Finding Satisfying Global States: All for One and One for All

Neeraj Mittal†, Alper Sen‡, Vijay K. Garg‡* and Ranganath Atreya†

†Dept. of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
neerajm@utdallas.edu
atreya@student.utdallas.edu

‡Dept. of Electrical and Computer Engg.
University of Texas at Austin
Austin, TX 78712, USA
sen@ece.utexas.edu
garg@ece.utexas.edu

Abstract

Given a distributed computation and a global predicate, predicate detection involves determining whether there exists *at least one* consistent cut (or global state) of the computation that satisfies the predicate. On the other hand, computation slicing is concerned with computing the smallest sub-computation—with the least number of consistent cuts—that contains *all* consistent cuts of the computation satisfying the predicate. In this paper, we investigate the relationship between predicate detection and computation slicing and show that the two problems are equivalent. Specifically, given an algorithm to detect a predicate b in a computation C, we derive an algorithm to compute the slice of C with respect to b. The time-complexity of the (derived) slicing algorithm is $O(n|E|)$ times the time-complexity of the detection algorithm, where n is the number of processes and E is the set of events. We discuss how the "equivalence" result of this paper can be utilized to derive a faster algorithm for solving the *general* predicate detection problem.

Slicing algorithms described in our earlier papers are all *off-line* in nature. In this paper, we also give an *on-line* algorithm for computing the slice for a predicate. The amortized time-complexity of the algorithm is $O(n(c+n))$ times the time-complexity of the detection algorithm, where c is the *average concurrency* in the computation.

Key words: analyzing distributed computation, predicate detection, global property evaluation, computation slicing, testing and debugging, on-line algorithm

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

1 Introduction

Writing correct distributed programs is a non-trivial task. Not surprisingly, distributed systems are particularly vulnerable to software faults. Testing and debugging is an effective way of improving the dependability of a software prior to its deployment. Software bugs that do persist after extensive testing and debugging have to be tolerated at runtime to ensure that the system continues to operate properly. Detecting a fault in an execution of a distributed system (*e.g.*, violation of mutual exclusion) is a fundamental problem that arises during testing and debugging as well as software fault tolerance.

In this paper, we focus on detecting those faults that can be expressed as predicates on variables of processes. For example, "no process has the token" can be written as $no\_token_1 \wedge no\_token_2 \wedge \cdots \wedge no\_token_n$, where no_token_i denotes the absence of token on process p_i. This gives rise to the *predicate detection problem* which involves determining whether there exists a consistent cut (or global state) of a distributed computation that satisfies given global predicate. (This problem is also referred to as detecting a predicate under *possibly* modality in the literature.) For example, a programmer debugging an implementation of a distributed mutual exclusion algorithm may want to test whether a given execution of the system contains a global state for which two or more processes are in their critical sections.

Detecting a global predicate in a distributed computation is a hard problem in general [18, 8]. The reason is the combinatorial explosion in the number of possible consistent cuts. Given n processes such that each process executes at most k events, the number of possible consistent cuts in the computation could be as large as $O(k^n)$. Finding a consistent cut that satisfies the given predicate may, therefore, require looking at a large number of consistent cuts. Many approaches to predicate detection exploit the structure of the predicate to detect it efficiently in a given computation. (Hereafter, when we say that an algorithm is efficient, it means that its (worst-case) time-complexity is polynomial in input size.) Polynomial-time algorithms have been developed for several useful classes of predicates (*e.g.*, conjunctive predicates [8], relational predicates [4] and so on).

In our earlier papers [9, 14], we introduce the notion of *computation slice*. Intuitively, slice is a concise representation of consistent cuts satisfying a certain condition. The slice of a computation with respect to a predicate is the sub-computation with the least number of consistent cuts that contains all consistent cuts of the computation for which the predicate evaluates to true. We show in [9] that slice is uniquely defined for all predicates. Suppose the number of consistent cuts of the slice is much smaller than those of the computation. Then, clearly, in order to detect a fault, rather than searching the state-space of the computation, it is more effective to search the state-space of the slice. We also identify a class of predicates, called *regular predicates*, for which the slice is *lean* [9, 14]. That is, the slice for a regular predicate contains precisely those consistent cuts for which the predicate evaluates to true.

As an illustration, suppose we want to detect the predicate $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3) \wedge (x_1 * x_2 + x_3 < 5)$ 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 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 2. The first event on each process (namely a on p_1, q on p_2 and u on p_3) "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, thirty in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute the slice of the computation with respect to the predicate

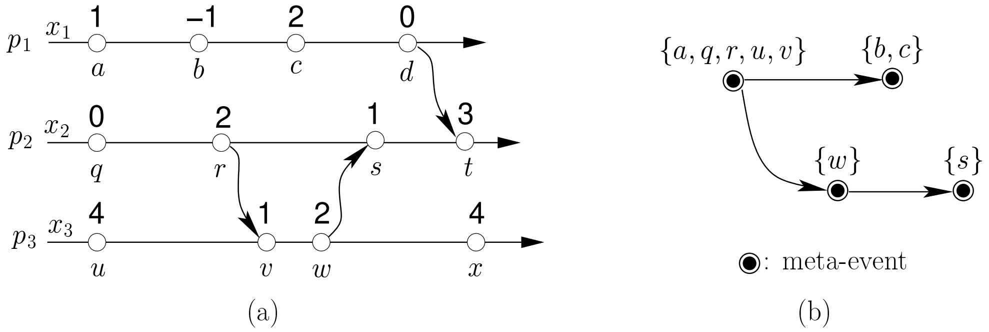

Figure 1: (a) A computation and (b) its slice with respect to $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$.

$(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ as follows. Immediately after executing b, the value of x_1 becomes -1 which does not satisfy $x_1 \geqslant 1$. To reach a consistent cut satisfying $x_1 \geqslant 1$, c has to be executed. In other words, any consistent cut in which only b has been executed but not c is of no interest to us and can be ignored. The slice is shown in Figure 1(b). It is modeled by a partial order on a set of meta-events; each *meta-event* consists of one or more "primitive" events. A consistent cut of the slice either contains all the events in a meta-event or none of them. (Intuitively, any consistent cut of the computation that contains only a partial set of events in a meta-event is of no relevance to us.) Moreover, a meta-event "belongs" to a consistent cut only if all its incoming neighbours are also contained in the cut. We can now restrict our search to the consistent cuts of the slice which are only six in number, namely $\{a, q, r, u, v\}$, $\{a, q, r, u, v, b, c\}$, $\{a, q, r, u, v, w\}$, $\{a, q, r, u, v, b, c, w\}$, $\{a, q, r, u, v, w, s\}$ and $\{a, q, r, u, v, b, c, w, s\}$. The slice has much fewer consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

The predicate detection problem is only concerned with determining whether there exists *at least one* consistent cut of the computation that satisfies the given predicate. Computation slicing, on the other hand, is concerned with computing (a succinct representation of) *all* consistent cuts of the computation for which the given predicate evaluates to true. Clearly, detecting a predicate is no harder than computing its slice in the sense that the predicate detection problem can be easily solved given the slice for the predicate (it suffices to test for the emptiness of the slice). In this paper, we prove the converse that detecting a predicate is no easier than computing its slice. In other words, given an algorithm A for detecting a predicate b, there exists an algorithm B for computing the slice for b such that the time-complexity of B is at most $O(n|E|)$ times the time-complexity of A, where n is the number of processes and E is the set of events. As a corollary, it can be derived that there exists a polynomial-time algorithm for detecting a predicate if and only if there exists a polynomial-time algorithm for computing its slice. Some examples of predicates for which we can now compute the slice efficiently include relational predicates [4] and co-linear predicates [8].

