Solving Computation Slicing Using Predicate Detection

Neeraj Mittal, *Member*, *IEEE Computer Society*, Alper Sen, *Member*, *IEEE*, and Vijay K. Garg, *Fellow*, *IEEE*

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 subcomputation (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 actually 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 $O(n|E|T)$, where n is the number of processes, E is the set of events, and $O(T)$ is the time complexity of the detection algorithm. We discuss how the "equivalence" result of this paper can be utilized to derive a faster algorithm for solving the *general* predicate detection problem in many cases. Slicing algorithms described in our earlier papers are all offline in nature. In this paper, we also present two online algorithms for computing the slice. The first algorithm can be used to compute the slice for a general predicate. Its amortized time complexity is $O(n(c+n)T)$ per event, where c is the *average concurrency* in the computation and $O(T)$ is the time complexity of the detection algorithm. The second algorithm can be used to compute the slice for a *regular* predicate. Its amortized time complexity is only $O(n^2)$ per event.

Index Terms—Program trace analysis, predicate detection, computation slicing, testing and debugging.

1 Introduction

Writing correct distributed programs is a nontrivial task. Not surprisingly, distributed systems are particularly vulnerable to software faults. Testing and debugging is an effective way of improving the dependability of software prior to its deployment. Software bugs that persist after extensive testing and debugging have to be tolerated at runtime to ensure that the system continues to operate properly. Detecting a fault (e.g., violation of a safety property such as mutual exclusion) in the execution of a distributed system 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 the 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 (distributed program execution) that satisfies a given global predicate (this problem is also referred to as detecting a predicate under the *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 simultaneously.

Detecting a global predicate in a distributed computation is a hard problem in general [2], [3], [4]. The reason is the combinatorial explosion in the number of possible consistent cuts. Finding a consistent cut that satisfies the given predicate may therefore require looking at a large number of consistent cuts. In fact, we prove in [4] that detecting a predicate in 2-conjunctive normal form (2-CNF), even when no two clauses contain variables from the same process, is an NP-complete problem in general. 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 each x_i is a Boolean variable on process p_i. Many algorithms for predicate detection exploit the structure of the predicate to detect it efficiently in a given computation (hereafter, we say that an algorithm is efficient if its (worst case) time complexity is polynomial in input size). Polynomial-time detection algorithms have been developed for several useful classes of predicates, such as conjunctive, linear, and semilinear predicates [3] and relational predicates [5].

We introduced the notion of *computation slice* in [6]. Intuitively, slice is a concise representation of consistent cuts satisfying a certain condition. The slice of a computation with respect to a predicate is a subcomputation such that 1) it contains all consistent cuts of the computation for which the predicate evaluates to true and 2) of all the subcomputations that satisfy 1), it has the least number of consistent cuts. Suppose the number of consistent cuts of the slice is

- *N. Mittal is with the Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083. E-mail: neerajm@utdallas.edu.*
- *A. Sen is with Freescale Semiconductor Inc., Austin, TX 78729. E-mail: alper.sen@freescale.com.*
- *V.K. Garg is with the Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712. E-mail: garg@ece.utexas.edu.*

Manuscript received 28 Sept. 2005; revised 18 Oct. 2006; accepted 24 Jan. 2007; published online 9 Feb. 2007.
Recommended for acceptance by R. Eigenmann.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0416-0905.
Digital Object Identifier no. 10.1109/TPDS.2007.1077.

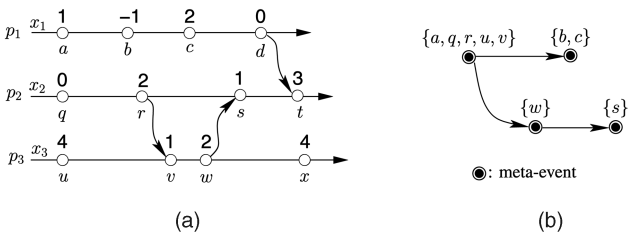

Fig. 1. (a) A computation and (b) its slice with respect to $(x_1 \geq 1) \wedge (x_3 \leq 3)$.

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 demonstrate this with the help of an example.

Suppose that we want to detect the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3) \wedge (x_1 * x_2 + x_3 < 5)$ in the computation shown in Fig. 1a. 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, 30 in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute the slice of the computation with respect to $(x_1 \geq 1) \wedge (x_3 \leq 3)$ as follows. Immediately after executing b, the value of x_1 becomes -1, which does not satisfy $x_1 \geq 1$. To reach a consistent cut satisfying $x_1 \geq 1$, 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 Fig. 1b. It is modeled by a partial order on a set of metaevents, and each *metaevent* consists of one or more "primitive" events. A consistent cut of the slice either contains all the events in a metaevent or none of them (intuitively, any consistent cut of the computation that contains only a partial set of events in a metaevent is of no relevance to us). Moreover, a metaevent "belongs" to a consistent cut only if all its incoming neighbors 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,

$$\{a, q, r, u, v\}, \{a, q, r, u, v, b, c\}, \{a, q, r, u, v, w\},$$
$$\{a, q, r, u, v, b, c, w\}, \{a, q, r, u, v, w, s\},$$
$$\text{and } \{a, q, r, u, v, b, c, w, s\}.$$

The slice has much fewer consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

The notion of a computation slice is similar to the notion of a *program slice*, which was introduced by Weiser in [7] to facilitate program debugging. A program slice consists of all those statements of a program that may potentially affect the value of *certain variables* at some *point of interest*. Program slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance [8], [9]. In spite of the apparent similarities, the two notions of slicing are quite different from each other. First, program slicing is applicable to both sequential and distributed programs. Computation slicing is applicable to (traces of) distributed programs only. Second, program slicing involves conducting data-flow analysis of the program [10], whereas computation slicing involves computing join-irreducible elements of a distributive lattice [6]. Third, program slicing is traditionally variable-based (statements that affect the value of a variable), whereas computation slicing is predicate-based (consistent cuts that satisfy a predicate). In fact, the two techniques for testing and debugging are orthogonal to each other and can be used in conjunction. For instance, to compute a program slice, the programmer needs to determine a point at which the program behaves in a faulty manner (that is, there is a mismatch between actual and expected values). For a sequential program, determining a point in an execution where a fault has occurred is a relatively easy problem. However, for a distributed program, as explained earlier, determining even whether a fault has occurred in an execution is an intractable problem in general. In this paper, we show that computation slicing can be used for reducing the state space to be analyzed to determine whether and where a fault has occurred. Recently, Li et al. [11] have extended the notion of program slicing, which is traditionally variable-based, to include *predicate-based* program slicing.

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, computing the slice for a predicate is at least as hard as detecting the predicate in the sense that the 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 show, somewhat surprisingly, that the two problems are actually equivalent by proving the converse; that is, *computing the slice for a predicate is no harder than detecting the predicate*. Specifically, 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. Using this result, we can now compute the slice efficiently for a larger class of predicates. Note that, since both predicate detection and computation slicing are NP-complete problems in general, they are already equivalent by using Cook's transformation [12]. We show that this notion of equivalence (based on Cook's transformation) is *weaker* than our notion of equivalence and cannot be used to derive efficient slicing algorithms in general.

