (d) graft of the two slices.

Moreover, the R-graph (rollback-dependency graph) defined by Wang [17] is a special case of the slice. The minimum and maximum consistent global checkpoints that contain a set of local checkpoints [17] can also be easily obtained using the slice.

In summary, this paper makes the following contributions:

- We generalize the notion of computation slice introduced in our earlier paper [6] and demonstrate that it exists for all global predicates.

- We establish that it is, in general, NP-complete to ascertain whether a global predicate has a non-empty slice.

- We given an application of slicing to monitoring global properties in distributed systems. Specifically, we give an algorithm to determine whether a global predicate satisfying certain properties is possibly true, invariant or controllable in a distributed computation using slicing.

- We develop an efficient algorithm to *graft* two slices. Grafting can be done in two ways. Given two slices, we can either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. Using grafting, we devise a polynomial-time algorithm to compute slices for complement of regular predicates, called *co-regular predicates*. We also show how grafting can be utilized to avoid examining many consistent cuts when detecting a predicate [12].

- We provide optimal algorithms to compute slices for special cases of regular predicates. The algorithm to compute slices (for regular predicates) in our earlier paper [6] has $O(N^2|E|)$ time-complexity where $N$ is the number of processes and $E$ is the set of events in the distributed system. The algorithms presented in this paper have $O(|E|)$ time-complexity which is optimal.

- Finally, we show that the results pertaining to consistent global checkpoints [13, 17] can be derived as special cases of computation slicing.

The paper is organized as follows. We present our model of distributed system in Section 2. In Section 3 we define computation slice formally. We prove that slice exists for all predicates in Section 4. We also establish that it is, in general, NP-complete to compute slice for a predicate. Furthermore, we give an application of slicing to monitoring global properties in distributed systems. In Section 5 we present an efficient representation of slice which is later used in Section 6 to develop an efficient algorithm to graft two slices. Optimal algorithms to compute slices for special cases of regular predicates are discussed in Section 7. We also demonstrate that many results pertaining to consistent global checkpoints can be derived as special cases of computation slicing. Finally, we present our conclusion in Section 8.

# 2 Model and Notation

## 2.1 Distributive Lattices

Given a lattice, its meet (infimum) and join (supremum) operators are denoted by $\sqcap$ and $\sqcup$, respectively. A lattice is *distributive* if and only if meet distributes over join [5]. Formally,

$$a \sqcap (b \sqcup c) \equiv (a \sqcap b) \sqcup (a \sqcap c)$$

It can be proved that meet distributes over join if and only if join distributes over meet.

## 2.2 Directed Graphs: Path- and Cut-Equivalence

Traditionally, a distributed computation is modeled as a partial order on a set of events [10]. In this paper we relax the restriction that the order on events must be a partial order, and use directed graphs to model distributed computations and 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 its set of vertices and edges, respectively. A subset of vertices of a directed graph forms a *consistent cut* if and only if the subset contains a vertex only if it also contains all its incoming neighbours. Formally,

$$C \text{ is a consistent cut of } G \;\triangleq\; \langle \forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) : f \in C \;\Rightarrow\; e \in C \rangle$$

Observe that a consistent cut either contains all vertices in a cycle or none of them. This observation can be generalized to a strongly connected component. Traditionally, the notion of consistent cut (*down-set* or *order ideal*) is defined for partially ordered sets [5]. Here, we extend the notion to sets with arbitrary orders. Let $\mathcal{C}(G)$ denote the set of consistent cuts of a directed graph $G$. 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.

**Theorem 1** *Given a directed graph $G$, $\langle \mathcal{C}(G); \subseteq \rangle$ forms a distributive lattice.*

*Proof:* Let $C_1$ and $C_2$ be consistent cuts of $G$. We define their meet and join as follows:

$$\begin{aligned} C_1 \sqcap C_2 &\;\triangleq\; C_1 \cap C_2 \\ C_1 \sqcup C_2 &\;\triangleq\; C_1 \cup C_2 \end{aligned}$$

It is sufficient to establish that $C_1 \cap C_2$ and $C_1 \cup C_2$ are consistent cuts of $G$ which can be easily verified. ∎

Again, the above theorem is a generalization of the result in lattice theory that the set of down-sets of a partially ordered set forms a distributive lattice [5].

A directed graph $G$ is *cut-equivalent* to a directed graph $H$, denoted by $G \overset{C}{\cong} H$, if and only if they have the same set of consistent cuts. Formally,

$$G \overset{C}{\cong} H \;\triangleq\; \mathcal{C}(G) = \mathcal{C}(H)$$

Let $\mathcal{P}(G)$ 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. A directed graph $G$ is *path-equivalent* to a directed graph $H$, denoted by $G \overset{P}{\cong} H$, if and only if a path from vertex $u$ to vertex $v$ in $G$ implies a path from vertex $u$ to vertex $v$ in $H$ and vice versa. Formally,

$$G \overset{P}{\cong} H \;\triangleq\; \mathcal{P}(G) = \mathcal{P}(H)$$

The next lemma explores the relation between the two notions.

**Lemma 2** *Let $G$ and $H$ be directed graphs with the same set of vertices. Then,*

$$\mathcal{P}(G) \subseteq \mathcal{P}(H) \;\equiv\; \mathcal{C}(G) \supseteq \mathcal{C}(H)$$

Lemma 2 implies that two directed graphs are cut-equivalent if and only if they are path-equivalent. Thus, in other words, in order to determine whether two directed graphs are cut-equivalent, it is *necessary and sufficient* to ascertain that they are path-equivalent. This is significant because path-equivalence can be verified in polynomial-time ($|\mathcal{P}(G)| = O(|\mathsf{V}(G)|^2)$) as compared to cut-equivalence which is computationally expensive to ascertain in general ($|\mathcal{C}(G)| = O(2^{|\mathsf{V}(G)|})$). In the rest of the chapter we use $\cong$ to denote both $\overset{C}{\cong}$ and $\overset{P}{\cong}$.

## 2.3 Distributed Computations as Directed Graphs

We assume an asynchronous distributed system with the set of processes $P = \{P_1, P_2, \ldots, P_N\}$. Processes communicate and synchronize with each other by sending messages over a set of reliable channels.

A *local computation* of a process is described by a sequence of events that transforms the *initial state* of the process into the *final state*. At each step, the *local state* of a process is captured by the initial state and the sequence of events that have been executed up to that step. Each event is a *send event*, a *receive event*, or an *internal event*. An event causes the local state of a process to be updated. Additionally, a send event causes a message to be sent and a receive event causes a message to be received. We assume the presence of fictitious *initial* and *final events* on each process $P_i$, denoted by $\bot_i$ and $\top_i$, respectively. The initial event occurs before any other event on the process and initializes the state of the process. The final events occurs after all other events on the process.

Let $proc(e)$ denote the process on which event $e$ occurs. The predecessor and successor events of $e$ on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist. We denote the order of events on process $P_i$ by $\leadsto_{P_i}$. Let $\leadsto_P$ be the union of all $\leadsto_{P_i}$s, $1 \leq i \leq N$, and $\underline{\leadsto}_P$ denote the reflexive closure of $\leadsto_P$.

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 $\rightarrow$. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that, for any computation $\langle E, \rightarrow \rangle$, $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [10]. We assume that all initial events belong to the same strongly connected component. Similarly, all final events belong to the same strongly connected component. This ensures that any non-trivial consistent cut will contain all initial events and none of the final events. As a result, every consistent cut of a computation in traditional model is a non-trivial consistent cut of the computation in our model and vice versa. Only non-trivial consistent cuts are of real interest to us. We will see later that our model allows us to capture empty slices in a very convenient fashion.

A distributed computation in our model can contain cycles. This is because whereas a

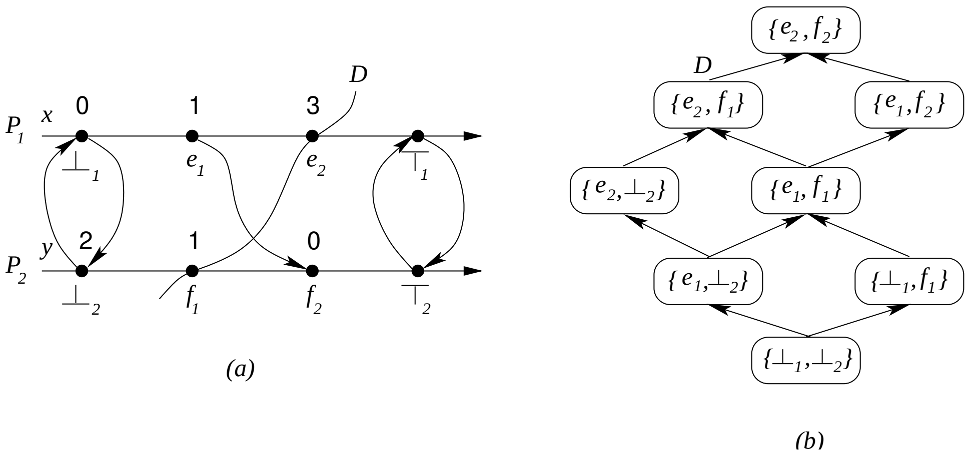


Figure 3: (a) A computation and (b) the lattice corresponding to its consistent cuts.