At first glance, it may seem that we are not any better off than we were before. After all, if predicate detection is equivalent to computation slicing, then how can slicing be used to improve to speed-up predicate detection? The following example illustrates that slicing can indeed facilitate predicate detection. Suppose we want to detect a predicate b which is a conjunction of two clauses b_1 and b_2. Further, suppose both b_1 and b_2 are relational predicates given by $x_1 + x_2 + \cdots + x_n \leqslant c$ and $y_1 + y_2 + \cdots + y_n \leqslant d$, respectively, where for each i, x_i and y_i are variables on process p_i, and c and d are constants. (Relational predicates are useful because they can be used to express conditions such as "at most k processes are in their critical sections at the same time".) Although both b_1 and b_2 can be independently detected in an efficient manner [4], the same does not apply for b. In particular, if the computation does not contain any consistent cut satisfying b_1 or b_2, then it also does not contain any consistent cut satisfying b. On the other hand, if the computation contains a consistent cut satisfying b_1 and also a consistent cut satisfying b_2, then it does not imply that it also contains a consistent cut satisfying b. This is because the set of consistent cuts that satisfy b_1 may be disjoint from the set of consistent cuts that satisfy b_2. In fact, we are not aware of any algorithm that can detect b efficiently. Therefore, to detect b, without computation slicing, we are forced to use techniques such as *breadth first search* [5], *depth first search* [1], and *partial-order methods* (a model-checking technique) [19]. *None of the above techniques* take advantage of the fact that b_1 and b_2 can be detected efficiently albeit independently. Using computation slicing technique, however, we can proceed as follows. First, we compute slices of the computation for b_1 and b_2, say S_1 and S_2, respectively. Next, we *compose* the slices S_1 and S_2 together to obtain the slice S, which contains consistent cuts common to both S_1 and S_2 [14]. Both steps have polynomial-time complexity. Finally, to detect b, clearly, it suffices to search the state-space of the slice S. In case only a small number of consistent cuts satisfy b, the state-space of S is likely to be small resulting in exponential speed-up. Note that if a relatively large number of consistent cuts satisfy a predicate, then almost all techniques for detecting the predicate would fair well on average. The technique based on computation slicing is most useful when only a relatively small number of consistent cuts satisfy the predicate. In that case, by spending only polynomial amount of time in computing the slice we can potentially throw away exponential number of consistent cuts, thereby obtaining an exponential speed-up overall. Our experimental results indicate that slicing can indeed lead to an exponential improvement over existing techniques for predicate detection in terms of time and space [15]. Thus, the result of this paper, rather than diminishing the benefits of computation slicing, actually enhances them by expanding the class of predicates for which the slice can be computed efficiently in polynomial-time.

Observe that other techniques for reducing the time-complexity [19] and/or the space-complexity [1] of predicate detection are orthogonal to slicing, and as such can be used in conjunction with slicing. Slicing can also be employed to reduce the search-space for detecting a predicate under other modalities including *definitely*, *invariant* and *controllable* [5, 14, 8] and their nestings, which generates a subset of temporal logic [17]. Other applications include tool for debugging a distributed program and a more effective visualization of a distributed computation using event abstraction [11]. For instance, a programmer can use slicing to focus his attention on only faulty consistent cuts of a computation, which may provide a valuable insight into the bug that caused the fault. An examination of the structure of the slice may be used to locate potentially problematic events.

Although in this paper our focus is on distributed systems, slicing has applications in areas such as combinatorics [7] as well. A combinatorial problem usually requires counting or enumerating structures that satisfy a given property. In [7], we cast the combinatorial problem as a distributed computation such that there is a bijection between the combinatorial structures satisfying a property b and the consistent cuts that satisfy a property equivalent to b. We then apply results in slicing a computation with respect to a predicate to obtain a slice consisting of only those consistent cuts that satisfy the desired property. This gives us an efficient algorithm to count or enumerate structures that satisfy b when the total set of structures is large but the set of structures satisfying b is small. Several applications of slicing for analyzing problems in integer partitions, set families, and the set of permutations are given in [7].

The algorithms described in our earlier papers [9, 14, 17] for computing a slice are all *off-line* in nature; they assume that the entire set of events is available *a priori*. While this is quite adequate for applications such as testing and debugging, for other applications such as software fault tolerance, it is desirable that the slice be computed incrementally in an *on-line* manner; as and when a new event is generated, the current slice is updated to reflect its arrival. The reason is that for software fault tolerance, it is important to detect the fault as early as possible before it can cause any severe damage. If the slice is computed only after a certain number of events have been collected and then analyzed for the presence of a faulty consistent cut, it may be too late for any meaningful recovery. At the same time, whenever an event arrives, the cost of incrementally updating the slice should be less than the cost of recomputing the slice from scratch using an off-line algorithm. The on-line algorithm is also useful when slicing is used to visualize a computation *in progress*. For instance, when debugging an implementation of distributed mutual exclusion algorithm, the programmer may want to "look" at only those consistent cuts that violate mutual exclusion. In this paper, we give an efficient incremental algorithm to compute the slice for a predicate given an efficient (off-line) algorithm to detect the predicate. The amortized time-complexity of the algorithm is $O(n(c+n))$ times the time-complexity of the detection algorithm, where c is the *average concurrency* in the computation. We define average concurrency in the computation to be the ratio of the number of concurrent pairs (including reflexive pairs) to the number of events. The average concurrency of a computation lies between 1 and $|E|$.

To summarize, in this paper, we prove that the problem of detecting a predicate in a computation is equivalent to the problem of computing the slice for the predicate. In addition, we give an efficient on-line algorithm for computing the slice for a predicate that can be detected efficiently. Due to the lack of space, the proofs of all lemmas and theorems have been moved to the appendix.

2 Model and Notation

We assume a loosely-coupled system consisting of n processes denoted by $P = \{p_1, p_2, \ldots, p_n\}$ communicating via asynchronous messages. We do not assume any shared memory or global clock. Processes execute events. Events on the same process are totally ordered. However, events on different processes are only partially ordered. Therefore, traditionally, a distributed computation is modeled as a partial order on a set of events [12]. 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 its vertices and edges, respectively. A subset of vertices of a directed graph forms a *consistent cut* if the subset contains a vertex only if it also contains all its incoming neighbours. Formally,

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

Note that a consistent cut either contains all vertices in a strongly connected component or none of them. 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. Also, 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 every vertex has a path to itself.

A *distributed computation* (or simply a *computation*) $\langle E, \rightarrow \rangle$ is 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 $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [12]. A distributed computation in our model can contain cycles. This is because whereas a computation in the traditional or 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 constitutes a *meta-event*.

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. When two events e and f occurred on the same process and e occurred before f in real-time, then we write $e \stackrel{P}{\rightarrow} f$, and let $\stackrel{P}{\underline{\rightarrow}}$ be the reflexive closure of $\stackrel{P}{\rightarrow}$. We assume the presence of fictitious initial and final events on each process. The initial event on process p_i, denoted by $\bot_i$, occurs before any other event on p_i. Likewise, the final event on process p_i, denoted by $\top_i$, occurs after all other events on p_i. We use final events only to ease the exposition of the slicing algorithms given in this paper. It *does not imply* that processes have to synchronize with each other at the end of the computation. For convenience, let $\bot$ and $\top$ denote the set of all initial events and final events, respectively. 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. Thus, every consistent cut of a computation in the traditional model is a non-trivial consistent cut of the corresponding computation in our model and vice versa. Only non-trivial consistent cuts are of interest to us.

A *global predicate* (or simply a *predicate*) is a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated on the state resulting after executing all events in the cut. If a predicate b evaluates to true for a consistent cut C, we say that "C satisfies b". We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* if it depends on variables of a single process.