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 speed up predicate detection? The following

example illustrates that slicing can indeed facilitate predicate detection. Suppose that we want to detect a predicate $b = b_1 \wedge b_2$ in a computation C. Further, assume that b_1 can be detected in an efficient manner, but the same does not apply for b. To detect b, without computation slicing, we are forced to search the state space of the computation C to locate a consistent cut that satisfies b, if it exists. None of the techniques that we are aware of can take advantage of the fact that b_1 can be detected efficiently when detecting b. Using computation slicing and the equivalence result in this paper, we can potentially throw away an exponential number of consistent cuts by spending only a polynomial amount of time by first computing the slice S_1 of the computation C with respect to b_1. To detect b, it is sufficient to search the state space of S_1 instead of the state space of C, resulting in an exponential speedup overall. If we can also detect b_2 efficiently, then we can further use it to our advantage as follows: We compute the slice S_2 of the computation C with respect to the predicate b_2 as well. Using the *slice composition* algorithms discussed in our earlier papers [6], [13], we can efficiently compute the slice S that contains only those consistent cuts that are common to both S_1 and S_2. Now, to detect b, clearly, it is sufficient to search the state space of the slice S, which may be much smaller than that of S_1, as well as S_2. Intuitively, by being able to compute slices efficiently for "simple" predicates (b_1 and b_2 in our example), we can potentially obtain a significant speedup when detecting more "complex" predicates (b in our example) composed from "simple" predicates using conjunction [6], disjunction [6], and temporal logic operators [13]. Thus, the equivalence result in this paper aids in the detection of "complex" predicates by *expanding the class of "simple" predicates for which the slice can be computed in polynomial time, resulting in exponential savings overall in general*. Our experimental results in [6] indicate that slicing can indeed lead to an exponential improvement over existing techniques for predicate detection in terms of time and space. Note that other techniques for reducing the time complexity [14] and/or the space complexity [15] of predicate detection are orthogonal to slicing and, as such, can be used in conjunction with it.

The algorithms described in our earlier papers [6], [13], [16] for computing a slice are all offline in nature: They assume that the entire set of events is available a priori. Although 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 online 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 while the system is executing as early as possible before it can cause any severe damage. 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 by using an offline algorithm.

In this paper, we also give two efficient algorithms for computing the slice in an incremental manner. The first algorithm can be used to compute the slice with respect to a general predicate, given an efficient (offline) algorithm to detect the predicate. Its amortized time complexity for updating the slice on the arrival of a new event is $O(n(c+n)T)$, where c is the *average concurrency* in the computation and $O(T)$ is the time complexity of the detection algorithm. We define average concurrency in a computation to be the ratio of the number of concurrent pairs of events (including reflexive pairs) to the number of events. The average concurrency c in a computation lies between 1 and $|E|$, where E denotes the set of events in the computation. In practice, however, due to interprocess communication, we expect c to be much smaller than $|E|$. The second algorithm can be used to compute the slice for a special class of predicates, namely, *regular* predicates (defined later). Its amortized time complexity for updating the slice is only $O(n^2)$ and is therefore much more efficient than the first algorithm (note that, unlike in the case of the first algorithm, there is no term due to the time complexity of the detection algorithm).

To summarize, our contributions in this paper are given as follows: First, we prove that the problem of detecting a predicate in a computation is equivalent to the problem of computing the slice of the computation with respect to the predicate. Hence, we extend the class of predicates for which we can devise efficient slicing algorithms. Second, we give two efficient algorithms for computing the slice in an online manner.

In [17], Kshemkalyani described a unifying framework for viewing a distributed computation at multiple levels of atomicity. In particular, Kshemkalyani defined an execution of a system in terms of certain *elementary* events. System executions at a coarser level of atomicity are then hierarchically composed using system executions at finer levels of atomicity by grouping multiple elementary events together into a single *compound* event. However, the system executions considered by Kshemkalyani are such that either communication events for the same message are grouped together or events on the same process are grouped together. In contrast, in a computation slice, events belonging to multiple messages and/or processes can be grouped together into a single metaevent depending on the predicate. Furthermore, our focus is on developing efficient algorithms for automatically computing the slice of a computation for a given predicate.

The paper is organized as follows: Section 2 describes the system model and notation used in this paper. In Section 3, we discuss the background on computation slicing necessary to understand the rest of the paper. We establish the equivalence of the two problems in Section 4. We also discuss its importance to solving the general predicate detection problem, which is NP-complete. The online algorithms for slicing are described in Section 5. Finally, Section 6 concludes the paper. Due to space constraints, some of the proofs have been omitted and can be found in the Appendix, which is available on the Computer Society Digital Library at http://computer.org/tpds/archives.htm.

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. We assume that the system is

reliable in the sense that processes do not fail and channels do not lose messages.

Processes change their states by executing 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 [18]. In this paper, we relax the restriction that the order of events must be a partial order. More precisely, we 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 $\mathrm{V}(G)$ and $\mathrm{E}(G)$ 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 neighbors. Formally,

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

Note that a consistent cut contains either 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 $\mathrm{V}(G)$ 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 [18]. Lamport's happened-before relation is defined as the smallest transitive relation satisfying the following properties: 1) if events e and f occur on the same process and e occurred before f in real time, then e happened before f, and 2) if events e and f correspond to send and receive events, respectively, of the same message, then e happened before f.

A distributed computation in our model can contain cycles. This is because, whereas a computation in the traditional (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 *metaevent*, and all events in a metaevent are executed atomically.

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 occur on the same process and e occured before f in real time, then we write $e \stackrel{P}{\rightarrow} f$ and let $\stackrel{P}{\rightarrow}$ 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 $\perp_i$, 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 $\perp$ 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 nontrivial 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 nontrivial consistent cut of the corresponding computation in our model and vice versa. Only nontrivial consistent cuts are of interest to us. The *frontier* of a cut C, denoted by $frontier(C)$, is defined as the set of those events in C whose successors are not in C. Formally,

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

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, then 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 the variables of a single process. In this paper, we only consider state-based nontemporal global predicates.

Example 1. Consider the computation depicted in Fig. 2a. 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 metaevent. Processes p_1, p_2, and p_3 host integer variables x, y, and z, respectively. The predicate $x \leq 1$ is local, whereas the predicate $x + y \leq z$ is not. The consistent cut $\{e_1, f_1, g_1\}$ satisfies $x + y \leq z$, but the consistent cut $\{e_1, e_2, f_1, f_2, g_1\}$ does not. □

3 Background

In this section, we briefly discuss the concepts pertaining to computation slicing, which are required to understand this paper. For a detailed description of computation slicing, the reader is referred to [6].

3.1 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 [6]). *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*

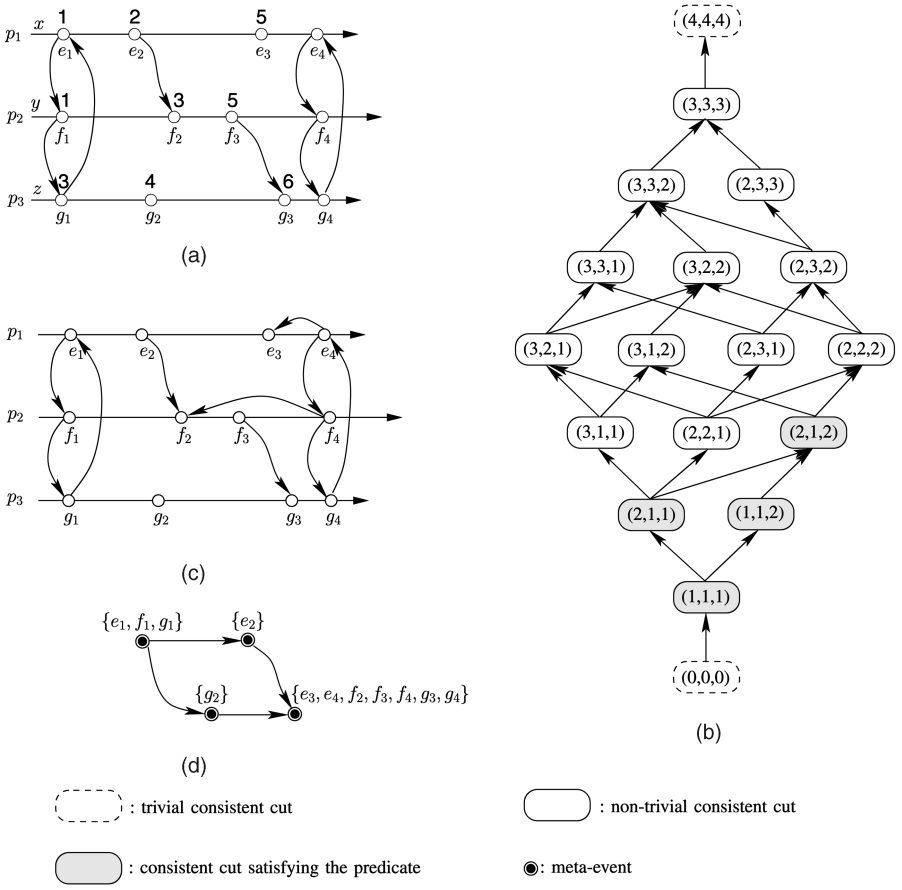

Fig. 2. (a) A computation. (b) The set of its consistent cuts. (c) The slice of the computation with respect to the predicate $x + y \leq z$ represented as a directed graph on the set of events. (d) The same slice represented as a partial order on the set of metaevents.

of the computation for which the predicate evaluates to true. Formally, given a computation $\langle E, \rightarrow \rangle$ and a predicate b,

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

We use $\text{slice}(\langle E, \rightarrow \rangle, b)$ to denote a slice of $\langle E, \rightarrow \rangle$ with respect to b. Note that $\langle E, \rightarrow \rangle = \text{slice}(\langle E, \rightarrow \rangle, \text{true})$. Therefore, a computation can be viewed as a slice. Likewise, a slice can be viewed as a computation at some level of abstraction [17]. We show in [6] that the 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 have identical metaevents.

