Distrib. Comput. (2005) 17: 251–277
Digital Object Identifier (DOI) 10.1007/s00446-004-0117-0

DISTRIBUTED COMPUTING

Techniques and applications of computation slicing*

Neeraj Mittal[1], **Vijay K. Garg**[2,**]

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA (e-mail: neerajm@utdallas.edu)
[2] Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA (e-mail: garg@ece.utexas.edu)

Received: 19 November 2003 / Revised: 29 July 2004
Published online: 7 February 2005 – © Springer-Verlag 2005

Abstract. Writing correct distributed programs is hard. In spite of extensive testing and debugging, software faults persist even in commercial grade software. Many distributed systems should be able to operate properly even in the presence of software faults. Monitoring the execution of a distributed system, and, on detecting a fault, initiating the appropriate corrective action is an important way to tolerate such faults. This gives rise to the predicate detection problem which requires finding whether there exists a consistent cut of a given computation that satisfies a given global predicate.

Detecting a predicate in a computation is, however, an NP-complete problem in general. In order to ameliorate the associated combinatorial explosion problem, we introduce the notion of computation slice. Formally, the slice of a computation with respect to a predicate is a (sub)computation with the least number of consistent cuts that contains all consistent cuts of the computation satisfying the predicate. Intuitively, slice is a concise representation of those consistent cuts of a computation that satisfy a certain condition. To detect a predicate, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice.

We prove that the slice of a computation is uniquely defined for all predicates. We also present efficient algorithms for computing the slice for several useful classes of predicates. For an arbitrary predicate, we establish that the problem of computing the slice is NP-complete in general. Nonetheless, for such a predicate, we develop an efficient heuristic algorithm for computing an *approximate* slice. Our experimental results demonstrate that slicing can lead to an exponential improvement over existing techniques for predicate detection in terms of time and space.

Keywords: analyzing distributed computation – predicate detection – predicate control – global property evaluation – testing and debugging – software fault tolerance

* Parts of this paper have appeared earlier in conference proceedings [GM01,MG01a,MG03a].

** 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 distributed programs is an error prone activity; it is hard to reason about them because they suffer from the combinatorial explosion problem. Software faults (bugs), in particular global faults, are caused by subtle interactions between various components of the system. As such, they may occur only for specific combinations of inputs and certain interleavings of events. This makes it difficult to eliminate them entirely using testing and debugging. In fact, in spite of extensive testing and debugging, software faults may persist even in commercial grade software. Many distributed systems should be able to operate properly even in the presence of software faults. Monitoring the execution of a distributed system, and, on detecting a fault, initiating the appropriate corrective action is an important way to tolerate such bugs.

A system for tolerating global faults will, in general, consist of three components: *program tracing module*, *fault detection module*, and *fault recovery module*. The program tracing module is responsible for recording the values of variables or objects being monitored (that is, on which the predicate depends) whenever they change. The fault detection module analyzes the trace to check for a possible occurrence of a fault. On detecting a fault, the fault recovery module takes the necessary corrective measure to recover from the fault. It could involve halting the program execution, or resetting the values of program variables, or rolling back the execution of the program to a consistent cut before the fault followed by replay (or retry), possibly under control. The ability to detect global faults is therefore a potentially important step in tolerating them. 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 finding a consistent cut of a distributed computation, if it exists, that satisfies the given global predicate. (This problem is also referred to as detecting a predicate under *possibly* modality in the literature.) Predicate detection problem also arises in other areas in distributed systems such as testing and debugging, for example, when setting conditional breakpoints.

Detecting a predicate in a computation is a hard problem in general [Gar02b,SS95,MG01b]. The reason is the combinatorial explosion in the number of possible consistent cuts. Given n processes each executing 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. In fact, we prove in [MG01b] that detecting a predicate in 2-CNF (conjunctive normal form), 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.