Example 1 *Consider the computation depicted in Figure 2(a). It has three processes, namely p_1, p_2 and p_3. The events e_1, f_1 and g_1 are the initial events, and the events e_4, f_4 and g_4 are the final events of the computation. The cut $A = \{e_1, e_2, e_3, e_4, f_1, g_1\}$ is not consistent because $g_4 \rightarrow e_4$ and $e_4 \in A$ but $g_4 \notin A$. The cut $\{e_1, e_2, f_1, f_2, g_1\}$ is consistent. 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 $x \leqslant 1$ is local whereas the predicate $x + y \leqslant z$ is not. The consistent cut $\{e_1, f_1, g_1\}$ satisfies $x + y \leqslant z$ but the consistent cut $\{e_1, e_2, f_1, f_2, g_1\}$ does not.* □

3 Background

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

3.1 Birkhoff's Theorem

We first describe some concepts needed to understand the theorem. A lattice is called *distributive* if its meet operator distributes over its join operator [6]. An element of a lattice is called *join-irreducible* if it cannot be expressed as join of two distinct elements (of the

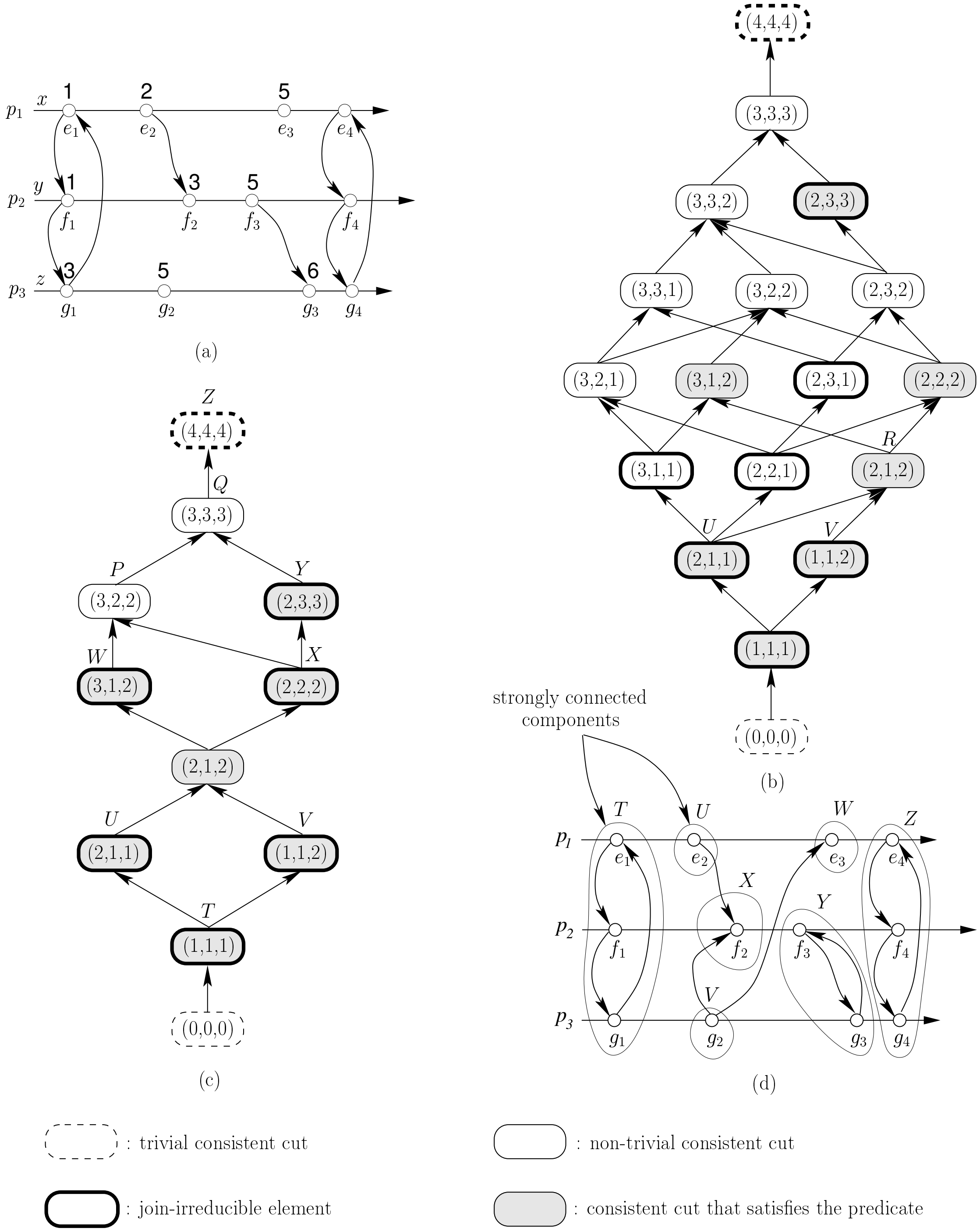

Figure 2: (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leqslant 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

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 that 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 $\langle E, \rightarrow \rangle$ Recall that the set of consistent cuts of $\langle E, \rightarrow \rangle$ is denoted by $\mathcal{C}(\langle E, \rightarrow \rangle)$. For reasons of clarity, we abbreviate $\mathcal{C}(\langle E, \rightarrow \rangle)$ by $\mathcal{C}(E)$. It can be proved 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 [13]. We show that the set $\mathcal{C}(E)$ satisfies no additional structural property [9, 14]. Further, the set of join-irreducible elements of $\mathcal{C}(E)$ is isomorphic to the set of strongly connected components of $\langle E, \rightarrow \rangle$ [9, 14].

Example 2 *Consider the computation shown in Figure 2(a). The (distributive) lattice spanned by the set of its consistent cuts is shown in Figure 2(b). 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 in the Hasse diagram.) 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 written as the join of some subset of these eight join-irreducible elements and vice versa. For instance, R (in Figure 2(b)) can be expressed as the join of U and V.* □

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.

3.2 Computation Slice

Informally, a *computation slice* (or simply a *slice*) is a concise representation of all those consistent cuts of the computation that satisfy a given predicate. For a computation $\langle E, \rightarrow \rangle$ and a predicate b, we use $\mathcal{C}_b(E)$ to denote the subset of those consistent cuts of $\mathcal{C}(E)$ that satisfy b. Let $\mathcal{I}_b(E)$ denote the set of all graphs on vertices E such that for every graph $G \in \mathcal{I}_b(E)$, $\mathcal{C}_b(E) \subseteq \mathcal{C}(G) \subseteq \mathcal{C}(E)$. We now define computation slice formally.

Definition 1 (slice [14]) *A* slice *of a computation with respect to a predicate is a directed graph with the least number of consistent cuts such that the graph contains all consistent cuts of the computation for which the predicate evaluates to true. Formally, given a computation $\langle E, \rightarrow \rangle$ and a predicate b,*

$$S \textit{ is a slice of } \langle E, \rightarrow \rangle \textit{ for } b \;\triangleq\; \langle \forall\, G : G \in \mathcal{I}_b(E) : |\mathcal{C}(S)| \leqslant |\mathcal{C}(G)| \rangle$$

We use $\mathsf{slice}(\langle E, \rightarrow \rangle, b)$ to denote a slice of $\langle E, \rightarrow \rangle$ with respect to b. Note that $\langle E, \rightarrow \rangle = \mathsf{slice}(\langle E, \rightarrow \rangle, \mathsf{true})$. Therefore a computation can be viewed as a slice. Likewise, a slice can be viewed as a computation at some level of abstraction [10].

We show in [14] that slice of a computation is *uniquely defined* for every predicate in the sense that if two graphs S and T constitute a slice of $\langle E, \rightarrow \rangle$ for b (as per the definition) then $\mathcal{C}(S) = \mathcal{C}(T)$. Moreover, S and T will have identical meta-events. The main idea behind the proof is as follows. We show that there exists a unique subset $\mathcal{D} \subseteq \mathcal{C}(E)$ satisfying the following conditions. First, $\mathcal{D}$ contains $\mathcal{C}_b(E)$, that is, $\mathcal{C}_b(E) \subseteq \mathcal{D}$. Second, 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 partially ordered set (or poset) induced on the elements of $\mathcal{JI}(\mathcal{D})$ by the relation $\subseteq$ as the slice of $\langle E, \rightarrow \rangle$ with respect to b. 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 E such that its set of consistent cuts is given by $\mathcal{D}$. Such a graph can be obtained by simply forming a strongly connected component out of each meta-event. The *poset representation* of a slice—given by a partial order on the set of meta-events (*e.g.*, Figure 1(b))—is better for presentation purposes. On the other hand, the *graph representation*—given by a directed graph on the set of events (*e.g.*, Figure 2(d))—is more suited for developing slicing algorithms.