Example 2. All consistent cuts of the computation in Fig. 2a that satisfy the predicate $x + y \leq z$ have been shaded in Fig. 2b. Every consistent cut is labeled with the number of events that have to be executed on each process to reach the cut. Furthermore, there is an edge from one consistent cut to another if the latter can be reached from the former by executing *exactly one* event. The slice of the computation with respect to the predicate $x + y \leq z$ is shown in Fig. 2c. It can be verified that every consistent cut of the computation that satisfies $x + y \leq z$ is indeed a consistent cut of the slice. The alternative representation of the slice as a partial order on a set of metaevents is shown in Fig. 2d. Basically, for every strongly connected component in a graph representation of a slice (for example, Fig. 2c), there is a metaevent in the partially ordered set (poset) representation of the slice (for example, Fig. 2d) and vice versa. □

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 nontrivial consistent cuts [6]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a nontrivial consistent cut. Efficient algorithms for computing the slice for some useful classes of predicates can be found in our earlier papers [6], [16].

3.2 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. We show in [6] that all such graphs (which have identical sets of consistent cuts) have the same transitive closure. The proof depends on the following lemma:

Lemma 1 ([6]). *Consider directed graphs G and H on the same set of vertices. Then, $\mathcal{P}(G) \subseteq \mathcal{P}(H) \equiv \mathcal{C}(G) \supseteq \mathcal{C}(H)$.*

Clearly, the lemma in turn implies the following:

Lemma 2 ([6]). *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, $\mathcal{P}(G) = \mathcal{P}(H)$). Typically, the time complexity of an algorithm involving slices depends on the size of the graph used to represent a slice: The fewer the number of edges in the graph, the more efficient the slicing algorithm becomes. Therefore, for efficiency reasons, we consider a special directed graph for capturing a slice, called the *skeletal representation* of a slice [6]. This graph 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. Let $F_b(e)$ be a vector of events, where the ith entry in the vector denotes the *earliest* event on process p_i reachable from e in the slice with respect to predicate b. The skeletal representation of a slice has the following edges:

1. For each event $e \notin \top$, there is an edge from e to $succ(e)$.
2. For each event e and process p_i, there is an edge from e to $F_b(e)[i]$.

Example 3. Consider the slice in Fig. 2c, with $b \triangleq x + y \leq z$. In the example, $F_b(e_1) = [e_1, f_1, g_1]$, $F_b(e_2) = [e_2, f_2, g_3]$, and $F_b(f_2) = [e_3, f_2, g_3]$. □

We now prove the equivalence of predicate detection and computation slicing.

4 The Two Problems

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

- **Containing Cut (CONTC) (Predicate Detection).** 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 $\text{CONTC}(G, b)$ for all G, we can derive an algorithm V for solving $\text{COMPS}(G, b)$ for all G such that the time complexity of V is within a polynomial factor of the time complexity of U and vice versa.

Note that, since both CONTC and COMPS are NP-complete problems in general, they are equivalent in the sense that, given an algorithm U to solve $\text{CONTC}(G, b)$ for all G and b, we can derive an algorithm V to solve $\text{COMPS}(G, b)$ for all G and b such that the time complexity of V is within a polynomial factor of the time complexity of U and vice versa.

It can be easily verified that our notion of equivalence is stronger than the notion of equivalence between NP-complete problems. Specifically, our notion of equivalence "fixes" the predicate b, whereas the one that exists between any two NP-complete problems does not. Therefore, given a polynomial-time algorithm for detecting b, the "standard" transformation that exists between any two NP-complete problems cannot be used in general to derive a *polynomial-time* algorithm for computing the slice for b (unless P = NP).

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.

4.1 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 is nonempty if and only if the graph contains a consistent cut that satisfies the predicate. Formally, $\text{CONTC}(G, b) \equiv \text{slice}(G, b)$ is nonempty.

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

Lemma 3. *There is a path from an event e to an event f in $\text{slice}(G, b)$ if and only if no consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e, f])$ satisfies b.*

Proof. We have

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

This establishes the lemma. □

Lemma 3 is useful, provided that it is possible to ascertain efficiently whether some consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e, f])$ satisfies b. Let $\widehat{G}[e, f]$ denote the directed graph obtained by adding an edge from f to $\perp_1$ and an edge from $\top_1$ to e. Intuitively, $\widehat{G}[e, f]$ consists of all those consistent cuts of G that contain f but not e. It suffices to show the following:

Lemma 4. $\mathcal{C}(\widehat{G}[e, f]) \setminus \{\emptyset, E\} \ = \ \mathcal{C}(G) \setminus \mathcal{C}(G[e, f])$.

Combining the two lemmas, we obtain the following:

Theorem 5. *There is a path from an event e to an event f in $\text{slice}(G, b)$ if and only if no consistent cut in $\widehat{G}[e, f]$ satisfies b; that is, $\text{CONTC}(\widehat{G}[e, f], b)$ evaluates to false.*

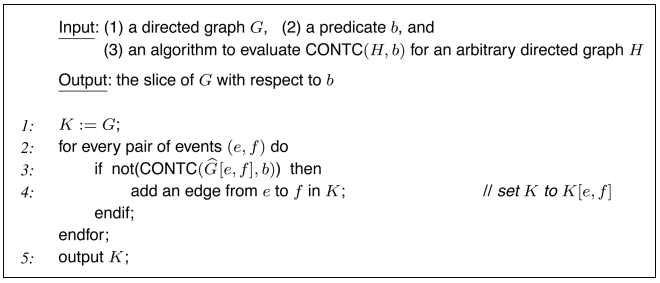

Fig. 3. An algorithm to solve COMPS by using an algorithm to solve CONTC.

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

Theorem 6. *The time complexity of the algorithm for solving* COMPS *described in Fig. 3 is $O(|E|^2T)$, where E is the set of events and $O(T)$ is the worst case time complexity of solving* CONTC.

Proof. 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 when using adjacency list representation. The overall time complexity of the for loop is therefore given by $O(|E|^2T)$, which is also the time complexity of the algorithm. □

In order to reduce the time complexity of the algorithm, we construct the skeletal representation of a slice, as defined in Section 3.2. 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 [6]. 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 Fig. 4. The next theorem establishes that the time complexity of the algorithm is $O(n|E|T)$.

Theorem 7. *The time complexity of the algorithm for computing $F_b(e)$ for all events e in Fig. 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.

Proof. Note that the while loop at line 5 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 nontrivial consistent cut. Therefore, 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 4. Hence, summing over all possible values for i, the time complexity of the outer for loop at line 2 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 $O(n|E|T)$. □

Recall that, using F_b, the graph corresponding to the skeletal representation of the slice can be easily constructed in $O(n|E|)$ time [6].