The approaches for solving the predicate detection problem can be divided into three categories. The first approach involves repeatedly computing global snapshots of the computation until the given predicate becomes true [CL85,Bou87, SK86]. This approach works only for stable predicates, that is, predicates that stay true once they become true. Some examples of stable predicates are termination and deadlock. The given predicate may not be stable and may turn true only between two successive snapshots. The second approach is based on searching the state-space of the computation. This approach involves incrementally building the lattice corresponding to the computation until the desired predicate turns true [CM91,JMN95,SUL00,AV01]. Unlike the first approach, this approach can be used to detect unstable predicates. However, the algorithms based on this approach typically have exponential running time. The third approach exploits the structure of the predicate itself – by imposing restrictions – to evaluate its value efficiently for a given computation. Polynomial-time algorithms have been developed for several useful classes of predicates including conjunctive predicates [Gar02b,HMSR98], linear and semi-linear predicates [CG98], and relational predicates [CG95].

We develop the *computation slicing* technique for reducing the size of the computation and therefore the number of consistent cuts to be analyzed for detecting a predicate. The *slice* of a computation with respect to a predicate is the (sub)computation satisfying the following two conditions. First, it contains *all* consistent cuts for which the predicate evaluates to true. Second, among all subcomputations that fulfill the first condition, it contains the *least* number of consistent cuts. Intuitively, slice is a *concise representation* of consistent cuts satisfying a given property. We establish that the slice of a computation is uniquely defined for all predicates. Since we expect global faults to be relatively rare, the state-space of the slice will be much smaller than that of the computation itself. Therefore, in order to detect a global fault, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice.

As an illustration, suppose we want to detect the predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ 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 -1. The first event on each process (namely a on p_1, e 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 two in total, to ascertain whether some consistent cut satisfies the predicate.

Alternatively, we can first compute the slice of the computation with respect to the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3)$. Initially, the value of x_3 is 4 which does not satisfy $x_3 \leqslant 3$. To reach a consistent cut satisfying $x_3 \leqslant 3$, v has to be executed. In other words, any consistent cut in which only u has been executed but not v 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 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 some of the 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 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 $\{a, e, f, u, v\}$, $\{a, e, f, u, v, b\}$, $\{a, e, f, u, v, w\}$, $\{a, e, f, u, v, b, w\}$, $\{a, e, f, u, v, w, g\}$ and $\{a, e, f, u, v, b, w, g\}$. The slice has much fewer consistent cuts than the computation itself – exponentially smaller in many cases – resulting in substantial savings.

The notion of computation slice is similar to the notion of *program slice*, which was introduced by Weiser in [Wei82] to facilitate program debugging. Suppose, during program testing, a programmer observes that, at a certain point, the values of some variables differ from their expected values. Clearly, to locate the error that caused the mismatch, the programmer needs to examine only those statements of the program that directly or indirectly influenced the values of the relevant variables when the mismatch is observed. A program slice, therefore, consists of all those statements of a program that may potentially affect the value of *certain variables* at some *point of interest*. Intuitively, program slicing may significantly reduce the amount of code that needs to be analyzed when debugging a program. Program slicing has been shown to be useful in program debugging, testing, program understanding and software maintenance [Ven95,KR97]. A detailed survey of various program slicing techniques can be found in [Tip95]. 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 distributed programs only. Second, program slicing involves conducting data flow analysis of the program [Che93]. Computation slicing, on the other hand, involves computing join-irreducible elements of a distributive lattice. 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.

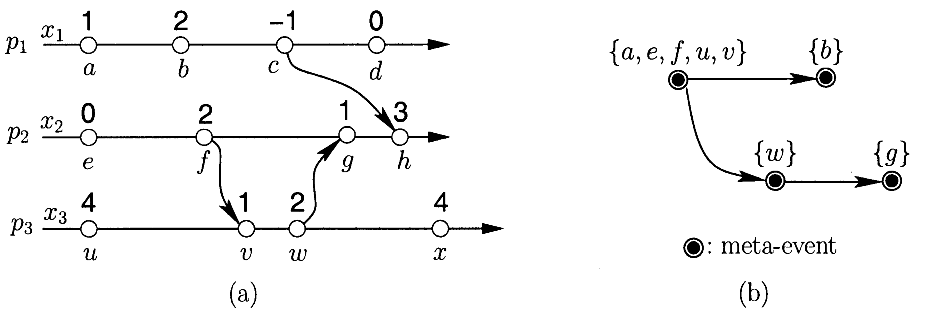


Fig. 1. a A computation and **b** its slice with respect to $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$