Example 3 *Consider the lattice of consistent cuts depicted in Figure 2(b). The consistent cuts that satisfy the predicate $x+y-z \leqslant 1$ have been shaded in the figure. Figure 2(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 2(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 2(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.* □

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 [14]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. Efficient algorithms for computing the slice for many useful classes of predicates such as regular predicates and their complements can be found elsewhere [9, 14, 15, 17].

4 The Two Problems

In this section, we study the relationship between the following two problems:

Containing Cut (CONTC) Given a directed graph G and a predicate b, does there exist a consistent cut of G that satisfies b?

Computing Slice (COMPS) Given a directed graph G and a predicate b, compute the slice of G with respect to b.

We say that CONTC and COMPS are *equivalent* if the following holds for every predicate b: given an algorithm U for solving CONTC(G, b) for all G, we can derive an algorithm V for solving COMPS(H, b) for all H such that the time-complexity of V is within a polynomial factor of the time-complexity of U, and vice versa. We say that an algorithm is *efficient* if its (worst-case) time-complexity is polynomial in input size. Also, a predicate is *efficiently detectable* if there exists an efficient algorithm to detect the predicate in a computation.

Note that since both CONTC and COMPS are NP-complete problems in general, they are equivalent in another sense: given a polynomial-time algorithm U to solve CONTC(G, b) for all G and b, we can derive a polynomial-time algorithm V to solve $\mathsf{COMPS}(H,c)$ for all H and c such that the time-complexity of V is within a polynomial factor of the time-complexity of U, and vice versa. This notion of equivalence is weaker than our notion of equivalence. This is because if we want to solve an instance of COMPS, say for G and b, then, using the transformation that exists between any two NP-complete problems, we may be required to solve an instance of CONTC for a predicate different from b.

4.1 The Main Result: Equivalence of COMPS and CONTC

From the definition of slice, clearly, it follows that the slice for a directed graph with respect to a predicate (not necessarily regular [9]) is non-empty if and only if the graph contains a consistent cut that satisfies the predicate. Formally,

$$\mathsf{CONTC}(G,b) \;\equiv\; \mathsf{slice}(G,b) \text{ is non-empty}$$

Therefore $\mathsf{COMPS}(G,b)$ is at least as hard as $\mathsf{CONTC}(G,b)$. We now prove the converse. Consider a directed graph G and a predicate b. Now, G and $\mathsf{slice}(G,b)$ are directed graphs on identical sets of vertices. However, more pairs of vertices are "connected" in $\mathsf{slice}(G,b)$ than in G. In the next lemma, we give a complete characterization of the pairs of vertices that are "connected" in $\mathsf{slice}(G,b)$. Let $G[e,f]$ denote the directed graph obtained by adding an edge from e to f in G.

Lemma 1 *There is a path from an event e to an event f in $\mathsf{slice}(G,b)$ if and only if no consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ satisfies b.*

Lemma 1 is useful provided it is possible to ascertain efficiently whether some consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ satisfies b. To that end, we show that the set $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ actually forms a sublattice and therefore can be captured faithfully using a directed graph. Let $\widehat{G}[e,f]$ denote the directed graph obtained by adding an edge from f to $\bot_1$ and an edge from $\top_1$ to e. It suffices to show the following:

Lemma 2 $\mathcal{C}(\widehat{G}[e,f]) \setminus \{\emptyset, E\} \;=\; \mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$

Combining the two lemmas, we obtain the following:

Theorem 3 *There is a path from an event e to an event f in $\mathsf{slice}(G,b)$ if and only if no consistent cut in $\widehat{G}[e,f]$ satisfies b, that is, $\mathsf{CONTC}(\widehat{G}[e,f],b)$ evaluates to false.*

Figure 3 depicts the algorithm for solving COMPS using an algorithm that solves CONTC. The algorithm constructs a directed graph that is transitively closed.

Theorem 4 *The time-complexity of the algorithm for solving COMPS described in Figure 3 is $O(|E|^2T)$, where E is the set of events and $O(T)$ is the worst-case time-complexity of solving CONTC.*

In order to reduce the time-complexity of the algorithm, we construct a graph that is not transitively closed but whose set of consistent cuts is the same as that of the slice. Such a graph is called the skeletal representation of a slice [14]. For an event e, let $F_b(e)$ denote a vector of events; the i^{th} entry of the vector refers to the earliest event f on process p_i such that there is a path from e to f in the slice [14]. (We assume that every event has

Input: (1) a directed graph G, (2) a predicate b, and
(3) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: the slice of G with respect to b

$K := G$;
for every pair of events (e, f) do
    if not(CONTC$(\widehat{G}[e, f], b)$) then
        add an edge from e to f in K; // *set K to $K[e, f]$*
    endif;
endfor;
output K;

Figure 3: An algorithm to solve COMPS using an algorithm to solve CONTC.

Input: (1) a directed graph G, (2) a predicate b, (3) a process p_x, and
(4) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: $F_b(e)$ for all events e on p_x

for each process p_i do // *compute $F_b(e)[i]$ for all events e on p_x*
    $f := \perp_i$;
    for each event e on p_x do // *visited in the order given by $\xrightarrow{P}$*
        while CONTC$(\widehat{G}[e, f], b)$ do
            $f := succ(f)$; // *advance to the next event on p_i*
        endwhile;
        $F_b(e)[i] := f$;
    endfor;
endfor;

Figure 4: An algorithm to compute $F_b(e)$ for all events e on process p_x.

a path to itself.) We now construct a graph with the following edges. First, there is an edge from an event to its successor, if it exists. Second, there is an edge from an event e to every event in $F_b(e)$. We show in [14] that the graph so obtained has the same set of consistent cuts as the slice. Further, it has only $O(n|E|)$ edges, where n is the number of processes. It is easy to verify that F_b is order-preserving which means that if $e \rightarrow f$ then $F_b(e)[i] \xrightarrow{P} F_b(f)[i]$ for each process p_i [14]. Consequently, it is possible to compute $F_b(e)[i]$ for *all* events e on a single process by scanning the computation *once* from left to right. The algorithm is presented in Figure 4. The next theorem establishes that the time-complexity of the algorithm is $O(n|E|T)$.

Theorem 5 *The time-complexity of the algorithm for computing $F_b(e)$ for all events e in Figure 4 is $O(n|E|T)$, where n is the number of processes, E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC.

4.2 Applications of the Result

Predicate detection is an important problem in distributed systems. Efficient detection algorithms have been developed for several useful classes of predicates. Examples include stable predicates [2], observer-independent predicates [3], conjunctive predicates [8], linear predicates [8], relational predicates [4], and their complements such as co-stable predicates and co-linear predicates [16]. In our earlier papers, we give efficient algorithms for computing the slice for regular predicates [9], co-regular predicates [14], linear predicates and k-local predicates for constant k [15]. Using the result of this paper, it is now possible to compute the slice efficiently for many more classes of predicates including stable and co-stable predicates, observer independent predicates, co-linear predicates, and relational predicates. For instance, an observer independent predicate can be detected in $O(n|E|)$ time using the algorithm presented in [3]. This implies that its slice can be computed in $O(n^2|E|^2)$ time using the algorithm given in Section 4.1. It is possible that a faster and more efficient slicing algorithm exists for an observer-independent predicate, which perhaps exploits the specific properties of the class of observer-independent predicates. Our result is still useful because it gives a ready-made algorithm for computing the slice.