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.

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,

$$C.frontier \triangleq \{e \in C \mid succ(e) \text{ exists } \Rightarrow succ(e) \notin C\}$$

A consistent cut is uniquely characterized by its frontier and vice versa. Thus sometimes, especially in figures, we specify a consistent cut by simply listing the events in its frontier instead of enumerating all its events. Two events are said to be *consistent* if and only if they are contained in the frontier of some consistent cut, otherwise they are *inconsistent*. It can be verified that events $e$ and $f$ are consistent if and only if there is no path in the computation from $succ(e)$, if it exists, to $f$ and from $succ(f)$, if it exists, to $e$. Also, note that, in our model, an event can be inconsistent with itself. Figure 3 depicts a computation and the lattice of its (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $D$ is represented by $\{e_2, f_1\}$.

## 2.4 Global Predicates

A *global predicate* (or simply a *predicate*) is a boolean-valued function defined on variables of processes. It is evaluated on events in the frontier of a consistent cut. Some examples are mutual exclusion and "at least one philosopher does not have any fork". We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* if and only if it depends on variables of at most one process. For example, "$P_i$ is in red state" and "$P_i$ does not have the token".

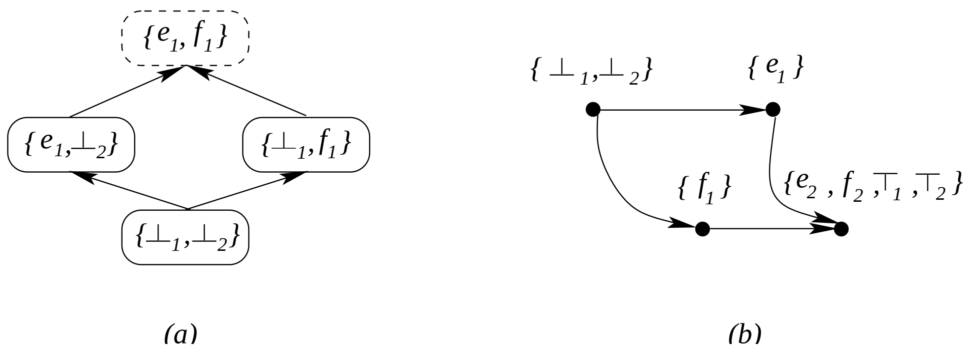


Figure 4: (a) The sublattice of the lattice in Figure 3(b) with respect to the predicate $((x < 2) \wedge (y > 1)) \bigvee (x < 1)$, and (b) the corresponding slice.

# 3 Slicing a Distributed Computation

In this section, we define the notion of slice of a computation with respect to a predicate. The definition given here is weaker than the definition given in our earlier paper [6]. However, slice now exists with respect to every predicate (not just specific predicates).

**Definition 1 (Slice)** *A* slice *of a computation with respect to a predicate is the smallest directed graph—with minimum number of consistent cuts—that contains all consistent cuts of the original computation that satisfy the predicate.*

We will later show that the smallest computation is well-defined for every predicate. A slice of computation $\langle E, \rightarrow \rangle$ with respect to a predicate $b$ is denoted by $\langle E, \rightarrow \rangle_b$. Note that $\langle E, \rightarrow \rangle = \langle E, \rightarrow \rangle_{\mathsf{true}}$. In the rest of the paper, we use the terms "computation" and "slice" interchangeably.

Figure 4(a) depicts the set of consistent cuts of the computation in Figure 3(a) that satisfy the predicate $((x < 2) \wedge (y > 1)) \bigvee (x < 1)$. The cut shown with dashed outline does not actually satisfy the predicate but has to be included to complete the sublattice. Figure 4(b) depicts the slice of the computation with respect to the predicate. In the figure, all events in a subset belong to the same strongly connected component.

In our model, every slice derived from the computation $\langle E, \rightarrow \rangle$ will have the trivial consistent cuts ($\emptyset$ and $E$) among its set of consistent cuts. Consequently, a slice is *empty* if and only if it has no non-trivial consistent cuts. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut.

A slice of a computation with respect to a predicate is *lean* if and only if every consistent cut of the slice satisfies the predicate.

## 4 Regular Predicates

A global predicate is *regular* if and only if the set of consistent cuts that satisfy the predicate forms a *sublattice* of the lattice of consistent cuts [6]. Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union will also satisfy the predicate. Some examples of regular predicates are any local predicate and channel predicates such as 'there are at most $k$ messages in transit from $P_i$ to $P_j$". The class of regular predicates is closed under conjunction [6]. We prove elsewhere [6] that the slice of a computation with respect to a predicate is lean if and only if the predicate is regular. We next show how slicing can be used to monitor predicates in distributed systems. Later, we use the notion of regular predicates to prove that the slice exists and is well-defined with respect to every predicate.

### 4.1 Using Slices to Monitor Regular Predicates

A predicate can be monitored under four modalities, namely *possibly*, *definitely*, *invariant* and *controllable* [3, 7, 16, 11]. A predicate is *possibly* true in a computation if and only if there is a consistent cut of the computation that satisfies the predicate. On the other hand, a predicate *definitely* holds in a computation if and if it eventually becomes true in all runs of the computation (a *run* is a path in the lattice of consistent cuts). The predicates *invariant*: $b$ and *controllable*: $b$ are duals of the predicates *possibly*: $b$ and *definitely*: $b$, respectively. Predicate detection normally involves detecting a predicate under *possibly* modality whereas predicate control involves monitoring a predicate under *controllable* modality. Monitoring has applications in the areas of testing and debugging and software fault tolerance of distributed programs.

The next theorem describes how *possibly*:$b$, *invariant*:$b$ and *controllable*:$b$ can be computed using the notion of slice when $b$ is a regular predicate. We do not yet know the complexity of computing *definitely*: $b$ when $b$ is regular. Given a directed graph $G$, let $scc(G)$ denote the number of strongly connected components of $G$.

**Theorem 3** *A regular predicate is*

*(1)* **possibly** *true in a computation if and only if the slice of the computation with respect to the predicate has at least one non-trivial consistent cut, that is, it has at least two strongly connected components. Formally,*

$$possibly: b \;\equiv\; scc(\langle E, \rightarrow \rangle_b) \geq 2$$

*(2)* **invariant** *in a computation if and only if the slice of the computation with respect to the predicate is cut-equivalent to the computation. Formally,*

$$invariant: b \;\equiv\; \langle E, \rightarrow \rangle_b \cong \langle E, \rightarrow \rangle$$

*(3)* **controllable** *in a computation if and only if the slice of the computation with respect to the predicate has the same number of strongly connected components as the computation. Formally,*

$$controllable : b \;\equiv\; scc(\langle E, \rightarrow\rangle_b) = scc(\langle E, \rightarrow\rangle)$$

*Proof:* The first two propositions are obvious. As for the last proposition, it can be verified that a regular predicate is controllable in a computation if and only if there exists a path from the initial to the final consistent cut in the lattice, generated by the set of consistent cuts of the computation, such that every consistent cut along the path satisfies the predicate [15]. This happens when and only when the length of the longest chain in the sublattice, spanned by the set of consistent cuts that satisfy the predicate, is equal to the length of the longest chain in the lattice, corresponding to the set of all consistent cuts. For a finite distributive lattice, the length of its longest chain is exactly equal to the number of its join-irreducible elements [5]. Moreover, for a directed graph, the number of join-irreducible elements of the lattice generated by its set of consistent cuts—both trivial and non-trivial—is same as the number of its strongly connected components. This is because, from Birkhoff's representation theorem for finite distributive lattices [5], the component graph [4] corresponding to a directed graph (which is the graph with vertices as the strongly connected components of the given directed graph) is isomorphic to the set of join-irreducible elements of the lattice generated by the consistent cuts of the directed graph. ■

Observe that the first proposition holds for any arbitrary predicate. Since detecting whether a predicate possibly holds in a computation is NP-complete in general [2, 14, 12], it is, in general, NP-complete to determine whether a predicate has a non-empty slice.

### 4.2 Regularizing a Non-Regular Predicate

In this section, we show that slice exists and is well-defined with respect to every predicate. We know that it is true for at least regular predicates [6]. In addition, the slice with respect to a regular predicate is lean. We exploit these facts and define a closure operator, denoted by $reg$, which, given a computation, converts an arbitrary predicate into a regular predicate satisfying certain properties. Given a computation $\langle E, \rightarrow\rangle$, let $\mathcal{R}(E)$ denote the set of predicates that are regular with respect to the computation ($\rightarrow$ is implicit).