4.2 Discussion

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 [19], observer-independent predicates [20], conjunctive predicates [3], linear predicates [3], relational predicates [5], and their complements, such as costable predicates and colinear predicates [21]. In our earlier paper [6], we gave efficient algorithms for computing the slice for regular predicates, coregular predicates, linear predicates, and k-local predicates for constant k. Using the result of this paper, it is now possible to compute the slice efficiently for many more classes of predicates including stable and costable predicates, observer-independent predicates, colinear predicates, and relational predicates. For instance, an observer-independent predicate can be detected in $O(n|E|)$ time by using the algorithm

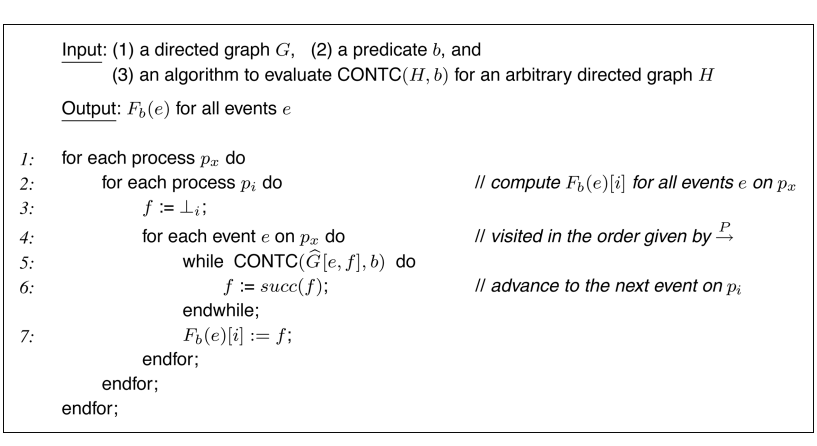

Fig. 4. An algorithm to compute $F_b(e)$ for all events e.

presented in [20]. This implies that its slice can be computed in $O(n^2|E|^2)$ time by using the algorithm given in Section 4.1.

As discussed in Section 1, by being able to compute slices efficiently for a larger class of predicates, namely, the class of all *efficiently detectable* predicates (for example, $x_1 \vee x_2$ and $x_3 \vee x_4$), we can potentially obtain a significant speedup when detecting more "complex" predicates composed from efficiently detectable predicates by using conjunction [6], disjunction [6], and temporal logic operators [13] (for example, $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$). This is achieved by first computing slices for efficiently detectable predicates and then composing these slices together to obtain a slice whose state space is likely to be much smaller than that of the computation. As a virtue of the *equivalence result* proved in Section 4.1, the first step, which involves computing slices, is guaranteed to have polynomial-time complexity. Note that the slice obtained after the two steps described above is in general not the exact slice for the predicate to be detected, but only an "approximate" slice [6]. The slice is "approximate" in the sense that it may contain consistent cuts not present in the actual slice [6]. Therefore, to detect the predicate, it is not enough to test for the emptiness of the slice, but rather, the entire state space of the slice has to be searched.

5 Online Algorithms for Computing the Slice

In this section, we describe two efficient algorithms for computing the slice in an online manner. First, we describe an algorithm to compute the slice for a general predicate by using an algorithm to *detect* the predicate. Basically, the online slicing algorithm is derived from the offline slicing algorithm described in Section 4.1. Therefore, as in the case of the offline algorithm, the online slicing algorithm is efficient whenever the detection algorithm is efficient. Next, we describe an online algorithm for computing the slice for a special class of predicates, namely, regular predicates, which is much more efficient than the first algorithm.

Before describing the algorithms, we state our assumptions and introduce 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 that 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 vector time stamp [22], [23].

Initially, the computation consists of only the fictitious (initial and final) events. Let the kth arriving event, with $k \geq 1$, be denoted by $e^{(k)}$ and let $G^{(k)}$ denote the resulting computation. Sometimes, we represent the computation more explicitly by using $\langle E^{(k)}, \rightarrow \rangle$ whenever necessary, where $E^{(k)}$ denotes the set of events and $\rightarrow$ denotes the transitive closure of the set of edges in $G^{(k)}$. Note that $\rightarrow$ on the set of nonfictitious events defines the Lamport's happened-before relation.

Clearly, a nontrivial consistent cut of $G^{(k-1)}$ is a nontrivial consistent cut of $G^{(k)}$ as well. We now present an online algorithm for computing the slice for a general predicate.

5.1 Computing the Slice for a General Predicate

Whenever a new event arrives, our online algorithm 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 $e^{(k)}$ occurs.

Definition 2 (Critical Event). *An event $e \in E^{(k-1)}$ is said to be a* critical event *(with respect to $e^{(k)}$) if $F_b^{(k-1)}(e)[i_k] = \top_{i_k}$.*

Intuitively, no nonfinal event on p_{i_k} is reachable from a critical event e in $\mathrm{slice}(G^{(k-1)}, b)$. This may change, however, on the arrival of $e^{(k)}$ because $e^{(k)}$ is an event on p_{i_k}. Let $critical(e^{(k)})$ denote the set of all events in $E^{(k-1)}$ that are critical with respect to $e^{(k)}$. We have,

Lemma 8. *Given an event $e \in E^{(k-1)}$ and a process p_i,*

$$(i \neq i_k) \vee (e \notin critical(e^{(k)})) \Rightarrow F_b^{(k-1)}(e)[i] \xrightarrow{P} F_b^{(k)}(e)[i], \quad (8.1)$$

$$(i \neq i_k) \wedge (e \in critical(e^{(k)})) \Rightarrow F_b^{(k)}(e)[i] \in \{e^{(k)}, \top_i\}. \quad (8.2)$$

Lemma 8 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 9. *Given an event $e \in E^{(k-1)}$ and a process p_i*

$$\begin{aligned}(e \rightarrow e^{(k)}) \wedge \Big((i \neq i_k) \vee (e \notin critical(e^{(k)}))\Big) \\ \Rightarrow F_b^{(k-1)}(e)[i] = F_b^{(k)}(e)[i],\end{aligned} \quad (9.1)$$

$$\begin{aligned}(e \rightarrow e^{(k)}) \wedge \Big((i = i_k) \wedge (e \in critical(e^{(k)}))\Big) \\ \Rightarrow F_b^{(k)}(e)[i] = e^{(k)}.\end{aligned} \quad (9.2)$$

Lemma 9 implies that F_b needs to be (re)computed only for two kinds of events in $E^{(k)}$: first, for the newly arrived event $e^{(k)}$, and 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. We have,

Lemma 10. *Given a process p_i,*

$$i \neq i_k \quad \Rightarrow \quad F_b^{(k)}(e^{(k)})[i] = F_b^{(k-1)}(\top_{i_k})[i], \quad (10.1)$$

$$i = i_k \quad \Rightarrow \quad F_b^{(k)}(e^{(k)})[i] = \min\{e^{(k)}, F_b^{(k-1)}(\top_{i_k})[i]\}. \quad (10.2)$$

```
Input: (1) a computation G^(k) = ⟨E^(k), →⟩,  (2) a predicate b,
       (3) for each event e ∈ 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 ∈ E^(k), F_b(e) now set to F_b^(k)(e)

// compute F_b for the new event using Lemma 10
1:  F_b(e^(k)) := F_b(⊤_{i_k});
2:  F_b(e^(k))[i_k] := min{e^(k), F_b(⊤_{i_k})[i_k]};
3:  for each event e in E^(k) do
        // if e is a critical event, then update the i_k^th entry of F_b(e) using Lemma 8
4:      if F_b(e)[i_k] = ⊤_{i_k} then F_b(e)[i_k] := e^(k); endif;
    endfor;

5:  for each process p_x do
6:      for each process p_i do
7:          f := ⊥_i;
8:          let s be the earliest event on p_x such that s ↛ e^(k);
9:          for each event e on p_x starting from s do
                // F_b is order-preserving and Lemma 8
10:             f := max{f, F_b(e)[i]};
11:             while (f ≠ ⊤_i) and CONTC(Ĝ^(k)[e, f], b) do
                    // advance to the next event on p_i
12:                 f := succ(f);
                endwhile;
13:             F_b(e)[i] := f;
            endfor;
        endfor;
    endfor;
```

Fig. 5. An online algorithm to update $F_b(e)$ for all events e on the arrival of a new event.