The computation slice for a predicate may contain consistent cuts that do not satisfy the predicate. We identify a class of predicates called *regular predicates* for which the slice is *lean*. In other words, the slice for a regular predicate contains precisely those consistent cuts for which the predicate evaluates to true. The set of consistent cuts satisfying a regular predicate forms a sublattice (of the lattice of consistent cuts). Some examples of regular predicates are: *conjunctive predicates*, which can be expressed as conjunction of local predicates, like "all processes are in red state" [Gar02b], and *monotonic channel predicates* such as "all control messages have been received" [Gar02b]. We prove that the class of regular predicates is closed under conjunction, that is, the conjunction of two regular predicates is also a regular predicate. We devise an efficient algorithm to compute the slice for a regular predicate. The time-complexity of the algorithm is $O(n^2|E|)$, where n is the number of processes and E is the set of events. In case the regular predicate can be decomposed into a conjunction of clauses, where each clause itself is a regular predicate but depends on variables of only a small subset of processes, a faster algorithm for computing the slice is provided. Also, for special cases of regular predicates such as conjunctive predicates and certain monotonic channel predicates, we derive *optimal* algorithms for computing the slice, which have $O(|E|)$ time-complexity.

In addition to regular predicates, we also design efficient algorithms to compute the slice for many classes of non-regular predicates such as *linear predicates* and *post-linear predicates* [Gar02b]. Our algorithms have time-complexity of $O(n^2|E|)$. We prove that it is intractable in general to compute the slice for an arbitrary predicate. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently. An approximate slice may be bigger than the actual slice but will be much smaller than the computation itself. To that end, we develop efficient algorithms to compose two slices. Specifically, given two slices, *composition*[1] involves computing either (1) the smallest slice that contains all consistent cuts common to both the slices, or (2) the smallest slice that contains all consistent cuts that belong to at least one of the slices. We use slice composition to efficiently compute the slice for a *co-regular predicate* – the complement of a regular predicate – and a *k-local predicate* – depends on variables of at most k processes – for constant k [SS95]. The algorithms have time-complexities of $O(n^2|E|^2)$ and $O(nm^{k-1}|E|)$, respectively, where m is the maximum number of events on a process. More importantly, we use slice composition to compute an approximate slice – in polynomial-time – for a predicate derived from other predicates for which it is possible to compute the "exact" slice efficiently, using $\wedge$ and $\vee$ operators. Example of such a predicate is: $(x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_1) \wedge (x_2 \vee x_3)$, where each x_i is a linear predicate. Finally, we conduct simulation tests to experimentally measure the effectiveness of computation slicing in pruning the search space when detecting a global fault. Our results indicate that slicing can lead to an *exponential* improvement over existing techniques in terms of time and space. Furthermore, other techniques for reducing the time-complexity [SUL00] and/or the space-complexity [AV01] are orthogonal to slicing, and as such can actually be used in conjunction with slicing. For instance, Alagar and Venkatesan's polynomial space algorithm [AV01] for searching the state-space of a computation can also be used for searching the state-space of a slice.

Although, in this paper, we focus on application of computation slicing to predicate detection, slicing can also be employed to reduce the search-space when monitoring a predicate under other modalities including *definitely*, *invariant* and *controllable* [CM91,Gar02b, BFR96,MG04]. We also show that many results pertaining to consistent global checkpoints can be derived as special cases of slicing. In particular, we furnish an alternate characterization of the condition under which individual local checkpoints can be combined with others to form a consistent global checkpoint (consistency theorem by Netzer and Xu [NX95]): a set of local checkpoints can belong to the same consistent global snapshot if and only if the local checkpoints in the set are mutually consistent (including with itself) in the slice. Moreover, the R-graph (rollback-dependency graph) defined by Wang [Wan97] is a special case of the slice. The minimum and maximum consistent global checkpoints that contain a set of local checkpoints [Wan97] can also be easily obtained using the slice. We have applied slicing to solve several problems in combinatorics as well [Gar02a].

In [Ksh98], Kshemkalyani describes a unifying framework for viewing a distributed computation at multiple levels of atomicity. In particular, Kshemkalyani defines an execution of a system in terms of certain *elementary* events. System executions at 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, 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 meta-event depending on the predicate. Furthermore, our focus is on developing ef-

[1] Composition was called grafting in our earlier paper [MG01a]

ficient algorithms for automatically computing the slice of a computation for a given predicate.