**Definition 2 (reg)** *Given a predicate $b$, we define $reg\,(b)$ as the predicate that satisfies the* *following conditions:*

*(1) it is regular, that is,* $reg\,(b) \in \mathcal{R}(E)$*,*

*(2) it is weaker than* $b$*, that is,* $b \Rightarrow reg\,(b)$*, and*

*(3) it is stronger than any other predicate that satisfies (1) and (2), that is,*
$\langle \forall\, b' \,:\, b' \in \mathcal{R}(E) :\; (b \Rightarrow b') \Rightarrow (reg\,(b) \Rightarrow b') \rangle$

Informally, $reg\,(b)$ is the *strongest regular predicate weaker* than $b$. In general, $reg\,(b)$ not only depends on the predicate $b$ but also on the computation under consideration. We assume the dependence on computation to be implicit and make it explicit only when necessary. The next theorem establishes that $reg\,(b)$ exists for every predicate $b$. Observe that the slice for $b$ is given by the slice for $reg\,(b)$. Thus slice exists and is well-defined for all predicates.

**Theorem 4** *Given a predicate* $b$*,* $reg\,(b)$ *exists and is well-defined.*

*Proof:* Let $\mathcal{R}_b(E)$ be the set of regular predicates in $\mathcal{R}(E)$ weaker than $b$. Observe that $\mathcal{R}_b(E)$ is non-empty because true is a regular predicate weaker than $b$ and therefore contained in $\mathcal{R}_b(E)$. We set $reg\,(b)$ to the conjunction of all predicates in $\mathcal{R}_b(E)$. Formally,

$$reg\,(b) \;\triangleq\; \bigwedge_{q \in \mathcal{R}_b(E)} q$$

It remains to be shown that $reg\,(b)$ as defined indeed satisfies the three required conditions. Condition (1) holds since the class of regular predicates is closed under conjunction. Condition (2) holds because every predicate in $\mathcal{R}_b(E)$ is weaker than $b$ and hence their conjunction is weaker than $b$. Finally, let $b'$ be a predicate that satisfies conditions (1) and (2). Note that $b' \in \mathcal{R}_b(E)$. Since conjunction of predicates is stronger than any of its conjunct, $reg\,(b)$ is stronger than $b'$. Thus $reg\,(b)$ satisfies condition (3). ■

Thus, given a computation $\langle E, \rightarrow \rangle$ and a predicate $b$, the slice of $\langle E, \rightarrow \rangle$ with respect to $b$ can be obtained by first applying $reg$ operator to $b$ to get $reg\,(b)$ and then computing the slice of $\langle E, \rightarrow \rangle$ with respect to $reg\,(b)$. Equivalently,

$$\langle E, \rightarrow \rangle_b = \langle E, \rightarrow \rangle_{reg\,(b)}$$

**Theorem 5** *reg is a closure operator. Formally,*

*(1)* $reg\,(b)$ *is weaker than* $b$*, that is,* $b \;\Rightarrow\; reg\,(b)$*,*

*(2) reg is monotonic, that is,* $(b \Rightarrow b') \;\Rightarrow\; (reg\,(b) \Rightarrow reg\,(b'))$*, and*

*(3) reg is idempotent, that is,* $reg\,(reg\,(b)) \;\equiv\; reg\,(b)$.

*Proof:* (*reg* $(b)$ *is weaker than* $b$) Follows from the definition.

(*reg is monotonic*) Since $reg\,(b')$ is weaker than $b'$, it is weaker than $b$. More precisely, $reg\,(b')$ is a regular predicate weaker than $b$. By definition, $reg\,(b)$ is the strongest regular predicate weaker than $b$. Therefore $reg\,(b)$ is stronger than $reg\,(b')$ or, in other words, $reg\,(b) \Rightarrow reg\,(b')$.

(*reg is idempotent*) Follows from the fact that $reg\,(b)$ is a regular predicate and is weaker than $reg\,(b)$. ∎

From the above theorem it follows that [5, Theorem 2.21],

**Corollary 6** $\langle \mathcal{R}; \Rightarrow \rangle$ *forms a lattice.*

The meet and join of two regular predicates $b_1$ and $b_2$ is given by

$$
\begin{aligned}
b_1 \sqcap b_2 &\;\triangleq\; b_1 \wedge b_2 \\
b_1 \sqcup b_2 &\;\triangleq\; reg\,(b_1 \vee b_2)
\end{aligned}
$$

The dual notion of $reg\,(b)$, the weakest regular predicate stronger than $b$, is conceivable. However, the following example proves that such a predicate may not be well-defined.

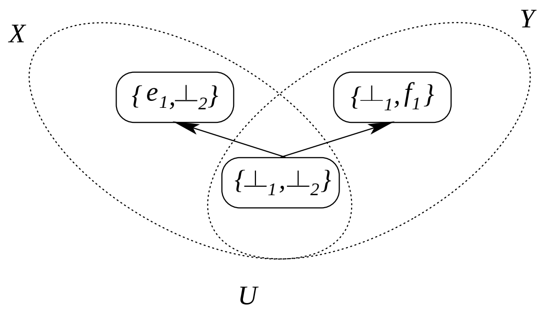


Figure 5: An example to illustrate that dual of $reg\,(b)$ is not well-defined.

Figure 5 shows the set of consistent cuts $U$ of the computation $\langle E, \rightarrow \rangle$ in Figure 3 that satisfy $((x < 2) \wedge (y > 1)) \bigvee (x < 1)$. Two different subsets $X$ and $Y$ of $U$, where $|X| = |Y|$, form maximal sublattices of $\mathcal{C}(\langle E, \rightarrow \rangle)$.

## 5 Representing a Slice

Observe that any directed graph that is cut-equivalent or path-equivalent to a slice constitutes its valid representation. However, for computational purposes, it is preferable to select those

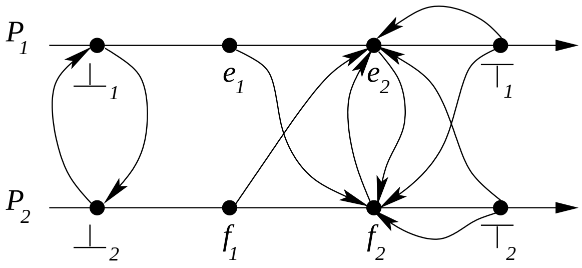


Figure 6: The skeletal representation of the slice in Figure 4(b).

graphs to represent a slice that have fewer edges and can be constructed cheaply. In this section, we show that every slice can be represented by a directed graph with $O(|E|)$ vertices and $O(N|E|)$ edges. Furthermore, the graph can be built in $O(N^2|E|)$ time.

Given a computation $\langle E, \rightarrow \rangle$, a regular predicate $b$ and an event $e$, let $J_b(e)$ denote the least consistent cut of $\langle E, \rightarrow \rangle$ that contains $e$ and satisfies $b$. In case no consistent cut containing $e$ that also satisfies $b$ exists, $J_b(e)$ is set to the trivial consistent cut $E$. Here, we use $E$ as a *sentinel* cut. Figure 6 depicts a directed graph that represents the slice shown in Figure 4(b). In the figure, $J_b(e_1) = \{\perp_1, e_1, \perp_2\}$ and $J_b(f_2) = \{\perp_1, e_1, e_2, \top_1, \perp_2, f_1, f_2, \top_2\}$.

The cut $J_b(e)$ can also be viewed as the least consistent cut of the slice $\langle E, \rightarrow \rangle_b$ that contains the event $e$. The results in [6] establish that it is sufficient to know $J_b(e)$ for each event $e$ in order to recover the slice. In particular, a directed graph with $E$ as the set of vertices and an edge from an event $e$ to an event $f$ if and only if $J_b(e) \subseteq J_b(f)$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. We also present an $O(N^2|E|)$ algorithm to compute $J_b(e)$ for each event $e$. However, the graph so obtained can have as many as $\Omega(|E|^2)$ edges.

Let $F_b(e, i)$ denote the earliest event $f$ on $P_i$ such that $J_b(e) \subseteq J_b(f)$. Informally, $F_b(e, i)$ is the earliest event on $P_i$ that is reachable from $e$ in the slice $\langle E, \rightarrow \rangle_b$. For example, in Figure 6, $F_b(e_1, 1) = e_1$ and $F_b(e_1, 2) = f_2$. Given $J_b(e)$ for each event $e$, $F_b(e, i)$ for each event $e$ and process $P_i$ can be computed in $O(N|E|)$ time. For the algorithm refer to Figure 7. Note that the comparison in the condition of the while loop can be performed in constant time. We now construct a directed graph that we call the *skeletal representation* of the slice with respect to $b$ and denote it by $G_b(E)$. The graph $G_b(E)$ has $E$ as the set of vertices and the following edges:

(1) for each event $e$, that is not a final event, there is an edge from $e$ to $succ(e)$, and

(2) for each event $e$ and process $P_i$, there is an edge from $e$ to $F_b(e, i)$.