Fig. 5 shows the algorithm to update the slice on the arrival of a new event. Our online slicing algorithm is basically derived from the offline slicing algorithm described in Fig. 4, with the following differences: First, lines 2-4 in the online version are not present in the offline version. Second, lines 8-10 in the online version replace line 4 in the offline version. In lines 1-2, we use Lemma 10 to compute the value of F_b for the new event $e^{(k)}$. In lines 3-4, we use Lemma 8 to update the i_kth entry of F_b for all events critical with respect to $e^{(k)}$. Intuitively, after executing lines 2-4 in the online algorithm, the following holds for all events $e \in E^{(k)}$:

$$\langle \forall i \ : \ 1 \leq i \leq n \ : \ F_b(e)[i] \xrightarrow{P} F_b^{(k)}(e)[i] \rangle.$$

In lines 8-9, we use Lemma 9 to start updating F_b for only those events on process p_x that did not happen before $e^{(k)}$ instead of unnecessarily examining all events on p_x. Finally, in line 10, for computing $F_b^{(k)}(e)[i]$, we use the order-preserving property of F_b and Lemma 8 to advance f to the latter of $F_b^{(k)}(pred(e))[i]$ and $F_b^{(k-1)}(e)[i]$.

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," which is 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 number of times that 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. Furthermore, 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 the same as that of computing the slice from scratch (note that the earliest event on a process that did not happen before $e^{(k)}$, that is, at line 8, can be determined in $O(1)$ time by using the Fidge/Mattern vector time stamp [22], [23]).

```
12a:    if f → e^(k) then
12b:        set f to the earliest event on p_i such that f ↛ e^(k);
12c:    else f := succ(f);
        endif;
```

Fig. 6. Improving the time complexity of the algorithm in Fig. 5.

To reduce the time complexity further, we proceed as follows: Suppose that, at line 11, $\text{CONTC}(\widehat{G}^{(k)}[e, f], b)$ evaluates to true and $f \rightarrow e^{(k)}$. It can be verified that, in that case, $\text{CONTC}(\widehat{G}^{(k)}[e, g], b)$ will also evaluate to true for all events g such that $g \rightarrow e^{(k)}$. Formally,

Lemma 11. *Consider an event $e \in E^{(k-1)}$ and a process p_i. Furthermore, let f be an event on p_i, with $f \rightarrow e^{(k)}$, such that $F_b^{(k-1)}(e)[i] \xrightarrow{P} f$. Then,*

$$(\text{CONTC}(\widehat{G}^{(k)}[e, f], b) \text{ evaluates to true}) \bigwedge (g \rightarrow e^{(k)})$$
$$\Rightarrow \text{CONTC}(\widehat{G}^{(k)}[e, g], b) \text{ evaluates to true.}$$

Proof. Since $f \rightarrow e^{(k)}$, $f \in E^{(k-1)}$. Furthermore,

$$F_b^{(k-1)}(e)[i] \xrightarrow{P} f.$$

Therefore, from Theorem 5, $\text{CONTC}(\widehat{G}^{(k-1)}[e, f], b)$ evaluates to false. Equivalently, there is no consistent cut of $\widehat{G}^{(k-1)}[e, f]$ that satisfies b. However, $\text{CONTC}(\widehat{G}^{(k)}[e, f], b)$ evaluates to true. This implies that there exists a consistent cut of $\widehat{G}^{(k)}[e, f]$, say, C, that satisfies b. Specifically, 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, which is a contradiction. Since $g \rightarrow e^{(k)}$, C also contains g. Therefore, $\text{CONTC}(\widehat{G}^{(k)}[e, g], b)$ also 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 that an instance of CONTC is evaluated to $O(|C_i^{(k)}| + |C_x^{(k)}| + 1)$. The modification is described in Fig. 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 that 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 2-4 over $|E|$ events is given by

$O(|E|^2)$, which can be ignored by assuming that $T = \Omega(|E|)$. Finally, the amortized time complexity for updating the slice *once*, that is, on the 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 12. *The time complexity of the algorithm to update the slice on the arrival of a new event, as described in Figs. 5 and 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.*

Observe that the ratio of the time required to incrementally update the slice to the time required to compute the slice from scratch is given by

$$O(n(c+n)T)/O(n|E|T) = O((c+n)/|E|),$$

which lies between $O(n/|E|)$ and $O(1)$. Therefore, even in the worst case, the (amortized) time complexity of the online algorithm is *within a constant factor* of the time complexity of the offline algorithm. Also, in case c is low, say, $O(n)$, the ratio is $O(n/|E|) = o(1)$ for any nontrivial computation. In this case, rather than computing the slice from scratch whenever an event arrives, it is much faster to update it by using the incremental algorithm.

The online algorithm in this section only assumes that the predicate can be detected efficiently: No other assumption is made about the structure of the predicate. Next, we describe a much more efficient algorithm to compute the slice incrementally, assuming that the predicate is regular.

5.2 Online Algorithm for Computing the Slice for a Regular Predicate

We say that a predicate is *regular* if it satisfies the following property: Given two consistent cuts satisfying the predicate, the cuts given by their set intersection and set union also satisfy the predicate [6]. Formally, b is regular if, for all consistent cuts C and D,

$$(C \text{ satisfies } b) \wedge (D \text{ satisfies } b) \Rightarrow (C \cap D \text{ satisfies } b) \wedge (C \cup D \text{ satisfies } b).$$

Some examples of regular predicates include

- *conjunctive predicates* (which can be expressed as a conjunction of local predicates) such as "every process is in the red state" [3] and
- *monotonic channel predicates* such as "all red messages have been received" [3].

Our online slicing algorithm for a regular predicate is derived from the offline slicing algorithm (for a regular predicate) described in [6]. The offline slicing algorithm in [6] depends on the notion of J_b, defined as follows: Consider a computation $G = \langle E, \rightarrow \rangle$ and a regular predicate b. Given an event e, $J_b(e)$ is defined as the *least consistent cut* of G that contains e and satisfies b [6]. In case no consistent cut containing e that also satisfies b exists or when $e \in \top$, $J_b(e)$ is set to E. We use the nonempty trivial consistent cut E as a *sentinel* cut and hereafter refer to it as the *default cut*. The offline algorithm for computing the slice for a regular predicate consists of two steps: First, $J_b(e)$ is computed for all events e. Next, $F_b(e)$ is computed for all events e by using the results of the first step. As explained earlier in Section 3, once we know $F_b(e)$ for all events e, we can easily construct the skeletal representation of the slice. We describe how we can incrementally compute/update J_b and F_b by using only $O(n^2)$ amortized time per event.

5.2.1 Updating J_b

We first describe the offline algorithm for computing $J_b(e)$ for all events e.

Background: Offline Algorithm for Computing J_b. To compute $J_b(e)$ for an event e, we define a predicate b_e as follows: b_e evaluates to true for a cut C if C satisfies b and contains e. Note that $J_b(e)$ can be interpreted as the *least* consistent cut of G for which b_e evaluates to true. The problem of determining $J_b(e)$ now reduces to the problem of finding the least consistent cut of G that satisfies b_e. To that end, we observe that b_e is a linear predicate and therefore satisfies the *linearity property* [3] defined as follows: Suppose we are currently at a consistent cut that does not satisfy b_e. The linearity property states that there exists a process p such that, if we do not advance along p, then we can never "reach" a consistent cut satisfying b_e if it exists. The event on p in the frontier of the cut is referred to as a *forbidden event* [3]. In most cases, it is possible to determine a forbidden event in $O(n)$ time.

Now, to find the least consistent cut that satisfies b_e, we scan the computation G once from left to right. Starting from the initial consistent cut $\perp$, we move forward one event at a time until we either reach a consistent cut that satisfies b_e or run out of events. In the latter case, $J_b(e)$ is set to E. In each step, the event to advance beyond is determined as follows: If the current cut is consistent, then we use the linearity property to find a forbidden event and move beyond that event. Otherwise, if the current cut is not consistent, then we advance beyond one of the inconsistent events. This leads to an $O(n|E|)$ algorithm to compute $J_b(e)$ for a *given event* e.