The paper is organized as follows. Section 2 describes our model of distributed system and the notation we use in this paper. We formally define the notion of computation slice in Sect. 3. In Sect. 4, we introduce the class of regular predicates, using which we establish the uniqueness of slice for all predicates in Sect. 5. Section 6 describes an efficient representation for slice. In Sect. 7 and Sect. 8, we discuss our results pertaining to regular predicates and general predicates, respectively. Finally, in Sect. 9, we describe our results in slicing and applications of slicing to solving problems in combinatorics.

2 Model and notation

We assume an asynchronous distributed system comprising of many processes which communicate with each other by sending messages over a set of channels. There is no global clock or shared memory. Processes are non-faulty and channels are reliable. Channels may be non-FIFO. Message delays are finite but unbounded.

Traditionally, a distributed computation is modeled as a partial order on a set of events [Lam78]. In this paper, we relax the restriction that the order on events must be a partial order. Instead, we use directed graphs on events to model distributed computations as well as slices. Directed event graphs allow us to handle both of them in a uniform and convenient manner.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote its set of vertices and edges, respectively. A subset of vertices of a directed graph forms a *consistent cut* if the subset contains a vertex only if it also contains all its incoming neighbors. Formally,

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

Observe that a consistent cut either contains all vertices in a cycle or none of them. This observation can be generalized to a strongly connected component. Traditionally, the notion of consistent cut (*down-set* or *order ideal*) is defined for partially ordered sets [DP90]. Here, we extend the notion to sets with arbitrary orders. Let $\mathcal{C}(G)$ denote the set of consistent cuts of a directed graph G. Observe that the empty set $\emptyset$ and the set of vertices $\mathsf{V}(G)$ trivially belong to $\mathcal{C}(G)$. We call them *trivial* consistent cuts. Let $\mathcal{P}(G)$ denote the set of *connected* pairs of vertices (u,v) such that there is a path from u to v in G. We assume that each vertex has a path to itself.

2.1 Directed graph: path- and cut-equivalence

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

Likewise, a directed graph G is *path-equivalent* to a directed graph H, denoted by $G \overset{P}{\cong} H$, if a path from vertex u to vertex v in G implies a path from vertex u to vertex v in H, and vice versa. Formally, $G \overset{P}{\cong} H \ \triangleq \ \mathcal{P}(G) = \mathcal{P}(H)$. We assume that every vertex has a path to itself. The next lemma explores the relation between the two notions.

Lemma 1. *Let G and H be directed graphs with the same set of vertices. Then,*

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

Evidently, Lemma 1 implies that two directed graphs are cut-equivalent if and only if they are path-equivalent. In other words, to determine whether two directed graphs are cut-equivalent, it is *necessary and sufficient* to ascertain that they are path-equivalent. This is significant because testing for cut-equivalence directly by verifying that the two graphs have the same set of consistent cuts is computationally expensive in general ($|\mathcal{C}(G)| = O(2^{|\mathsf{V}(G)|})$). However, testing for path-equivalence by verifying that the two graphs have the same set of connected vertex pairs is cheap and can be accomplished in polynomial-time ($|\mathcal{P}(G)| = O(|\mathsf{V}(G)|^2)$). In the rest of the paper, we use $\cong$ to denote both $\overset{C}{\cong}$ and $\overset{P}{\cong}$.

2.2 Distributed computation as directed graph

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 only to 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 [Lam78]. 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 can be viewed as a *meta-event*; all events in a meta-event should be executed atomically.

We denote the set of processes in the system by $P = \{p_1, p_2, \ldots, p_n\}$. For an event e, let $proc(e)$ denote the process on which 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 events e and f occur on the same process and e occurs before f in real-time, then we write $e \overset{P}{\rightarrow} f$. Let $\overset{P}{\underline{\rightarrow}}$ denote the reflexive closure of $\overset{P}{\rightarrow}$.

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 facilitate the description 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. As a result, 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 real interest to us. As we will see later, the extended model allows us to capture empty slices in a very convenient fashion.

The *frontier* of a consistent 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 consistent cut *passes through* an event if the event belongs to the frontier of the cut. Two events are said to be *consistent* if they are contained in the frontier of some consistent cut, otherwise they are *inconsistent*. It can be verified that events e and f are consistent if and only if there is no path in the computation from $succ(e)$, if it exists, to f and from $succ(f)$, if it exists, to e. Note that, in the extended model, in contrast to the traditional model, an event can be inconsistent with itself.