Since the problem of predicate detection is NP-complete in general [18, 8], the problem of computing the slice is also NP-complete. For such predicates, we can still compute an *approximate slice* efficiently; an approximate slice may be bigger than the actual slice but much smaller than the computation itself. Using our algorithms for composing slices in [14] and the result of this paper, it is now possible to compute an approximate slice in polynomial-time for a predicate composed from "efficiently detectable" predicates using $\wedge$ and $\vee$ operators. An example of such a predicate is $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_{n-1} \vee x_n)$, where x_i is a boolean variable on process p_i. This is significant because our experimental results show that using an approximate slice instead of the computation can lead to an exponential improvement over existing techniques for predicate detection in terms of time and space [15].

5 An On-Line Algorithm for Computing the Slice

In this section, we present an on-line algorithm for computing the slice for a predicate for which the slice can indeed be computed efficiently in an off-line manner. Our on-line slicing algorithm is basically derived from the off-line algorithm for computing the slice described in Figure 4. On generation of a new event in the system, our on-line algorithm updates the current slice to reflect the arrival of the new event.

Before discussing the algorithm, we state our assumptions and describe some notation. We assume that a newly arrived event is "enabled" in the sense that all events that happened-before it have already arrived and been incorporated into the slice. This can be achieved by buffering the new event—in case it is not "enabled"—and processing it later when it becomes "enabled". Whether an event is "enabled" can be determined efficiently by examining its Fidge/Mattern's vector timestamp.

Initially, the computation consists of only the fictitious—initial and final—events. Let the k^{th} arriving event, $k \geqslant 1$, be denoted by $e^{(k)}$, and let $G^{(k)}$ denote the resulting computation. Sometimes we represent the computation more explicitly using $\langle E^{(k)}, \rightarrow \rangle$ whenever necessary, where $E^{(k)}$ denote the set of events and $\rightarrow$ denote the set of edges in $G^{(k)}$. Without loss of generality, assume that $G^{(k)}$ is a transitively closed graph and thus $\rightarrow$ is a transitive relation. Note that $\rightarrow$ on the set of non-fictitious events defines the Lamport's happened-before relation. Clearly, every non-trivial consistent cut of $G^{(k-1)}$ is a consistent cut of $G^{(k)}$ as well. Furthermore, every consistent cut of $G^{(k)}$ that is not a consistent cut of $G^{(k-1)}$ contains $e^{(k)}$.

The on-line algorithm, whenever a new event arrives, computes the new slice by updating $F_b(e)$ for each event e. We use ${F_b}^{(k)}$ to refer to the value of F_b for the computation $G^{(k)}$. Now, in order to incorporate an event into the slice, we may have to recompute the entry $F_b(e)[i]$ for each event e and every process p_i. First, we show that the new value for an entry cannot move "backward" in the space-time diagram. Let p_{i_k} denote the process on which the event $e^{(k)}$ occurred. An event $e \in E^{(k-1)}$ is said to be a *critical event* if $F_b^{(k-1)}(e)[i_k] = \top_{i_k}$. Intuitively, no nonfinal event on p_{i_k} is reachable from e in $\mathsf{slice}(G^{(k-1)}, b)$. This may change, however, on arrival of $e^{(k)}$ because $e^{(k)}$ is an event on p_{i_k}. Let $critical(k)$ denote the set of all events in $E^{(k-1)}$ that are critical with respect to $e^{(k)}$. Formally,

Lemma 6 *Given an event $e \in E^{(k-1)}$ and a process p_i,*

$$(i \neq i_k) \vee (e \notin critical(k)) \;\Rightarrow\; F_b^{(k-1)}(e)[i] \overset{P}{\rightarrow} F_b^{(k)}(e)[i] \tag{6.1}$$

$$(i = i_k) \wedge (e \in critical(k)) \;\Rightarrow\; F_b^{(k)}(e)[i] \in \{e^{(k)}, \top_i\} \tag{6.2}$$

Lemma 6 may greatly restrict the amount of work that needs to be done in order to recompute F_b. In particular, to determine the new value of $F_b(e)[i]$ for an event e and a process p_i, rather than starting the scan from $\perp_i$, we can instead start the scan from the old value of $F_b(e)[i]$. The next lemma specifies the conditions under which either $F_b(e)[i]$ will not change or can be determined cheaply.

Lemma 7 *Given an event $e \in E^{(k-1)}$ and a process p_i,*

$$(e \rightarrow e^{(k)}) \wedge \Big((i \neq i_k) \vee (e \notin critical(k))\Big) \;\Rightarrow\; F_b^{(k-1)}(e)[i] = F_b^{(k)}(e)[i] \tag{7.1}$$

$$(e \rightarrow e^{(k)}) \wedge \Big((i = i_k) \wedge (e \in critical(k))\Big) \;\Rightarrow\; F_b^{(k)}(e)[i] = e^{(k)} \tag{7.2}$$

Lemma 7 implies that F_b needs to be (re)computed only for two kinds of events in $E^{(k)}$. First, for the newly arrived event $e^{(k)}$. Second, for those events in $E^{(k-1)}$ that did not happen-before $e^{(k)}$. Actually, F_b for the newly arrived event can be determined rather easily.

Lemma 8 *Given a process p_i,*

$$i \neq i_k \;\Rightarrow\; F_b^{(k)}(e^{(k)})[i] = F_b^{(k-1)}(\top_{i_k})[i] \tag{8.1}$$

$$i = i_k \;\Rightarrow\; F_b^{(k)}(e^{(k)})[i] = \min\{e^{(k)},\, F_b^{(k-1)}(\top_{i_k})[i]\} \tag{8.2}$$

Figure 5 shows the algorithm to update the slice on arrival of a new event. We now analyze the time-complexity of the algorithm. For a set of events X, let X_i denote the subset of those events that occurred on process p_i. Note that for an event e in $E^{(k-1)}$, if $e^{(k)} \rightarrow e$ then $e \in \top$; otherwise, when e was incorporated into the slice, it was not "enabled"—a contradiction. As a result, events in $E^{(k-1)}$ that did not happen-before $e^{(k)}$ consists of either those events that are concurrent with $e^{(k)}$ or the final events. Now, let $C^{(k)}$ contain those events from $E^{(k)}$ that are concurrent with $e^{(k)}$. It can be verified that, given processes p_i and p_x, the

Input: (1) a computation $G^{(k)} = \langle E^{(k)}, \rightarrow \rangle$, (2) a predicate b,
(3) for each event $e \in E^{(k-1)}$, $F_b(e)$ currently set to $F_b^{(k-1)}(e)$, and
(4) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: for each event $e \in E^{(k)}$, $F_b(e)$ now set to $F_b^{(k)}(e)$

1 $F_b(e^{(k)}) := F_b(\top_{i_k})$; // *compute F_b for the new event*
2 for each event e in $E^{(k)}$ do
3   if $F_b(e)[i_k] = \top_{i_k}$ then $F_b(e)[i_k] := e^{(k)}$; endif; // *is e a critical event?*
endfor;

4 for each process p_x do
5   for each process p_i do
6   let e be the earliest event on p_x such that $e \not\rightarrow e^{(k)}$;
7         $f := F_b(pred(e))[i]$;
8         $done :=$ false;
9   while not($done$) do
10             $f := \max\{f, F_b(e)[i]\}$; // *$F_b$ is order-preserving and Lemma 6*
11   while $(f \neq \top_i)$ and CONTC$(\widehat{G}^{(k)}[e, f], b)$ do
12                 $f := succ(f)$; // *advance to the next event on p_i*
            endwhile;
13             $F_b(e)[i] := f$;
14   if $e = \top_x$ then $done :=$ true;
15   else $e := succ(e)$; // *advance to the next event on p_x*
            endif;
        endwhile;
    endfor;
endfor;

Figure 5: An on-line algorithm to update $F_b(e)$ for all events e on arrival of a new event.