We prove in [6] that J_b is order preserving; that is, if $e \rightarrow f$, then $J_b(e) \subseteq J_b(f)$. This implies that, when computing $J_b(succ(e))$, rather than starting the scan from the initial consistent cut, we can start the scan from $J_b(e)$. As a result, it is possible to compute $J_b(e)$ for all events e on a *given process* in a single scan of the computation in $O(n|E|)$ time.

Online Algorithm for Updating J_b. We modify the offline algorithm for computing J_b to update J_b in an incremental manner. To that end, we categorize events into two: those for which J_b evaluates to a nontrivial consistent cut and those for which J_b evaluates to the default cut. Note that, for a nonfinal event e, $J_b(e)$ is the default cut if and only if no consistent cut of the computation satisfies b_e. We prove that, given an event e, once $J_b(e)$ evaluates to a nontrivial consistent cut, it never changes again.

Lemma 13. *For each event $e \in E^{(k-1)}$,*

$$J_b^{(k-1)}(e) \text{ is a nontrivial consistent cut of } G^{(k-1)} \Rightarrow J_b^{(k-1)}(e) = J_b^{(k)}(e).$$

Lemma 13 is significant because it may greatly reduce the set of events for which J_b has to be recomputed. Specifically, J_b has to be updated for only those events for which it currently evaluates to the default cut. This is

```
Input: (1) a computation G^(k) = ⟨E^(k), →⟩, (2) a regular predicate b, and
       (3) for each event e ∈ E^(k-1), J_b(e) currently set to J_b^(k-1)(e)
Output: for each event e ∈ E^(k), J_b(e) now set to J_b^(k)(e)

// initialize J_b for the new event
1:  J_b(e^(k)) := the default cut;
// add e^(k) to the set of events whose J_b has to be recomputed
2:  recompute_{i_k} := min{recompute_{i_k}, e^(k)};

3:  for each process p_i do
        // update J_b(e) for all events e on process p_i by resuming the scan from current_i
4:      done := (recompute_i = ⊤_i);
5:      while not(done) do
            // update J_b(recompute_i), if required
6:          found := false;
7:          while not(found) and not(done) do
8:              if there exist events g and h in current_i's frontier such that succ(g) → h then
                    // current_i is not a consistent cut
9:                  if succ(g) ∉ ⊤ then current_i := current_i ∪ {succ(g)};
10:                 else done := true; endif;                    // terminate the scan
                else                                             // current_i is a consistent cut
11:                 if current_i satisfies b_{recompute_i} then found := true;
                    else
                        // invoke the linearity property
12:                     g := a forbidden event of current_i with respect to b_{recompute_i};
13:                     if succ(g) ∉ ⊤ then current_i := current_i ∪ {succ(g)};
14:                     else done := true; endif;                // terminate the scan
                    endif;
                endif;
            endwhile;
15:         if (found) then
                // found a consistent cut that contains recompute_i and satisfies b
16:             J_b(recompute_i) := current_i;
                // move to the next event on p_i
17:             recompute_i := succ(recompute_i);
            endif;
18:         if not(found) or (recompute_i = ⊤_i) then
19:             done := true;                                    // terminate the scan
            endif;
        endwhile;
    endfor;
```

Fig. 7. Algorithm to update $J_b(e)$ for each event e on the arrival of a new event when b is regular.

because, if $J_b(e)$ is currently set to the default cut for an event e, then it implies that the current computation does not contain any consistent cut that contains e and satisfies b. However, the arrival of a new event may result in the generation of such consistent cuts in the new computation.

Fig. 7 describes the algorithm for updating J_b on the arrival of a new event. To be able to efficiently determine the set of events for which J_b may have to be updated, for each process p_i, we maintain a variable $recompute_i$. Intuitively, $recompute_i$ keeps track of the *earliest event* on process p_i for which J_b currently evaluates to the default cut. The order-preserving property of J_b guarantees that, for every event e on p_i before $recompute_i$, the current value of $J_b(e)$ is a nontrivial consistent cut, and for every event e on p_i after $recompute_i$, the current value of $J_b(e)$ is the default cut. In addition to $recompute_i$, we maintain another variable $current_i$ that keeps track of how far we have advanced in our scan of the computation from left to right when computing $J_b(recompute_i)$. Our online algorithm for computing J_b basically emulates the offline version. However, we terminate the scan as soon as we realize that the next step involves advancing the scan to include a final event (lines 10 and 14). The reason is given as follows: Suppose the next step in the scan involves moving beyond event e on process p_i to the final event $\top_i$. It is possible that more events may be executed on p_i after e in the future. The offline algorithm, which will have knowledge of all these events, will never make the corresponding move. We also terminate the scan if we fail to find a consistent cut that contains $recompute_i$ and satisfies b (line 19). In contrast, in the offline algorithm, the scan is terminated only after J_b has been computed for all events on a process.

Initially, for each process p_i, $recompute_i$ is $\bot_i$ and $current_i$ is $\bot$. The correctness of the algorithm follows from the fact that every step in the online version that advances the scan to a nonfinal event is a valid step in the offline version as well.

We now analyze the time complexity of the algorithm. Let $move_i^{(k)}$ denote the set of events on process p_i, beyond which $recompute_i$ moves on the arrival of the kth event. Formally,

$$move_i^{(k)} \triangleq \{e \mid recompute_i \text{ advances beyond } e \text{ on arrival of } e^{(k)}\}.$$

Further, let $scanAdvance_i^{(k)}$ denote the set of events beyond which $current_i$ advances on arrival of the kth event. Formally,

$$scanAdvance_i^{(k)} \triangleq \{e \mid succ(e) \text{ is added to } current_i \text{ on arrival of } e^{(k)}\}.$$

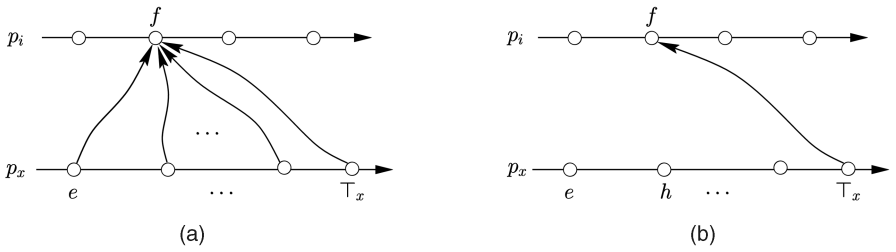

Fig. 8. Reducing the time complexity of updating F_b. In the figure, $f = F_b^{(k-1)}(e)[i]$ and $J_b^{(k-1)}(f) = E^{(k-1)}$.

The time complexity of the ith iteration of the for loop at line 3 of the algorithm is given by

$$O(n(1 + |move_i^{(k)}| + |scanAdvance_i^{(k)}|)).$$

This is because, once $recompute_i$ moves beyond an event e on process p_i, from Lemma 13, J_b of e does not change again. Moreover, once $current_i$ advances beyond an event e, e is not considered again for updating J_b of an event on process p_i. Therefore, the overall time complexity of updating J_b on the arrival of $|E|$ events is given by

$$\begin{aligned}
&\textstyle\sum_{k=1}^{|E|}\sum_{i=1}^{n} O(n(1 + |move_i^{(k)}| + |scanAdvance_i^{(k)}|))\\
=\ &\{\textit{simplifying and changing the order of summations}\}\\
&O(n^2|E|) + n\textstyle\sum_{i=1}^{n}\sum_{k=1}^{|E|} O(|move_i^{(k)}| + |scanAdvance_i^{(k)}|)\\
=\ &\{\textit{there are } |E| \textit{ events in total and } |E_i| \textit{ events on process } p_i\}\\
&O(n^2|E|) + n\textstyle\sum_{i=1}^{n} O(|E_i| + |E|)\\
=\ &\{\textit{simplifying}\}\\
&O(n^2|E|)
\end{aligned}$$

Hence, the time complexity, when amortized over $|E|$ events, is given by $O(n^2)$. Our amortized time complexity analysis assumes that the representation of the default cut does not change as new events arrive. Otherwise, the time complexity of the ith iteration of the for loop in line 3 is given by $O(n(1 + |E_i| + |scanAdvance_i^{(k)}|))$, which results in the amortized time complexity of $O(n|E|)$. Technically, the default cut after the arrival of $k-1$ events, given by $E^{(k-1)}$, is different from the default cut after the arrival of k events, given by $E^{(k)}$. However, as far as computing J_b is concerned, they have the same meaning, serve the same purpose, and can therefore be represented in the same way.