The skeletal representation of the slice depicted in Figure 4(b) is shown in Figure 6. To prove that the graph $G_b(E)$ is actually cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$, it suffices to show the following:

**Theorem 7** *For all events $e$ and $f$, $J_b(e) \subseteq J_b(f) \equiv (e, f) \in \mathcal{P}(G_b(E))$.*

Input: $J_b(e)$ for each event $e$ on processes $P_i$ and $P_j$.

Output: $F_b(e, j)$ for each event $e$ on $P_i$.

Algorithm:

```
current := ⊥_j;
for each event e on P_i do
    while J_b(e) ⊈ J_b(current) do current := succ(current); endwhile;
    F_b(e, j) := current;
endfor;
```

Figure 7: An algorithm to compute $F_b$ from $J_b$.

*Proof:* ($\Rightarrow$) Consider events $e$ and $f$ such that $J_b(e) \subseteq J_b(f)$ and let $P_i = proc(f)$. By definition of $F_b(e,i)$, $F_b(e,i) \succeq_P f$. By construction, there is an edge from $e$ to $F_b(e,i)$ and a path from $F_b(e,i)$ to $f$ in $G_b(E)$. Thus $(e,f) \in \mathcal{P}(G_b(E))$.

($\Leftarrow$) It suffices to show that $J_b(e) \subseteq J_b(f)$ for each edge $(e,f)$ in $G_b(E)$. Consider an edge of the first kind from an event $e$, $e \notin E.\top$, to its successor, $succ(e)$. Observe that $J_b(succ(e))$ contains $e$. Thus the cut $J_b(e) \cap J_b(succ(e))$ is (1) a consistent cut of $\langle E, \rightarrow \rangle$, (2) contains $e$ and (3) satisfies $b$ (because $b$ is regular). Since $J_b(e)$ is the *least* cut that satisfies these properties, $J_b(e) \subseteq J_b(e) \cap J_b(succ(e))$ implying that $J_b(e) \subseteq J_b(succ(e))$. For an edge of the second kind, by definition of $F_b(e,i)$, $J_b(e) \subseteq J_b(F_b(e,i))$. ■

Besides having computational benefits, the skeletal representation of a slice can be used to devise a simple and efficient algorithm to graft two slices.

# 6 Grafting Two Slices

In this section, we present algorithm to graft two slices which can be done with respect to meet or join. Informally, the former case corresponds to the smallest slice that contains all consistent cuts common to both slices whereas the latter case corresponds to the smallest slice that contains consistent cuts of both slices. In other words, given slices $\langle E, \rightarrow \rangle_{b_1}$ and $\langle E, \rightarrow \rangle_{b_2}$, where $b_1$ and $b_2$ are regular predicates, we provide algorithm to compute the slice $\langle E, \rightarrow \rangle_b$, where $b$ is either $b_1 \sqcap b_2 = b_1 \wedge b_2$ or $b_1 \sqcup b_2 = reg\,(b_1 \vee b_2)$. Grafting enables us to compute the slice for an arbitrary boolean expression of local predicates—by rewriting it in DNF—although it may require exponential time in the worst case. Later, in this section, we present an efficient algorithm based on grafting to compute slice for a *co-regular predicate* (complement of a regular predicate). We also show how grafting can be used to avoid examining many consistent cuts when detecting a predicate under *possibly* modality.

### 6.1 Grafting with respect to Meet: $\mathbf{b} \equiv \mathbf{b_1} \sqcap \mathbf{b_2} \equiv \mathbf{b_1} \wedge \mathbf{b_2}$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if and only if the cut satisfies $b_1$ as well as $b_2$. Let $F_{\min}(e, i)$ denote the earlier of events $F_{b_1}(e, i)$ and $F_{b_2}(e, i)$, that is, $F_{\min}(e, i) = \min\{F_{b_1}(e, i), F_{b_2}(e, i)\}$. The following lemma establishes that, for each event $e$ and process $P_i$, $F_{\min}(e, i)$ cannot occur before $F_b(e, i)$.

**Lemma 8** *For each event $e$ and process $P_i$, $F_b(e, i) \;\underline{\rightsquigarrow}_P\; F_{\min}(e, i)$*

*Proof:* It suffices to prove that $F_b(e, i) \underline{\rightsquigarrow}_P F_{b_1}(e, i)$ and $F_b(e, i) \underline{\rightsquigarrow}_P F_{b_2}(e, i)$ for each event $e$ and process $P_i$. Assume, on the contrary, that $F_{b_1}(e, i) \rightsquigarrow_P F_b(e, i)$ for some event $e$ and process $P_i$. For convenience, let $F_{b_1}(e, i) = f$. Consider $J_b(f)$. Observe that $J_b(f)$ contains $f$. Moreover, since it is a consistent cut of $\langle E, \rightarrow \rangle_b$, it is also a consistent cut of $\langle E, \rightarrow \rangle_{b_1}$. By construction of $G_{b_1}(E)$, any consistent of $\langle E, \rightarrow \rangle_{b_1}$ that contains $f$ also contains $e$ (because $f = F_{b_1}(e, i)$) implying that $J_b(f)$ contains $e$. Since $J_b(f)$ is the *least* consistent cut of $\langle E, \rightarrow \rangle_b$ that contains $f$, there is a path from $e$ to $f$ in $G_b(E)$. Using Theorem 7, $J_b(e) \subseteq J_b(f)$ which contradicts our choice of $F_b(e, i)$. ■

We now construct a directed graph $G_{\min}(E)$ that is similar to $G_b(E)$ except that we use $F_{\min}(e, i)$ instead of $F_b(e, i)$ in its construction. The following theorem proves that $G_{\min}(E)$ is, in fact, cut-equivalent to $G_b(E)$.

**Theorem 9** *$G_{\min}(E)$ is cut-equivalent to $G_b(E)$.*

*Proof:* Observe that any path in $G_{b_1}(E)$ or $G_{b_2}(E)$ is also a path in $G_{\min}(E)$. Formally, $\mathcal{P}(G_{b_1}(E)) \subseteq \mathcal{P}(G_{\min}(E))$ and $\mathcal{P}(G_{b_2}(E)) \subseteq \mathcal{P}(G_{\min}(E))$. Therefore, using Lemma 2, $\mathcal{C}(G_{\min}(E)) \subseteq \mathcal{C}(G_{b_1}(E))$ and $\mathcal{C}(G_{\min}(E)) \subseteq \mathcal{C}(G_{b_2}(E))$. This implies that $\mathcal{C}(G_{\min}(E)) \subseteq \mathcal{C}(G_{b_1}(E)) \cap \mathcal{C}(G_{b_2}(E))$. However, by definition, $\mathcal{C}(G_b(E)) \supseteq \mathcal{C}(G_{b_1}(E)) \cap \mathcal{C}(G_{b_2}(E))$. As a result, $\mathcal{C}(G_{\min}(E)) \subseteq \mathcal{C}(G_b(E))$. Furthermore, using Lemma 8, we can deduce that any path in $G_{\min}(E)$ is a path in $G_b(E)$ or $\mathcal{P}(G_{\min}(E)) \subseteq \mathcal{P}(G_b(E))$. Therefore, again using Lemma 2, $\mathcal{C}(G_b(E)) \subseteq \mathcal{C}(G_{\min}(E))$. Combining the two, we get that $G_{\min}(E) \cong G_b(E)$. ■

Roughly speaking, the aforementioned algorithm computes the union of the sets of edges of each slice. Note that, in general, $F_b(e, i)$ need not be same as $F_{\min}(e, i)$ For example, Figure 8(b) Figure 8(c) show slices of the computation in Figure 8(a) with respect to some regular predicates $b_1$ and $b_2$, respectively. The graft of the two slices is shown in Figure 8(d). Here, $F_{b_1}(a, 3) = \top_3$