2.3 Global predicate

A *global predicate* (or simply a *predicate*) is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the values of the variables 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. Note that it is possible to evaluate a local predicate with respect to an *event* on the appropriate process. In case the predicate evaluates to true, the event is called a *true event*; otherwise, it is called a *false event*. Further, a predicate is said to be *k-local* if it depends on variables of at most k processes [SS95]. For example, suppose x_i is an integer variable on process p_i for each $i \in [1 \ldots n]$. Then, $x_1 + x_2 < 3$ is an example of 2-local predicate, and $x_1 * x_2 + x_3 < 6$ an example of 3-local predicate.

In this paper, when deriving the time-complexity of a slicing algorithm, we assume that the time-complexity of evaluating a predicate for a given consistent cut is $O(n)$, where n is the number of processes in the computation. In case the time-complexity is higher, the time-complexity of the slicing algorithm will increase proportionately.

3 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. Recall that the set of consistent cuts of a computation $\langle E, \rightarrow \rangle$ is denoted by $\mathcal{C}(\langle E, \rightarrow \rangle)$. For reasons of clarity, we abbreviate $\mathcal{C}(\langle E, \rightarrow \rangle)$ by $\mathcal{C}(E)$. Also, for 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). *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,*

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

We use $\langle E, \rightarrow \rangle_b$ to denote a slice of $\langle E, \rightarrow \rangle$ with respect to b. We show in Sect. 5 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 above definition) then $\mathcal{C}(S) = \mathcal{C}(T)$. In that case, from Lemma 1, $\mathcal{P}(S) = \mathcal{P}(T)$. In other words, although graphs S and T may have different sets of edges, their transitive closures are identical. Note that $\langle E, \rightarrow \rangle$ can be written as $\langle E, \rightarrow \rangle_{\mathsf{true}}$. Therefore a computation can also be viewed as a slice. In the rest of the paper, we use the terms "computation", "slice" and "directed graph" interchangeably.

Note that every slice derived from the computation $\langle E, \rightarrow \rangle$ will have the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. Thus a slice is *empty* if it has no non-trivial consistent cuts. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called *lean*. Formally,

Definition 2 (lean slice). *The slice of a computation with respect to a predicate is* lean *if every consistent cut of the slice satisfies the predicate.*

An interesting question to ask is: "For what class of predicates is the slice always lean?" To answer this question, we introduce the class of regular predicates.

4 Regular predicate

Informally, the set of consistent cuts that satisfy a regular predicate is closed under set intersection and set union. Formally,

Definition 3 (regular predicate). *A global predicate is called* regular *if, given two consistent cuts that satisfy the predicate, the consistent cuts given by their set intersection and set union also satisfy the predicate. Mathematically, given a regular predicate b and consistent cuts C_1 and C_2,*

$$\begin{gathered}(C_1 \text{ satisfies } b) \wedge (C_2 \text{ satisfies } b) \\ \Rightarrow \\ (C_1 \cap C_2 \text{ satisfies } b) \wedge (C_1 \cup C_2 \text{ satisfies } b)\end{gathered}$$

It can be verified that a local predicate is regular. Hence the following predicates are regular.

- process p_i is in "red" state
- the leader has sent all "prepare to commit" messages

We now provide more examples of regular predicates. Consider a function $f(x, y)$ with two arguments such that it is monotonic in its first argument x but anti-monotonic in its second argument y. Some examples of the function f are: $x - y$, $3x - 5y$, x/y when $x, y > 0$, and $\log_y x$ when $x, y \geqslant 1$. We establish that the predicates of the form $f(x, y) < c$ and $f(x, y) \leqslant c$, where c is some constant, are regular when either both x and y are monotonically non-decreasing variables or both x and y are monotonically non-increasing variables.