Theorem 14. *The amortized time complexity of the algorithm to update J_b once on the arrival of a new event, described in Fig. 7, is $O(n^2)$, where n is the number of processes.*

5.2.2 Updating F_b

Once the value of J_b is available for all events, the value of F_b can be computed for all events by using the following lemma:

Lemma 15. $\text{CONTC}(\widehat{G}[e, f], b)$ evaluates to false $\equiv e \in J_b(f)$.

Lemma 15 implies that F_b can be updated incrementally by using the online slicing algorithm for a general predicate described in Figs. 5 and 6 except for line 11, which is changed from "while $(f \neq \top_1) \wedge \text{CONTC}(\widehat{G}^{(k)}[e, f], b)$ do" to "while $(f \neq \top_i) \wedge (e \notin J_b(f))$ do." The expression "$e \notin J_b(f)$" can be evaluated in $O(1)$ time by comparing the last event on process p_x in $J_b(f)$ with the event e. Therefore, the amortized time complexity for updating F_b on the arrival of a new event is given by $O(n(c+n))$, which can be as high as $O(n|E|)$ in the worst case.

To understand how the amortized time complexity can be reduced from $O(n(c+n))$ to $O(n^2)$, consider an event $e \in E^{(k-1)}$ and a process p_i and let $f = F_b^{(k-1)}(e)[i]$. In case $J_b^{k-1}(f)$ evaluates to the default cut, it can be verified that the value of $F_b^{k-1}(h)[i]$ for all events h that occurred after e on $proc(e)$ is also f. Formally,

$$\begin{aligned}
&J_b^{(k-1)}(f) \text{ evaluates to the default cut} \Rightarrow\\
&\quad \langle \forall h \in E^{(k-1)} \ : \ e \xrightarrow{P} h \ : \ F_b^{(k-1)}(h)[i] = f \rangle.
\end{aligned}$$

This is depicted in Fig. 8a. Now, consider the arrival of the next event $e^{(k)}$. Suppose $F_b(e)[i]$ changes and advances to the successor of f (that is, $F_b^{(k)}(e)[i] = succ(f)$). It can be verified that the value of $F_b(h)[i]$ for all events h with $e \xrightarrow{P} h$ will change as well and advance to at least the successor of f (that is, $succ(f) \xrightarrow{P} F_b^{(k)}(h)[i]$). In the worst case, the new value of $F_b(h)[i]$ may be $succ(f)$ for all events h with $e \xrightarrow{P} h$. This behavior may be repeated a number of times. In other words, every new event that arrives may cause F_b to be updated for a large number of events on $proc(e)$ (all events h with $e \xrightarrow{P} h$ in our example). This, in turn, implies that a large number of edges in the slice may have to be updated whenever a new event arrives, thereby increasing the amortized time complexity for updating F_b.

Note that the skeletal representation of a slice contains an edge from every event to its successor. From Lemma 2, it follows that the subgraph in Fig. 8a can be substituted with the subgraph in Fig. 8b as far as representing the slice is concerned. For example, in Fig. 8b, consider an event $h \in E^{(k-1)}$ with $e \xrightarrow{P} h$. Since there is a path from h to $\top_x$ and there is an edge from $\top_x$ to f, it implies that there is a path from h to f. Therefore, we define another vector $\widehat{F}_b$, which is derived from F_b as follows:

$$\widehat{F}_b(e)[i] \triangleq \begin{cases} F_b(e)[i] : (J_b(F_b(e)[i]) \text{ is a nontrivial consistent cut}) \text{ or } (e \in \top) \\ \text{null} : \text{otherwise.} \end{cases}$$

Clearly, the graph constructed using $\widehat{F}_b$ instead of F_b constitutes a valid representation of the slice. Consequently, it suffices to give an online algorithm for updating $\widehat{F}_b$ on the arrival of a new event. Note that, once $\widehat{F}_b(e)[i]$, where $e \notin \top$, attains a nonnull value, it does not change again. Formally,

```
Input: (1) a computation G^(k) = ⟨E^(k), →⟩,  (2) a regular predicate b,
       (3) for each event e ∈ E^(k-1), F̂_b(e) currently set to F̂_b^(k-1)(e), and
       (4) for each event e ∈ E^(k), J_b(e) currently set to J_b^(k)(e)
Output: for each event e ∈ E^(k), F̂_b(e) now set to F̂_b^(k)(e)

1:  F̂_b(e^(k)) := null;
    // update the set of earliest events on process p_{i_k}:
    //   earliest event on p_{i_k} with respect to p_i is the first event e on p_{i_k} for which F̂_b(e)[i] is a null value
2:  for each process p_i do earliest_{i_k,i} := min{e^(k), earliest_{i_k,i}};  endfor;

3:  for each process p_x do
4:      for each process p_i do
            // set e to the earliest event on p_x for which F̂_b(e)[i] is a null value
5:          e := earliest_{x,i};
6:          f := F̂_b(⊤_x)[i];
7:          done := false;
            // start the scan from e on p_x and f on p_i
8:          while not(done) do
9:              while (f ≠ ⊤_i) and (e ∉ J_b(f)) do
10:                 f := succ(f);                          // advance to the next event on p_i
                endwhile;
11:             if (J_b(f) evaluates to the default cut) then
                    // F̂_b(e)[i] is a null value
12:                 F̂_b(⊤_x)[i] := f;
13:                 done := true;                          // terminate the scan
                else
                    // F̂_b(e)[i] is a non-null value
14:                 F̂_b(e)[i] := f;
15:                 if (e = ⊤_x) then done := true;        // terminate the scan
16:                 else e := succ(e);                     // advance to the next event on p_x
                    endif;
                endif;
17:             if (done) then earliest_{x,i} := e;  endif;
            endwhile;
        endfor;
    endfor;
```

Fig. 9. An online algorithm to update $\widehat{F}_b(e)$ for all events e on the arrival of a new event.

Observation 16. *We have*

$$(\widehat{F}_b^{(k-1)}(e)[i] \neq \text{null}) \wedge (e \notin \top) \Rightarrow \widehat{F}_b^{(k-1)}(e)[i] = \widehat{F}_b^{(k)}(e)[i].$$

The online algorithm for updating $\widehat{F}_b$ on the arrival of a new event is described in Fig. 9. The main idea is that when scanning events on processes p_x and p_i from left to right to compute $\widehat{F}_b(e)[i]$ (lines 8-16), we stop the scan once we reach an event on process p_i for which J_b evaluates to the default cut (lines 11-13). We now analyze the time complexity of the algorithm. Given processes p_x and p_i, let $A^{(k)}(x,i)$ denote the number of times that e is advanced to its next event at line 16 on the arrival of $e^{(k)}$. Likewise, let $B^{(k)}(x,i)$ denote the number of times that f is advanced to its next event at line 10 on the arrival of $e^{(k)}$. Clearly, the number of times that the while loop at line 8 is executed is given by $O(1 + |A^{(k)}(x,i)| + |B^{(k)}(x,i)|)$. It can be easily verified that

$$\sum_{k=1}^{|E|} |A^{(k)}(x,i)| \leq |E_x| \quad \text{and} \quad \sum_{k=1}^{|E|} |B^{(k)}(x,i)| \leq |E_i|.$$