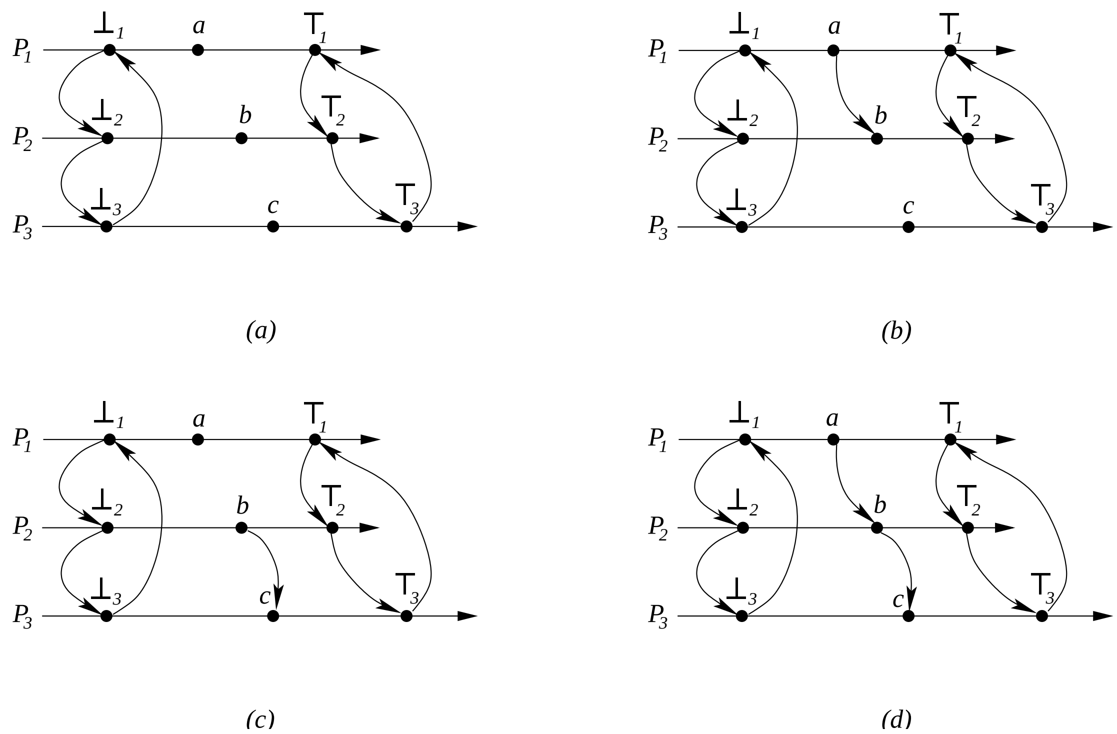


Figure 8: $F_b(e,i)$ may not be same as $\min\{F_{b_1}(e,i), F_{b_2}(e,i)\}$.

and $F_{b_2}(a,3) = \top_3$ but $F_b(a,3) = c$. This algorithm can be generalized to conjunction of an arbitrary number of regular predicates.

### 6.2 Grafting with respect to Join: $\mathbf{b} \equiv \mathbf{b_1} \sqcup \mathbf{b_2} \equiv \mathbf{reg}(\mathbf{b_1} \vee \mathbf{b_2})$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if the cut satisfies either $b_1$ or $b_2$. The dual of the graph $G_{\min}(E)$—min replaced by max—denoted by $G_{\max}(E)$ (surprisingly) turns out to be cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. As before, let $F_{\max}(e,i)$ denote the later of events $F_{b_1}(e,i)$ and $F_{b_2}(e,i)$, that is, $F_{\max}(e,i) = \max\{F_{b_1}(e,i), F_{b_2}(e,i)\}$. The following lemma establishes that, for each event $e$ and process $P_i$, $F_b(e,i)$ cannot occur before $F_{\max}(e,i)$.

**Lemma 10** *For each event $e$ and process $P_i$, $F_{\max}(e,i) \;\underline{\rightsquigarrow}_P\; F_b(e,i)$.*

We now construct a directed graph $G_{\max}(E)$ that is similar to $G_b(E)$ except that we use $F_{\max}(e,i)$ instead of $F_b(e,i)$ in its construction. The following theorem proves that $G_{\max}(E)$ is, in fact, cut-equivalent to $G_b(E)$.

**Theorem 11** *$G_{\max}(E)$ is cut-equivalent to $G_b(E)$.*

The proofs can be found in the appendix. Intuitively, the above-mentioned algorithm computes the intersection of the sets of edges of each slice. In this case, in contrast to the former case, $F_b(e,i)$ is actually identical to $F_{\max}(e,i)$. The reason is as follows. By definition of $F_b(e,i)$—the earliest event on $P_i$ that is reachable from $e$ in the slice $\langle E, \rightarrow\rangle_b$—and Theorem 11, we can infer that $F_b(e,i) \succeq_P F_{\max}(e,i)$. Combining it with Lemma 10, we obtain the required result. This algorithm can be generalized to disjunction of an arbitrary number of regular predicates.

### 6.3 Applications of Grafting

#### 6.3.1 Computing Slices for Co-Regular Predicates

Given a regular predicate, we give an algorithm to compute the slice of a computation with respect to its negation—a co-regular predicate. In particular, we express the negation as disjunction of polynomial number of regular predicates. The slice can then be computed by grafting together slices for each disjunct.

Let $\langle E, \rightarrow\rangle$ be a computation and $\langle E, \rightarrow\rangle_b$ be its slice with respect to a regular predicate $b$. For convenience, let $\rightarrow_b$ be the edge relation for the slice. *We assume that both $\rightarrow$ and $\rightarrow_b$ are transitive relations.* Our objective is to find a property that *distinguishes* the consistent cuts that belong to the slice from the consistent cuts that do not. Consider events $e$ and $f$ such that $e \not\rightarrow f$ but $e \rightarrow_b f$. Then, clearly, a consistent cut that contains $f$ but does not contain $e$ cannot belong to the slice. On the other hand, every consistent cut of the slice that contains $f$ also contains $e$. This motivates us to define a predicate $prevents(f,e)$ as follows:

$$C \text{ satisfies } prevents(f,e) \triangleq (f \in C) \wedge (e \notin C)$$

Here $C$ is a consistent cut of $\langle E, \rightarrow\rangle$. The next lemma establishes that $prevents(f,e)$ is in fact a regular predicate.

**Lemma 12** *$prevents(f,e)$ is a regular predicate.*

*Proof:* Let $C_1$ and $C_2$ be consistent cuts of $\langle E, \rightarrow\rangle$ that satisfy $prevents(f,e)$. Thus $(f \in C_1) \wedge (e \notin C_1)$ and $(f \in C_2) \wedge (e \notin C_2)$ hold. Since $f \in C_1$ and $f \in C_2$, $f \in C_1 \cap C_2$. Also, since $e \notin C_1$, $e \notin C_1 \cap C_2$. Therefore $(f \in C_1 \cap C_2) \wedge (e \notin C_1 \cap C_2)$ holds or, in other words, $C_1 \cap C_2$ satisfies $prevents(f,e)$. Similarly, $C_1 \cup C_2$ satisfies $prevents(f,e)$. ∎

Finally, Theorem 13 establishes that $\neg b$ can be expressed as disjunction of $prevents$'.

**Theorem 13** *Let $C$ be a consistent cut of $\langle E, \rightarrow\rangle$. Then,*

$$C \textit{ satisfies } \neg b \equiv \langle \exists\, e, f : (e \rightarrow_b f) \wedge (e \not\rightarrow f) : C \textit{ satisfies } prevents(f,e)\rangle$$

*Proof:* For convenience, we define a predicate $ensures(f, e)$ as the logical negation of the predicate $prevents(f, e)$. Informally, a consistent satisfies $ensures(f, e)$ if and only if it contains $f$ only if it also contains $e$. Note that if $e$ and $f$ are events such that $e \rightarrow f$ then $C$ must satisfy $ensures(f, e)$. A similar observation holds for a consistent cut of the slice $\langle E, \rightarrow\rangle_b$ ($\cong \langle E, \rightarrow_b\rangle$) when $e \rightarrow_b f$.

($\Leftarrow$) Suppose for some events $e$ and $f$ such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds $C$ satisfies $prevents(f, e)$. Then, clearly, $C$ contains $f$ but does not contain $e$. Consequently, it is not a consistent cut of $\langle E, \rightarrow\rangle_b$ and therefore does not satisfy $b$.