12a   if $f \rightarrow e^{(k)}$ then
12b   set f to the earliest event on p_i such that $f \not\rightarrow e^{(k)}$;
12c   else $f := succ(f)$;
    endif;

Figure 6: Improving the time-complexity of the algorithm in Figure 5.

number of times an instance of CONTC is invoked at line 11 is given by $O(|E_i^{(k)}| + |C_x^{(k)}|)$. This is because between two consecutive invocations of CONTC, either e or f advances to its next event. Further, whereas e, if different from $\top_x$, is constrained to be concurrent with $e^{(k)}$, there is no such constraint on f. Summing over all possible values for i and x, CONTC is invoked $O(n|E^{(k)}|)$ times. This gives us a time-complexity of $O(n|E|T)$ for updating the slice, which is same as that of computing the slice from scratch. (Note that the earliest event on a process that did not happen-before $e^{(k)}$—at line 6—can be determined in $O(1)$ time using the Fidge/Mattern's vector timestamp.)

To reduce the time complexity further, we proceed as follows. Suppose, at line 11,

$\mathsf{CONTC}(\widehat{G}^{(k)}[e,f],b)$ evaluates to true and $f \rightarrow e^{(k)}$. It can be verified that, in that case, $\mathsf{CONTC}(\widehat{G}^{(k)}[e,g],b)$ will also evaluate to true for all events g such that $g \rightarrow e^{(k)}$. Formally,

Lemma 9 *Consider an event $e \in E^{(k-1)}$ and a process p_i. Further, let f be an event on p_i with $f \rightarrow e^{(k)}$ such that $F_b^{(k-1)}(e)[i] \overset{P}{\rightarrow} f$. Then,*

$$\left(\begin{array}{c}(\mathsf{CONTC}(\widehat{G}^{(k)}[e,f],b) \textit{ evaluates to true}) \bigwedge \\ (g \rightarrow e^{(k)})\end{array}\right) \Rightarrow \mathsf{CONTC}(\widehat{G}^{(k)}[e,g],b) \textit{ evaluates to true}$$

Therefore, when the condition of the while loop at line 11 evaluates to true and $f \rightarrow e^{(k)}$, rather than advancing f to $succ(f)$, we can advance f directly to the earliest event on p_i that did not happen-before $e^{(k)}$. This reduces the number of times an instance of CONTC is evaluated to $O(|C_i^{(k)}| + |C_x^{(k)}| + 1)$. The modification is described in Figure 6. Now, summing over all possible values for i and x, when $e^{(k)}$ arrives, CONTC needs to be invoked $O(n|C^{(k)}| + n^2)$ times to update the slice. Next, summing over the arrival of $|E|$ events, the total number of times CONTC is invoked is given by $O(n|C| + n^2|E|)$, where C is the set of concurrent pairs of events in the computation. Assuming that the time-complexity of solving CONTC increases with the number of events, the overall time-complexity is given by $O(n|C|T + n^2|E|T)$, where $O(T)$ is the worst-case time-complexity of solving CONTC for a computation consisting of $|E|$ events. Note that the time-complexity of executing lines 1-3, over $|E|$ events, is given by $O(|E|^2)$, which can be ignored assuming that $T = \Omega(|E|)$. Finally, the amortized time-complexity for updating the slice *once*—on arrival of an event—is given by $O(n(c+n)T)$, where $c = |C|/|E|$ denotes the average concurrency in the computation. Formally,

Theorem 10 *The time-complexity of the algorithm to update the slice on arrival of a new event, described in Figure 5 and Figure 6, amortized over $|E|$ events, is $O(n(c+n)T)$, where n is the number of processes, c is the average concurrency in the computation and $O(T)$ is the worst-case time-complexity of solving CONTC for a computation consisting of $|E|$ events.*

In case c is low, say $O(n)$, the on-line algorithm has an amortized time-complexity of $O(n^2T)$. In this case, therefore, rather than computing the slice from scratch whenever an event arrives, it is much faster to *update* it using the incremental algorithm. The (on-line) algorithm in this section only assumes that the predicate can be detected efficiently; no other assumption is made about the structure of the predicate. For a special class of predicates, however, namely regular predicates [9], we have developed a much faster $O(n^2)$ amortized time-complexity algorithm to compute the slice in an on-line manner.

6 Conclusion

In this paper, we show the equivalence of two problems in distributed systems, namely predicate detection and computation slicing. We also give an efficient incremental algorithm to compute the slice for a predicate that can be detected efficiently.

References

[1] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, August 2001.

[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 (TOPLAS)*, 17(1):157–179, 1995.

[4] C. Chase and V. K. Garg. On Techniques and their Limitations for the Global Predicate Detection Problem. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[5] 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.

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

[7] V. K. Garg. Algorithmic Combinatorics based on Slicing Posets. In *Proceedings of the 22nd Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Kanpur, India, December 2002.

[8] V. K. Garg. *Elements of Distributed Computing*. John Wiley and Sons, Incorporated, New York, NY, 2002.

[9] 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.

[10] A. D. Kshemkalyani. A Framework for Viewing Atomic Events in Distributed Computations. *Theoretical Computer Science*, 196(1-2):45–70, April 1998.

[11] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. POET: Target-System Independent Visualizations of Complex Distributed-Applications Executions. *The Computer Journal*, 40(8), 1997.

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

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

[14] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[15] N. Mittal and V. K. Garg. Software Fault Tolerance of Distributed Programs using Computation Slicing. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 105–113, Providence, Rhode Island, May 2003.

[16] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in the Happened-Before Model. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Florida, April 2002.

[17] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89. Elsevier, 2003.

[18] 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 (LNCS)*, pages 318–332, France, September 1995.

[19] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 264–279. Springer-Verlag, July 2000.

A Skeletal Representation of a Slice

In general, there can be multiple directed graphs with the same set of consistent cuts. Therefore more than one graph may constitute a valid representation of the given slice. The following lemma states that all such graphs are in fact related.

Lemma A-1 ([14]) *Consider directed graphs G and H on the same set of vertices. Then,*

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

which in turn implies that:

Lemma A-2 ([14]) *Consider directed graphs G and H on the same set of vertices. Then,*

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

In other words, two directed graphs G and H, on identical sets of vertices, are *cut-equivalent* (that is, $\mathcal{C}(G) = \mathcal{C}(H)$) if and only if they are *path-equivalent* (that is, $\mathcal{P}(G) = \mathcal{P}(H)$). We consider a special directed graph to capture a slice, called the *skeletal representation* of a slice [14]. Let $F_b(e)$ be a vector of events, where the i^{th} entry in the vector denotes the earliest event on process p_i reachable from e in the slice. The skeletal representation of a slice has the following edges:

1. for each event $e \notin \top$, 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 $F_b(e)[i]$.

An advantage of the skeletal representation is that it has $O(|E|)$ vertices and only $O(n|E|)$ edges, where n is the number of processes and E is the set of events, and hence generally leads to more efficient algorithms involving slices.

B Omitted Proofs

Proof for Lemma 1: We have,

$$
\begin{aligned}
& \text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G, b) \\
\equiv\ & \{ \text{ definition of } \mathsf{slice}(G, b) \text{ } \} \\
& (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G, b)) \wedge \Big(\mathcal{C}(\mathsf{slice}(G, b)) \subseteq \mathcal{C}(G)\Big) \\
\equiv\ & \{ \text{ from Lemma A-1, } \mathcal{C}(\mathsf{slice}(G, b)) \subseteq \mathcal{C}(G) \equiv \mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G, b)) \text{ } \} \\
& (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G, b)) \wedge \Big(\mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G, b))\Big) \\
\equiv\ & \{ \text{ definition of } G[e, f] \text{ } \} \\
& \mathcal{P}(G[e, f]) \subseteq \mathcal{P}(\mathsf{slice}(G, b)) \\
\equiv\ & \{ \text{ from Lemma A-1 } \} \\
& \mathcal{C}(\mathsf{slice}(G, b)) \subseteq \mathcal{C}(G[e, f]) \\
\equiv\ & \{ \text{ } \mathcal{C}(\mathsf{slice}(G, b)) \text{ contains all consistent cuts of } \mathcal{C}(G) \text{ satisfying } b \text{ } \} \\
& \text{no consistent cut in } \ \mathcal{C}(G) \setminus \mathcal{C}(G[e, f]) \ \text{ satisfies } b
\end{aligned}
$$

This establishes the lemma. □

Proof for Lemma 2: Consider a *non-trivial* consistent cut C in $\mathcal{C}(G)$. It suffices to show that $C \in \mathcal{C}(\widehat{G}[e, f]) \equiv C \notin \mathcal{C}(G[e, f])$.