Now, the time complexity of updating $\widehat{F}_b$ on the arrival of $|E|$ events is given by

$$\begin{aligned}
&\textstyle\sum_{k=1}^{|E|} \sum_{x=1}^{n} \sum_{i=1}^{n} O(1 + |A^{(k)}(x,i)| + |B^{(k)}(x,i)|) \\
&= \{\textit{changing the order of summations}\} \\
&\quad \textstyle\sum_{x=1}^{n} \sum_{i=1}^{n} \sum_{k=1}^{|E|} O(1 + |A^{(k)}(x,i)| + |B^{(k)}(x,i)|) \\
&= \{\textit{simplifying}\} \\
&\quad \textstyle\sum_{x=1}^{n} \sum_{i=1}^{n} O(|E| + |E_x| + |E_i|) \\
&= \{\textit{simplifying}\} \\
&\quad O(n^2|E|)
\end{aligned}$$

Therefore, we have:

Theorem 17. *The amortized time complexity of the algorithm to update $\widehat{F}_b$ once on the arrival of a new event, described in Fig. 9, is $O(n^2)$, where n is the number of processes.*

6 Conclusion

In this paper, we proved the equivalence of two problems in distributed computing, namely, predicate detection and computation slicing. As a virtue of our equivalence result, it is now possible to compute the slice efficiently for a much larger class of predicates. This, in turn, helps reduce the time and space for detecting more "complex" predicates. We also presented two efficient online algorithms for computing the slice. The first algorithm is general in nature and can be used to incrementally compute the slice for any predicate for which it is possible to devise an efficient detection algorithm. The second algorithm can be used to incrementally compute the slice for a regular predicate.

At present, all of our slicing algorithms, offline and online, are centralized in nature. For future work, we plan to develop *distributed* algorithms for computing the slice. Furthermore, our focus so far has been on using slicing to detect a predicate under the *possibly* modality. Sometimes, it is desirable to detect a predicate under other modalities as well, such as *definitely*, *invariant*, and *controllable* [24], [3],

[25], [26]. For instance, we may want to detect whether a predicate eventually holds along *all paths* of a computation (referred to as the *definitely* modality) [24], [3]. In the future, we plan to investigate how slicing can be used to detect predicates under other modalities.

ACKNOWLEDGMENTS

A preliminary version of this paper appeared in the *Proceedings of the International Parallel and Distributed Processing Symposium* (IPDPS) 2004. This work was done while Alper Sen was a student in the Department of Electrical and Computer Engineering at the University of Texas at Austin. Vijay K. Garg was supported in part by US National Science Foundation Grants ECS-9907213 and CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

REFERENCES

[1] N. Mittal, A. Sen, V.K. Garg, and R. Atreya, "Finding Satisfying Global States: All for One and One for All," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS '04),* Apr. 2004.

[2] S.D. Stoller and F. Schneider, "Faster Possibility Detection by Combining Two Approaches," *Proc. Ninth Int'l Workshop Distributed Algorithms (WDAG '95),* pp. 318-332, Sept. 1995.

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

[4] N. Mittal and V.K. Garg, "On Detecting Global Predicates in Distributed Computations," *Proc. 21st IEEE Int'l Conf. Distributed Computing Systems (ICDCS '01),* pp. 3-10, Apr. 2001.

[5] C. Chase and V.K. Garg, "On Techniques and Their Limitations for the Global Predicate Detection Problem," *Proc. Ninth Int'l Workshop Distributed Algorithms (WDAG '95),* pp. 303-317, Sept. 1995.

[6] N. Mittal and V.K. Garg, "Techniques and Applications of Computation Slicing," *Distributed Computing,* vol. 17, no. 3, pp. 251-277, Feb. 2005.

[7] M. Weiser, "Programmers Use Slices When Debugging," *Comm. ACM,* vol. 25, no. 7, pp. 446-452, 1982.

[8] G. Venkatesh, "Experimental Results from Dynamic Slicing of C Programs," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 2, pp. 197-216, 1995.

[9] B. Korel and J. Rilling, "Application of Dynamic Slicing in Program Debugging," *Proc. Third Int'l Workshop Automatic Debugging (AADEBUG '97),* M. Kamkar, ed., pp. 43-57, May 1997.

[10] J. Cheng, "Slicing Concurrent Programs—A Graph-Theoretical Approach," *Proc. First Int'l Workshop Automated and Algorithmic Debugging (AADEBUG '93),* pp. 223-240, 1993.

[11] H.F. Li, J. Rilling, and D. Goswami, "Granularity-Driven Dynamic Predicate Slicing Algorithms for Message Passing Systems," *Automated Software Eng.,* vol. 11, no. 1, pp. 63-89, Jan. 2004.

[12] S.A. Cook, "The Complexity of Theorem Proving Procedures," *Proc. Third Ann. ACM Symp. Theory of Computing (STOC '71),* pp. 151-158, 1971.

[13] A. Sen and V.K. Garg, "Formal Verification of Simulation Traces Using Computation Slicing," *IEEE Trans. Computers,* vol. 56, no. 4, pp. 511-527, Apr. 2007.

[14] S.D. Stoller, L. Unnikrishnan, and Y.A. Liu, "Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods," *Proc. 12th Int'l Conf. Computer Aided Verification (CAV '00),* pp. 264-279, July 2000.

[15] S. Alagar and S. Venkatesan, "Techniques to Tackle State Explosion in Global Predicate Detection," *IEEE Trans. Software Eng.,* vol. 27, no. 8, pp. 704-714, Aug. 2001.

[16] A. Sen and V.K. Garg, "Partial Order Trace Analyzer (POTA) for Distributed Programs," *Proc. Third Workshop Runtime Verification (RV '03),* vol. 89, 2003.

[17] A.D. Kshemkalyani, "A Framework for Viewing Atomic Events in Distributed Computations," *Theoretical Computer Science,* vol. 196, nos. 1-2, pp. 45-70, Apr. 1998.

[18] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[19] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, Feb. 1985.

[20] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier, "Local and Temporal Predicates in Distributed Systems," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 1, pp. 157-179, 1995.

[21] A. Sen and V.K. Garg, "Detecting Temporal Logic Predicates in the Happened-Before Model," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS '02),* Apr. 2002.

[22] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms: Proc. Third Int'l Workshop Distributed Algorithms (WDAG '89),* pp. 215-226, 1989.

[23] C.J. Fidge, "Logical Time in Distributed Computing Systems," *Computer,* vol. 24, no. 8, pp. 28-33, Aug. 1991.

[24] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging (WPDD '91),* pp. 163-173, 1991.

[25] A. Sen and V.K. Garg, "On Checking Whether a Predicate Definitely Holds," *Proc. Third Workshop Formal Approaches to Testing of Software (FATES '03),* pp. 15-29, July 2003.

[26] A. Tarafdar and V.K. Garg, "Predicate Control: Synchronization in Distributed Computations with Look Ahead," *J. Parallel and Distributed Computing,* vol. 64, no. 2, pp. 219-237, Feb. 2004.

Neeraj Mittal received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, in 1995 and the MS and PhD degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a codirector of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking, and databases. He is a member of the IEEE Computer Society.

Alper Sen received the BSc and MSc degrees in electrical and electronics engineering from the Middle East Technical University, Ankara, Turkey, in 1995 and 1997, respectively, and the PhD degree in electrical and computer engineering from the University of Texas at Austin in 2004. He is currently a researcher in the Verification Tools Research and Development group at Freescale Semiconductor Inc., Austin. His research interests include hardware and software verification, concurrent and distributed systems, and formal methods. He is a member of the IEEE.

Vijay K. Garg received the BTech degree in computer science from the Indian Institute of Technology, Kanpur, in 1984 and the MS and PhD degrees in electrical engineering and computer science from the University of California, Berkeley, in 1985 and 1988, respectively. He is currently a full professor in the Department of Electrical and Computer Engineering and the director of the Parallel and Distributed Systems Laboratory at the University of Texas at Austin. His research interests are in the areas of distributed systems and discrete event systems. He is the author of the books *Elements of Distributed Computing* (John Wiley & Sons, 2002) and *Principles of Distributed Systems* (Kluwer, 1996) and a coauthor of the book *Modeling and Control of Logical Discrete Event Systems* (Kluwer, 1995). He is a fellow of the IEEE.