Lemma 2. *Let x and y be monotonically non-decreasing variables. Then the predicates $f(x, y) < c$ and $f(x, y) \leqslant c$ are regular predicates.*

Proof. We show that the predicate $f(x, y) < c$ is regular. The proof for the other predicate is similar and has been omitted. For a consistent cut C, let $x(C)$ and $y(C)$ denote the values of variables x and y, respectively, immediately after all events in C are executed. Consider consistent cuts C_1 and C_2 that satisfy the predicate $f(x, y) < c$. Note that, by definition of $C_1 \cap C_2$, $y(C_1 \cap C_2)$ is either $y(C_1)$ or $y(C_2)$. Without loss of generality, assume that $y(C_1 \cap C_2) = y(C_1)$. Then,

$$
\begin{array}{cl}
 & f(x(C_1 \cap C_2), y(C_1 \cap C_2)) \\
= & \{ \text{ assumption } \} \\
 & f(x(C_1 \cap C_2), y(C_1)) \\
\leqslant & \left\{ \begin{array}{l} x \text{ is monotonically non-decreasing implies} \\ x(C_1 \cap C_2) \leqslant x(C_1), \text{and } f \text{ is monotonic in } x \end{array} \right\} \\
 & f(x(C_1), y(C_1)) \\
< & \{ \ C_1 \text{ satisfies the predicate } f(x, y) < c \ \} \\
 & c
\end{array}
$$

Thus $C_1 \cap C_2$ satisfies the predicate $f(x, y) < c$. Likewise, it can be proved that $C_1 \cup C_2$ satisfies the predicate $f(x, y) < c$. □

It can be established that Lemma 2 holds even when both x and y are monotonically non-increasing variables. Similar results can be proved for the case when $<$ and $\leqslant$ are replaced by $>$ and $\geqslant$, respectively. All the above-mentioned results can be combined as follows:

Theorem 1. *Let f be a function with two arguments such that it is monotonic in its first argument and anti-monotonic in its second argument. Then the predicate of the form $f(x, y)$ `relop` c, where `relop` $\in \{<, \leqslant, >, \geqslant\}$ and c is some constant, is regular when either both x and y are monotonically non-decreasing variables or both x and y are monotonically non-increasing variables.*

By substituting $f(x, y)$ with $x - y$, x with "the number of messages that process p_i has sent to process p_j so far" and y with "the number of messages sent by process p_i that process p_j has received so far", it can be verified that the following predicates are regular.

- no outstanding message in the channel from process p_i to process p_j
- at most k messages in transit from process p_i to process p_j
- at least k messages in transit from process p_i to process p_j

We next show that the conjunction of two regular predicates is also a regular predicate.

Theorem 2. *The class of regular predicates is closed under conjunction.*

The proof is given in the appendix. The closure under conjunction implies that the following predicates are also regular.

- any conjunction of local predicates
- no process has the token and no channel has the token
- every "request" message has been "acknowledged" in the system

5 Establishing the uniqueness of slice

In this section, we show that the slice of a computation with respect to a predicate is uniquely defined. Our approach is to first prove that the slice for a regular predicate is uniquely defined using which we show that the slice is uniquely defined even for a predicate that is not regular.

5.1 Regular predicate

It is well known in distributed systems that the set of all consistent cuts of a computation forms a lattice under the subset relation [JZ88,Mat89]. We ask the question: does the lattice of consistent cuts satisfy any additional property? The answer to this question is in affirmative. Specifically, we show that the set of consistent cuts of a directed graph not only forms a lattice but that the lattice is *distributive*. A lattice is said to be *distributive* if meet distributes over join [DP90]. Formally,

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

where $\sqcap$ and $\sqcup$ denote the meet (infimum) and join (supremum) operators, respectively. (It can be proved that meet distributes over join if and only if join distributes over meet.)

Theorem 3. *Given a directed graph G, $\langle \mathcal{C}(G); \subseteq \rangle$ forms a distributive lattice.*

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

$$
\begin{array}{rcl}
C_1 \sqcap C_2 & \triangleq & C_1 \cap C_2 \\
C_1 \sqcup C_2 & \triangleq & C_1 \cup C_2
\end{array}
$$

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

The above theorem is a generalization of the result in lattice theory that the set of down-sets of a partially ordered set forms a distributive lattice [DP90]. We further prove that the set of consistent cuts (of a directed graph) does not satisfy any additional structural property. To that end, we need the notion of *join-irreducible* element defined as follows.