($\Rightarrow$) We prove the contrapositive. Suppose for all events $e$ and $f$ such that $(e \rightarrow_b f)$ and $(e \not\rightarrow f)$ hold $C$ does not satisfy $prevents(f, e)$. Our obligation is to show that $C$ does not satisfy $\neg b$ or, in other words, $C$ satisfies $b$. That is $C$ is a consistent cut of $\langle E, \rightarrow\rangle_b$. It suffices to prove that for all events $e$ and $f$ if $e \rightarrow_b f$ then $C$ satisfies $ensures(f, e)$. Suppose $e \rightarrow_b f$. There are two cases: $e \not\rightarrow f$ or $e \rightarrow f$. In the first case, by our supposition, $C$ satisfies $ensures(f, e)$. In the second case, since $C$ is a consistent cut of $\langle E, \rightarrow\rangle$, $C$ satisfies $ensures(f, e)$. In either case, $C$ satisfies $ensures(f, e)$ which implies that $C$ is a consistent cut of $\langle E, \rightarrow\rangle_b$. This establishes the contrapositive. ∎

For example, consider the computation $\langle E, \rightarrow\rangle$ Figure 9(a) and its lattice of consistent cuts shown in Figure 9(b). Figure 9(c) depicts the sublattice that contains precisely those consistent cuts that satisfy the predicate $b = x_1 \wedge x_2$ and the corresponding slice $\langle E, \rightarrow\rangle_b \cong \langle E, \rightarrow_b\rangle$ is shown in Figure 9(d). The slice contains two additional edges $(\top_1, b)$ and $(\top_2, d)$, thereby introducing paths from $\top_1$ and $\top_2$ to both $b$ and $d$, and paths from $b$ to $d$ and from $d$ to $b$ which were not present in the computation. Without loss of generality, assume that both $\rightarrow$ and $\rightarrow_b$ are transitive, that is, $\rightarrow = \mathcal{P}(\langle E, \rightarrow\rangle)$ and $\rightarrow_b = \mathcal{P}(\langle E, \rightarrow_b\rangle)$. Here, $\top_2 \rightarrow_b d$ but $\top_2 \not\rightarrow d$. Thus $prevents(d, \top_2)$ constitutes one of the disjuncts in the expression for $\neg b = \neg x_1 \vee \neg x_2$ as given by Theorem 13. The subset of consistent cuts that satisfy $prevents(d, \top_2)$, namely $\{W, Y, Z\}$, is depicted in Figure 9(e). As expected, it forms a sublattice of the lattice corresponding to $\mathcal{C}(\langle E, \rightarrow\rangle)$. Similarly, the sublattice for $prevents(b, d)$, another disjunct in the expression for $\neg b$, is shown in Figure 9(f). It can be verified that $\neg b$, as given by Theorem 13, is:

$$\begin{aligned}\neg b \;\equiv\;& prevents(b, \top_1) \vee prevents(b, \top_2) \vee prevents(d, \top_1) \vee \\ & prevents(d, \top_2) \vee prevents(b, d) \vee prevents(d, b)\end{aligned}$$

In this case, $prevents(b, \top_1) \equiv prevents(b, \top_2)$ and $prevents(d, \top_1) \equiv prevents(d, \top_2)$.

### 6.3.2 Pruning State Space for Predicate Detection

Detecting a predicate under *possibly* modality is NP-complete in general [2, 14, 12]. Using grafting, we can reduce the search space for predicates composed from local predicates using

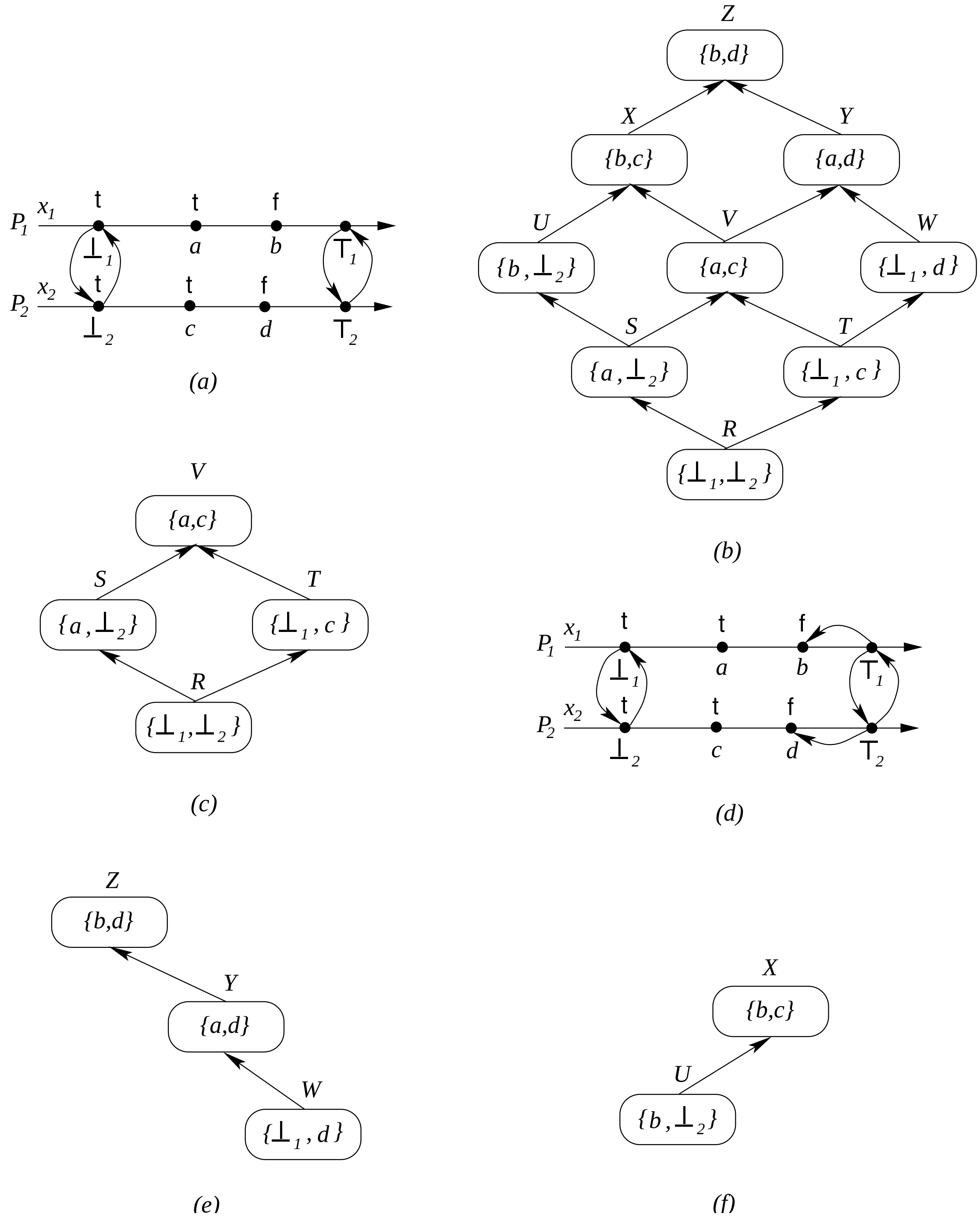


Figure 9: (a) A computation, (b) its lattice of (non-trivial) consistent cuts, (c) sublattice consisting of consistent cuts that satisfy the predicate $x_1 \wedge x_2$, (d) the slice of the computation with respect to the predicate, (e) sublattice of consistent cuts that satisfy $prevents(d, \top_2)$, and (f) sublattice of consistent cuts that satisfy $prevents(b, d)$.

$\neg$, $\wedge$ and $\vee$ operators. We first transform the predicate into an equivalent predicate in which

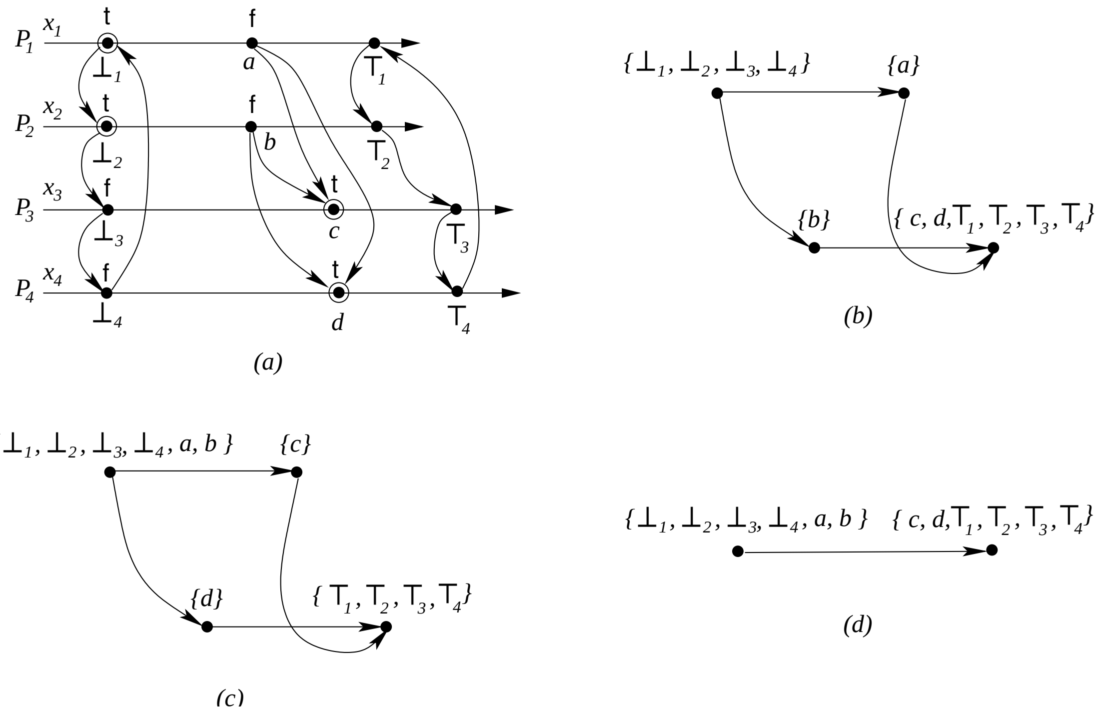


Figure 10: (a) A computation, (b) its slice with respect to $x_1 \vee x_2$, (c) its slice with respect to $x_3 \vee x_4$, and (d) graft of the two slices.