(⇒) We need to prove that $C \in \mathcal{C}(\widehat{G}[e, f]) \Rightarrow C \notin \mathcal{C}(G[e, f])$. Intuitively, it means that $G[e, f]$ and $\widehat{G}[e, f]$ do not have any common non-trivial consistent cut. Equivalently, $\mathcal{C}(G[e, f]) \cap \mathcal{C}(\widehat{G}[e, f]) = \{\emptyset, E\}$. We have,

$$
\begin{aligned}
& C \in \mathcal{C}(\widehat{G}[e, f]) \\
\Rightarrow\ & \{ \text{ definition of } \widehat{G}[e, f] \text{ } \} \\
& (f \in C) \wedge (e \notin C) \\
\Rightarrow\ & \{ \text{ definition of } G[e, f] \text{ } \} \\
& C \notin \mathcal{C}(G[e, f])
\end{aligned}
$$

(⇐) We need to prove that $C \notin \mathcal{C}(G[e, f]) \Rightarrow C \in \mathcal{C}(\widehat{G}[e, f])$. Intuitively, it means that every non-trivial consistent cut of G is either a consistent cut of $G[e, f]$ or a consistent cut of $\widehat{G}[e, f]$. Equivalently, $\mathcal{C}(G[e, f]) \cup \mathcal{C}(\widehat{G}[e, f]) = \mathcal{C}(G)$. The proof consists of two steps. In the first step, we show that if C is not a consistent cut of $G[e, f]$, then it is the case that $e \notin C$ and $f \in C$. Assume that $C \notin \mathcal{C}(G[e, f])$. Therefore there exist events u and v such that there is a path from u to v in $G[e, f]$, $u \notin C$ and $v \in C$. Since C is a consistent cut of G, there is no path from u to v in G. That is, $(u, v) \in \mathcal{P}(G[e, f])$ but $(u, v) \notin \mathcal{P}(G)$.

Thus every path from u to v in $G[e,f]$ contains the edge (e,f)—possibly more than once. This implies that $(u,e) \in \mathcal{P}(G)$ and $(f,v) \in \mathcal{P}(G)$. Since $(u,e) \in \mathcal{P}(G)$ and $u \notin C$, $e \notin C$. Similarly, since $(f,v) \in \mathcal{P}(G)$ and $v \in C$, $f \in C$. Therefore $e \notin C$ and $f \in C$.

In the second step, we show that if C is not a consistent cut of $\widehat{G}[e,f]$, then it is the case that $(e \in C) \vee (f \notin C)$. Assume that $C \notin \mathcal{C}(\widehat{G}[e,f])$. Hence there exist events x and y such that there is a path from x to y in $\widehat{G}[e,f]$, $x \notin C$ and $y \in C$. Since C is a consistent cut of G, there is no path from x to y in G. That is, $(x,y) \in \mathcal{P}(\widehat{G}[e,f])$ but $(x,y) \notin \mathcal{P}(G)$. Thus every path from x to y in $\widehat{G}[e,f]$ involves the edge $(f, \bot_1)$ or the edge $(\top_1, e)$ or both. Consider any path from x to y in $\widehat{G}[e,f]$. In case the first edge to appear in the path—out of the two edges—is $(f, \bot_1)$, there is a path from x to f in G which implies that $f \notin C$. Also, if the last edge to appear in the path—out of the two edges—is $(\top_1, e)$, then there is path from e to y in G which implies that $e \in C$. If neither of the case holds, then the first edge to appear in the path is $(\top_1, e)$ and the last edge to appear in the path is $(f, \bot_1)$. It can be verified that, in this case, there is a path from e to f in G implying that $f \in C \Rightarrow e \in C$, or, equivalently, $(e \in C) \vee (f \notin C)$. In any case, $(e \in C) \vee (f \notin C)$ holds—a logical negation of what we obtained in the first step.

Combining the two steps, it can be inferred that it cannot be the case that C is neither a consistent cut of $G[e,f]$ nor a consistent cut of $\widehat{G}[e,f]$. This establishes the lemma. □

Proof for Theorem 4: The initialization at line 1 requires $O(|E|^2)$ time, where E is the set of events, because G has $|E|$ vertices and therefore $O(|E|^2)$ edges. The for loop at line 2 executes $|E|^2$ times. Each iteration of the for loop requires solving an instance of CONTC. The construction of the particular instance of CONTC involves adding two edges to G, and therefore can be done in $O(1)$ time. Depending on the result of the if statement at line 3, an edge may be required to be added to K at line 4, which can be done in $O(1)$ time. At the end of the iteration, the two edges that were added to G have to be deleted. The deletion can be accomplished in $O(1)$ time by maintaining pointers to the two edges if using adjacency list representation. The overall time-complexity of the for loop is $O(|E|^2 T)$, which is also the time-complexity of the algorithm. □

Proof for Theorem 5: Note that the while loop at line 4 terminates in at most $|E_i| + |E_x|$ iterations, where E_i and E_x denote the set of events on processes p_i and p_x, respectively. This is because between two consecutive iterations of the while loop, either e or f advances to its next event. Also, the directed graph $\widehat{G}[e,f]$ when $f = \top_i$ has an edge from the final event $\top_i$ to the initial event $\bot_1$ implying that $\widehat{G}[e, \top_i]$ has no non-trivial consistent cut. Therefore $\mathsf{CONTC}(\widehat{G}[e,f], b)$ when $f = \top_i$ will, trivially, evaluate to false. This gives a time-complexity of $O((|E_i| + |E_x|)\, T)$ for the inner for loop at line 3. Hence, summing over all possible values for i, the time-complexity of the outer for loop at line 1 is $O((|E| + n|E_x|)\, T)$. This implies that the overall time-complexity of computing $F_b(e)$ for all events e on *all* processes is $O(n|E|T)$. □

Proof for Lemma 6: First, consider an event $f \in E^{(k-1)}$ on process p_i. We have,

$$
\begin{array}{cl}
& f \xrightarrow{P} F_b^{(k-1)}(e)[i] \\
\equiv & \{\ F_b^{(k-1)}(e)[i] \text{ is the earliest event on } p_i \text{ reachable from } e \text{ in } \mathsf{slice}(G^{(k-1)}, b)\ \} \\
& (e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k-1)}, b)) \\
\equiv & \{\ f \in E^{(k-1)} \text{ and using Theorem 3 }\} \\
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k-1)}[e, f] : C \text{ satisfies } b \rangle \\
\equiv & \{\ \text{definition of } \widehat{G}^{(k-1)}[e, f]\ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\Rightarrow & \{\ \text{every non-trivial consistent cut of } G^{(k-1)} \text{ is a consistent cut of } G^{(k)}\ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\equiv & \{\ \text{definition of } \widehat{G}^{(k)}[e, f]\ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k)}[e, f] : C \text{ satisfies } b \rangle \\
\equiv & \{\ \text{using Theorem 3 }\} \\
& (e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k)}, b)) \\
\equiv & \{\ F_b^{(k)}(e)[i] \text{ is the earliest event on } p_i \text{ reachable from } e \text{ in } \mathsf{slice}(G^{(k)}, b)\ \} \\
& f \xrightarrow{P} F_b^{(k)}(e)[i]
\end{array}
$$

In other words, whenever $f \in E^{(k-1)}$, $f \xrightarrow{P} F_b^{(k-1)}(e)[i]$ implies that $f \xrightarrow{P} F_b^{(k)}(e)[i]$. Now, we prove the two implications.