$\neg$ is applied directly to the local predicates and never to more complex expressions. We next build the parse tree for the resultant boolean expression; all local predicates occupy leaf nodes whereas all operators occupy non-leaf nodes. We then recursively compute the slice working our way up from leaf nodes to the root. For a local predicate, we use the algorithm described later in Section 7. For the negation operator, $\neg$, we use the algorithm for computing slice for a co-regular predicate. For conjunction and disjunction operators, $\wedge$ and $\vee$, we use the suitable grafting algorithm depending on the operator.

For example, consider the computation in Figure 10(a) (same as in Figure 2 but with additional fictitious final events and initial events marked explicitly) and the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$. We first compute slices for predicates $x_1$, $x_2$, $x_3$ and $x_4$. We then graft the first two and the last two slices together with respect to join to get slices for each clause as shown in Figure 10(b) and Figure 10(c). For ease of understanding, the events belonging to the same strongly connected component are shown together in a subset. Finally, we graft the slices for both clauses together with respect to meet. The slice obtained will contain all consistent cuts that satisfy the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$. The final slice is depicted in Figure 10(d). In this case, the computation has seven consistent cuts whereas the slice contains only one consistent cut. Note that the final slice corresponds to the predicate $reg\,(reg\,(x_1 \vee x_2) \wedge reg\,(x_3 \vee x_4))$ and not to $reg\,((x_1 \vee x_2) \wedge (x_3 \vee x_4))$. This is expected because detecting even a predicate in

2-CNF when no two clauses contain variables from the same process is NP-complete in general [12].

The result of Section 6.3.1 enables us to generalize this approach to predicates that are composed from arbitrary regular predicates using $\neg$, $\wedge$ and $\vee$ operators. We plan to conduct experiments to quantitatively evaluate the effectiveness of our approach. Although here we focus on detecting a predicate under *possibly* modality, computation slicing can be used to reduce search space for monitoring a predicate under other modalities too.

# 7 Optimal Algorithms for Slicing

The algorithm we presented in [6] to compute slices for regular predicates has $O(N^2|E|)$ time-complexity, where $N$ is the number of processes and $E$ is the set of events. In this section, we present *optimal* algorithms for computing slices for special cases of regular predicates. Our algorithms will have $O(|E|)$ time-complexity. We present the optimal algorithm for conjunctive predicates here. The optimal algorithm for channel predicates is in the appendix.

A *conjunctive predicate* is a conjunction of local predicates. For example, "$P_1$ is in red state" $\wedge$ "$P_2$ is in green state" $\wedge$ "$P_3$ is in blue state". Given a set of local predicates, one for each process, we can categorize events on each process into *true events* and *false events*. An event is a true event if and only if the corresponding local predicate evaluates to true, otherwise it is a false event.

To compute the slice of a computation $\langle E, \rightarrow \rangle$ for a conjunctive predicate $b$, we construct a directed graph $H_b(E)$ with vertices as events in the computation and the following edges:

(1) From an event, that is not a final event, to its successor.

(2) From a send event to the corresponding receive event.

(3) From the successor of a false event to the false event.

For the purpose of building the graph, we assume that all final events are true events. Thus every false event has a successor. The first two kinds of edges ensure that the Lamport's happened-before relation is captured in the graph. The algorithm is illustrated Figure 11. In the figure, all true events have been encircled. We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy the conjunctive predicate.

**Lemma 14** *Every consistent cut of $H_b(E)$ satisfies $b$.*

*Proof:* It is sufficient to prove that no consistent cut of $H_b(E)$ contains a false event in its frontier. Consider a consistent cut $C$ of $H_b(E)$. Assume, on the contrary, that $C$ contains a

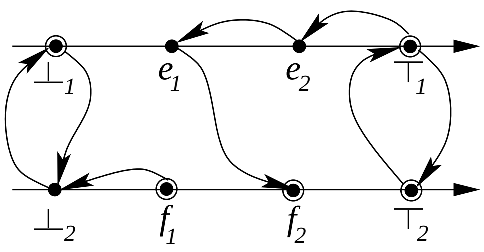


Figure 11: An optimal algorithm to compute the slice for a conjunctive predicate.

false event, say $e$, in its frontier. Since every false event has a successor, by construction, there is an edge from the successor of $e$, say $f$, to $e$. Therefore $f$ also belongs to $C$. This contradicts the fact that $e$ is the last event on its process to be contained in $C$. ■

We next show that the above constructed graph retains all consistent cuts of the computation that satisfy the conjunctive predicate.

**Lemma 15** *Every consistent cut of $\langle E, \rightarrow \rangle$ that satisfies $b$ is a consistent cut of $H_b(E)$.*

*Proof:* Consider a consistent cut $C$ of $\langle E, \rightarrow \rangle$ that satisfies $b$. Assume, on the contrary, that $C$ is not a consistent cut of $H_b(E)$. Thus there exist events $e$ and $f$ such that there is an edge from $e$ to $f$ in $H_b(E)$, $f$ belongs to $C$ but $e$ does not. Since $C$ is a consistent cut of $\langle E, \rightarrow \rangle$, the edge from $e$ to $f$ could only of type (3). Equivalently, $e$ and $f$ occur on the same process, $e$ is the successor of $f$, and $f$ is a false event. Again, since $f$ is contained in $C$ but its successor $e$ is not, $f$ belongs to the frontier of $C$. However, $C$ satisfies $b$ and therefore cannot contain any false event in its frontier. ■

From the previous two lemmas, it follows that:

**Theorem 16** *$H_b(E)$ is cut-equivalent to $\langle E, \rightarrow \rangle_b$.*

It is easy to see that $H_b(E)$ has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be constructed in $O(|E|)$ time. The slice can be computed by finding out the strongly connected components of $H_b(E)$. Thus the algorithm has $O(|E|)$ time-complexity. It also gives us an $O(|E|)$ algorithm to evaluate *possibly*$: b$ when $b$ is a conjunctive predicate (see Theorem 3).

By defining a local predicate (evaluated on an event) to be true if and only if the event corresponds to a local checkpoint, it can be easily verified that there is a *zigzag path* [13, 17] from a local checkpoint $c$ to a local checkpoint $c'$ in a computation if and only if there is a path from *succ*$(c)$, if it exists, to $c'$ in the corresponding slice—which can be ascertained by comparing $J_b(succ(c))$ and $J_b(c')$. An alternate formulation of the consistency theorem in [13] can thus be obtained as:

**Theorem 17** *A set of local checkpoints can belong to the same consistent global snapshot if and only if the local checkpoints in the set are mutually consistent (including with itself) in the corresponding slice.*

Moreover, the R-graph (rollback-dependency graph) [17] is path-equivalent to the slice when each contiguous sequence of false events on a process is collapsed into the nearest true event that occurs later on the process. The minimum consistent global checkpoint that contains a set of local checkpoints [17] can be computed by taking the set union of $J_b$'s for each local checkpoint in the set. The maximum consistent global checkpoint can be similarly obtained by using the dual of $J_b$.

## 8 Conclusion

In this paper, the notion of slice introduced in our earlier paper [6] is generalized and its existence for all global predicates is established. The intractability of computing the slice, in general, is also proved. An optimal algorithm to compute slices for special cases of predicates is provided. Moreover, an efficient algorithm to graft two slices is also given. Application of slicing in general and grafting in particular to global property evaluation of distributed programs is discussed. Finally, the results pertaining to consistent global checkpoints [13, 17] are shown to be special cases of computation slicing.

As future work, we plan to study grafting in greater detail. Specifically, we plan to conduct experiments to quantitatively evaluate its effectiveness in weeding out unnecessary consistent cuts from examination during state space search for predicate detection. Another direction for future research is to extend the notion of slicing to include temporal predicates.

## References

[1] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[3] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1991.

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

[6] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.

[7] V. K. Garg and B. Waldecker. Detection of Unstable Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991.

[8] B. Korel and R. Ferguson. Dynamic Slicing of Distributed Programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.