(6.1) Consider an event $f \in E^{(k)}$ on process p_i. Suppose $f \xrightarrow{P} F_b^{(k-1)}(e)[i]$. In case i is different from i_k, f is different from $e^{(k)}$ and therefore $f \in E^{(k-1)}$. On the other hand, if $e \notin \mathit{critical}(k)$, then $F_b^{(k-1)}(e)[i] \xrightarrow{P} e^{(k)}$ and therefore $f \in E^{(k-1)}$. In either case, $f \in E^{(k-1)}$. Using the above result, $f \xrightarrow{P} F_b^{(k)}(e)[i]$. In other words, whenever $f \in E^{(k)}$, $f \xrightarrow{P} F_b^{(k-1)}(e)[i]$ implies that $f \xrightarrow{P} F_b^{(k)}(e)[i]$. This, in turn, means that $F_b^{(k-1)}(e)[i] \xrightarrow[-]{P} F_b^{(k)}(e)[i]$.

(6.2) Consider an event $f \in E^{(k)}$ on process p_i, where $i = i_k$. Suppose $f \xrightarrow{P} e^{(k)}$. Clearly, $f \in E^{(k-1)}$. Further, in case e is a critical event, $f \xrightarrow{P} F_b^{(k-1)}(e)[i]$. Using the above result, $f \xrightarrow{P} F_b^{(k)}(e)[i]$. In other words, whenever $f \in E^{(k)}$, $f \xrightarrow{P} e^{(k)}$ implies that $f \xrightarrow{P} F_b^{(k)}(e)[i]$. This, in turn, means that $e^{(k)} \xrightarrow[-]{P} F_b^{(k)}(e)[i]$. □

Proof for Lemma 7: *(7.1)* From Lemma 6, $F_b^{(k-1)}(e)[i] \xrightarrow[-]{P} F_b^{(k)}(e)[i]$. Assume, on the contrary, that $F_b^{(k-1)}(e)[i] \xrightarrow{P} F_b^{(k)}(e)[i]$. For convenience, let $f = F_b^{(k-1)}(e)[i]$. We have,

$$
\begin{array}{cl}
& f \xrightarrow{P} F_b^{(k)}(e)[i] \\
\equiv & \{\ F_b^{(k)}(e)[i] \text{ is the earliest event on } p_i \text{ reachable from } e \text{ in } \mathsf{slice}(G^{(k)}, b)\ \} \\
& (e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k)}, b)) \\
\equiv & \{\ \text{using Theorem 3 }\}
\end{array}
$$

$$
\begin{aligned}
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k)}[e,f] : C \text{ satisfies } b \rangle \\
\equiv\; & \{ \text{ definition of } \widehat{G}^{(k)}[e,f] \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\Rightarrow\; & \{ \ e \rightarrow e^{(k)} \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \wedge (e^{(k)} \notin C) \rangle \\
\equiv\; & \{ \text{ a consistent cut of } G^{(k)} \text{ that does not contain } e^{(k)} \text{ is a consistent cut of } G^{(k-1)} \ \} \\
& \langle \exists C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad (e^{(k)} \notin C) \rangle \\
\equiv\; & \{ \ e^{(k)} \text{ is not an event in } G^{(k-1)} \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\equiv\; & \{ \text{ definition of } \widehat{G}^{(k-1)}[e,f] \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k-1)}[e,f] : C \text{ satisfies } b \rangle \\
\equiv\; & \{ \text{ using Theorem 3 } \} \\
& (e,f) \notin \mathcal{P}(\mathsf{slice}(G^{(k-1)}, b)) \\
\Rightarrow\; & \{ \ f \text{ is } F_b^{(k-1)}(e)[i] \ \} \\
& \text{a contradiction}
\end{aligned}
$$

Therefore $F_b^{(k-1)}(e)[i] = F_b^{(k)}(e)[i]$.

(7.2) Note that $\mathcal{C}(\mathsf{slice}(G^{(k)}, b)) \subseteq \mathcal{C}(G^{(k)})$. In case $e \rightarrow e^{(k)}$, there is a path from e to $e^{(k)}$ in $G^{(k)}$. Thus, from Lemma A-1, there is a path from e to $e^{(k)}$ in $\mathsf{slice}(G^{(k)}, b)$ as well. Consequently, $F_b^{(k)}(e)[i] \xrightarrow{P} e^{(k)}$. From Lemma 6, $F_b^{(k)}(e)[i]$ is either $e^{(k)}$ or $\top_i$. This, in turn, implies that $F_b^{(k)}(e)[i]$ is $e^{(k)}$. □

Proof for Lemma 8: Consider an event $f \in E^{(k)}$ with $f \neq e^{(k)}$. We have,

$$
\begin{aligned}
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k)}[e^{(k)},f] : C \text{ satisfies } b \rangle \\
\equiv\; & \{ \text{ definition of } \widehat{G}^{(k)}[e^{(k)},f] \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k)} : (f \in C) \wedge (e^{(k)} \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\equiv\; & \{ \text{ a consistent cut of } G^{(k)} \text{ that does not contain } e^{(k)} \text{ is a consistent cut of } G^{(k-1)} \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (e^{(k)} \notin C) \wedge (C \text{ satisfies } b) \rangle \\
\equiv\; & \{ \ e^{(k)} \text{ is not an event in } G^{(k-1)} \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (C \text{ satisfies } b) \rangle \\
\equiv\; & \{ \text{ definition of } \widehat{G}^{(k-1)}[\top_{i_k}, f] \ \} \\
& \langle \exists\, C : C \text{ is a consistent cut of } \widehat{G}^{(k-1)}[\top_{i_k}, f] : C \text{ satisfies } b \rangle
\end{aligned}
$$

(8.1) Clearly, using the above result and Theorem 3, $F_b^{(k)}(e^{(k)}) = F_b^{(k-1)}(\top_{i_k})[i]$.

(8.2) In case $F_b^{(k-1)}(\top_{i_k})[i]$ is different from $\top_{i_k}$, using the above result and Theorem 3, we get that $F_b^{(k)}(e^{(k)}) = F_b^{(k-1)}(\top_{i_k})[i]$. On the other hand, if $F_b^{(k-1)}(\top_{i_k})[i] = \top_{i_k}$, then $\mathsf{CONTC}(\widehat{G}^{(k)}[e^{(k)}, e^{(k)}], b)$ evaluates to false and therefore $F_b^{(k)}(e^{(k)})[i] = e^{(k)}$. $\square$

Proof for Lemma 9: Since $f \overset{P}{\to} e^{(k)}$, $f \in E^{(k-1)}$. Further, $F_b^{(k-1)}(e)[i] \overset{P}{\underline{\to}} f$. From Theorem 3, $\mathsf{CONTC}(\widehat{G}^{(k-1)}[e, f], b)$ evaluates to false. Equivalently, there is no consistent cut of $\widehat{G}^{(k-1)}[e, f]$ that satisfies b. However, $\mathsf{CONTC}(\widehat{G}^{(k)}[e, f], b)$ evaluates to true. This implies that there exists a consistent cut of $\widehat{G}^{(k)}[e, f]$, say C, that satisfies b. In other words, C is a consistent cut of $G^{(k)}$, $f \in C$, $e \notin C$ and C satisfies b. Clearly, $e^{(k)} \in C$; otherwise, C is a consistent cut of $\widehat{G}^{(k-1)}[e, f]$ that satisfies b—a contradiction. Since $g \to e^{(k)}$, C also contains g. Therefore $\mathsf{CONTC}(\widehat{G}^{(k)}[e, g], b)$ also evaluates to true. $\square$