[9] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In Mariam Kamkar, editor, *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, Linköping, Sweden, May 1997.

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

[11] N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, Portland, Oregon, July 2000.

[12] N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, April 2001.

[13] R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[14] S. D. Stoller and F. Schneider. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, volume 972 of *Lecture Notes in Computer Science*, pages 318–332, France, September 1995.

[15] A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, 1998.

[16] A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[17] Y.-M. Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.

[18] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.

# A Appendix

## A.1 Omitted Proofs

*Proof for Lemma 10:* It suffices to prove that $F_{b_1}(e,i)\underline{\rightsquigarrow}_P F_b(e,i)$ and $F_{b_2}(e,i)\underline{\rightsquigarrow}_P F_b(e,i)$ for each event $e$ and process $P_i$. Assume, on the contrary, that $F_b(e,i) \rightsquigarrow_P F_{b_1}(e,i)$ for some event $e$ and process $P_i$. For convenience, let $f = F_b(e,i)$. Consider $J_{b_1}(f)$. Since $J_{b_1}(f)$ is a consistent cut of $\langle E, \rightarrow\rangle_{b_1}$, it is also a consistent cut of $\langle E, \rightarrow\rangle_b$. Note that any consistent cut of $\langle E, \rightarrow\rangle_b$ that contains $f$ must contain $e$. This is because, by construction of $G_b(E)$, there is an edge from $e$ to $f$ in $G_b(E)$ (as $f = F_b(e,i)$). Observe that $J_{b_1}(f)$ is the *least* consistent cut of $\langle E, \rightarrow\rangle_{b_1}$ that contains $f$, and $J_{b_1}(f)$ contains $e$. Thus there is a path from $e$ to $f$ in $G_{b_1}(E)$. From Theorem 7, $J_{b_1}(e) \subseteq J_{b_1}(f)$ contradicting our choice of $F_{b_1}(e,i)$. ■

*Proof for Theorem 11:* Observe that any path in $G_{\max}(E)$ is a path in $G_{b_1}(E)$ as well as $G_{b_2}(E)$. More precisely, $\mathcal{P}(G_{\max}(E)) \subseteq \mathcal{P}(G_{b_1}(E))$ and $\mathcal{P}(G_{\max}(E)) \subseteq \mathcal{P}(G_{b_2}(E))$. Therefore, using Lemma 2, $\mathcal{C}(G_{b_1}(E)) \subseteq \mathcal{C}(G_{\max}(E))$ and $\mathcal{C}(G_{b_2}(E)) \subseteq \mathcal{C}(G_{\max}(E))$. This implies that $\mathcal{C}(G_{\max}(E)) \supseteq \mathcal{C}(G_{b_1}(E)) \cup \mathcal{C}(G_{b_2}(E))$. However, by definition, $G_b(E)$ is the directed graph with minimum number of consistent cuts that contains all consistent cuts of $G_{b_1}(E)$ as well as $G_{b_1}(E)$. Thus $\mathcal{C}(G_{\max}(E)) \supseteq \mathcal{C}(G_b(E))$. Furthermore, using Lemma 10, we can deduce that any path in $G_b(E)$ is a path in $G_{\max}(E)$ or $\mathcal{P}(G_b(E)) \subseteq \mathcal{P}(G_{\max}(E))$. Therefore, again using Lemma 2, $\mathcal{C}(G_{\max}(E)) \subseteq \mathcal{C}(G_b(E))$. Combining the two, we get that $G_{\max}(E) \cong G_b(E)$. ■

## A.2 Optimal Algorithm to Compute the Slice for Certain Monotonic Channel Predicates

We present an optimal algorithm to compute slices with respect to monotonic channel predicates such as:

- $\bigwedge_{i,j \in [1..n]}$ (at most $k_{ij}$ messages in transit from process $P_i$ to process $P_j$), and
- $\bigwedge_{i,j \in [1..n]}$ (at least $k_{ij}$ messages in transit from process $P_i$ to process $P_j$)

We only provide algorithm for the first predicate here. The algorithm for the second predicate is very similar and has been omitted. Let $snd\langle i,j\rangle(m)$ denote the send event on process $P_i$ that corresponds to the send of $m^{th}$ message to process $P_j$. Similarly, let $rcv\langle i,j\rangle(m)$ denote the receive event on process $P_i$ that corresponds to the receive of $m^{th}$ message from process $P_j$.

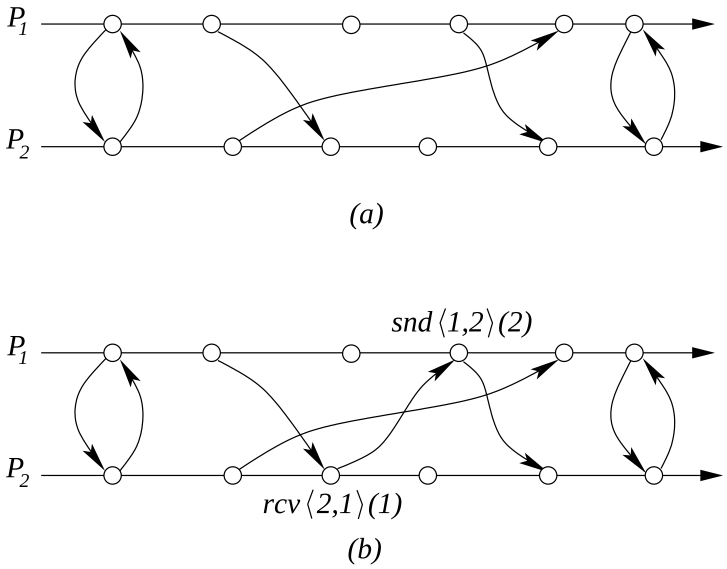


Figure 12: (a) A computation, and (b) its slice with respect to the monotonic channel predicate "at most one message in transit in any channel".

Consider a computation $\langle E, \rightarrow \rangle$ and a monotonic channel predicate $b$ discussed in the previous paragraph. As in the case of conjunctive predicate, we construct a graph $H_b(E)$ with vertices as the events in $E$ and the following edges:

(1) from an event, that is not a final event, to its successor,

(2) from a send event to the corresponding receive event, and

(3) from a receive event $rcv\langle j, i\rangle(m)$ to the send event $snd\langle i, j\rangle(m + k_{ij})$, if it exists.

As before, the first two types of edges ensure that the Lamport's happened-before relation [10] is contained in $\mathcal{P}(H_b(E))$. Consider the computation shown in Figure 12(a) and the monotonic channel predicate "at most one message in transit in any channel". Here, $k_{12} = k_{21} = 1$. The corresponding graph constructed as described is shown in Figure 11(b). We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy the channel predicate.

**Lemma 18** *Every consistent cut of $H_b(E)$ satisfies $b$.*

*Proof:* Consider a consistent cut $C$ of $H_b(E)$ and processes $P_i$ and $P_j$. Let $snd\langle i, j\rangle(m)$ be the send event corresponding to the last message sent by $P_i$ to $P_j$ such that $snd\langle i, j\rangle(m) \in C$. Since $C$ is a consistent cut of $H_b(E)$ and there is an edge from $rcv\langle j, i\rangle(m - k_{ij})$ to $snd\langle i, j\rangle(m)$,

$rcv\langle j,i\rangle(m-k_{ij})$ also belongs to $C$. This implies that there are at most $k_{ij}$ messages in transit from $P_i$ to $P_j$. ∎

We next show that the above constructed graph retains all consistent cuts of the computation that satisfy the channel predicate.

**Lemma 19** *Every consistent cut of $\langle E,\rightarrow\rangle$ that satisfies $b$ is a consistent cut of $H_b(E)$.*

*Proof:* Consider a consistent cut $C$ of $\langle E,\rightarrow\rangle$ that satisfies $b$. Assume, on the contrary, that $C$ is not a consistent cut of $H_b(E)$. Thus there exist events $e$ and $f$ such that there is a path from $e$ to $f$ in $H_b(E)$, $f$ belongs to $C$ but $e$ does not. Since $C$ is a consistent cut of $\langle E,\rightarrow\rangle$, the edge from $e$ to $f$ could only of type (3). Let $e$ be $rcv\langle j,i\rangle(m)$ and $f$ be $snd\langle i,j\rangle(m+k_{ij})$. Since $C$ satisfies $b$, $rcv\langle j,i\rangle(m)$ or, equivalently, $e$ belongs to $C$—a contradiction. ∎

From the previous two lemmas, it follows that:

**Theorem 20** *$H_b(E)$ is cut-equivalent to $\langle E,\rightarrow\rangle_b$.*

It is easy to see that the graph $H_b(E)$ has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in $O(|E|)$ time. Thus the algorithm has $O(|E|)$ overall time-complexity.