Definition 4 (join-irreducible element [DP90]). *An element of a lattice is* join-irreducible *if (1) it is not the least element of the lattice, and (2) it cannot be expressed as join of two distinct elements, both different from itself. Formally, $a \in L$ is join-irreducible if*

$$\langle \exists\, x \in L :: x < a \rangle \ \wedge$$
$$\langle \forall\, x, y \in L : a = x \sqcup y : (a = x) \vee (a = y) \rangle$$

Pictorially, an element of a lattice is join-irreducible if and only if it has exactly one lower cover, that is, it has exactly one incoming edge in the corresponding Hasse diagram. The

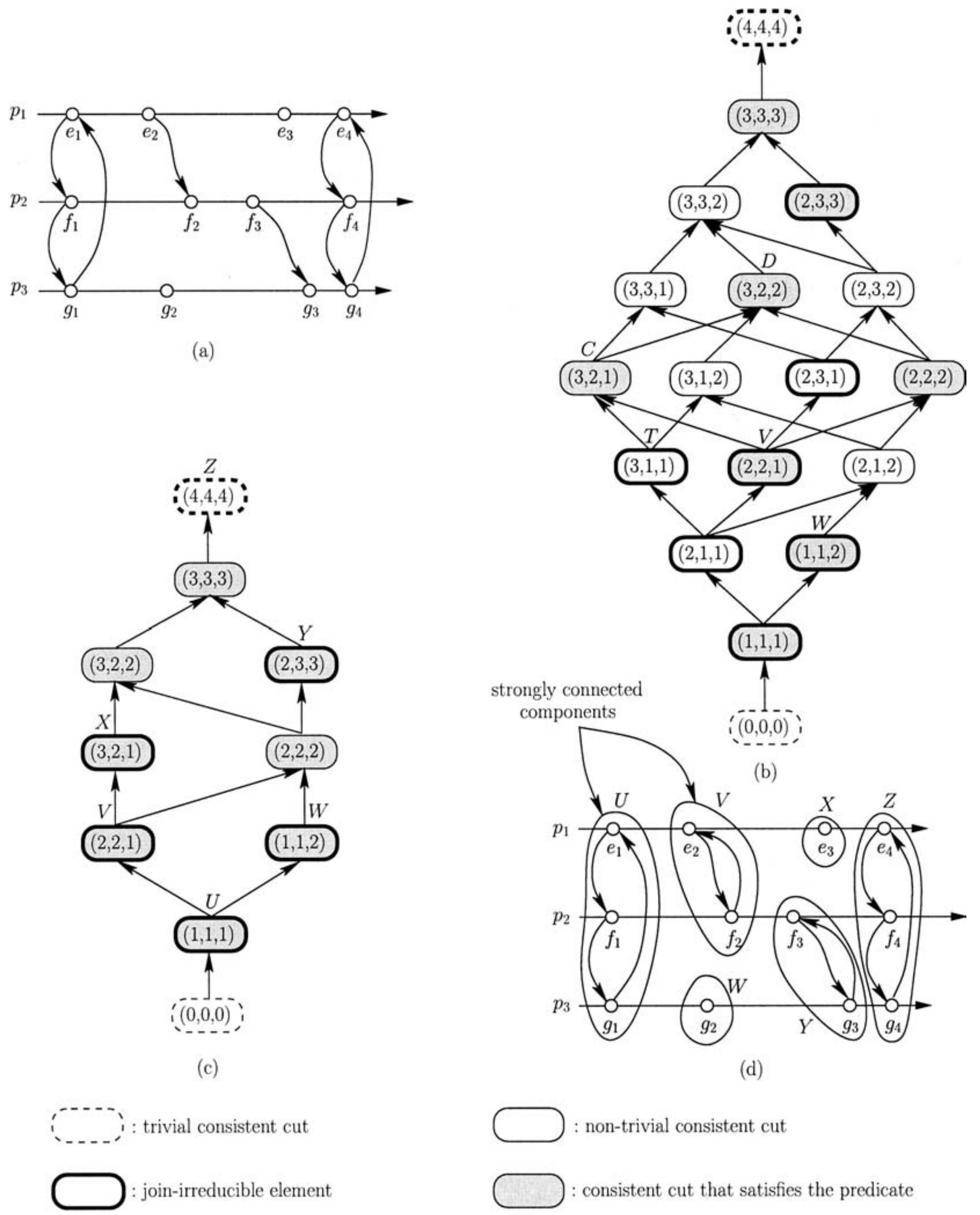

(a)
(b)
(c)
(d)
strongly connected components
: trivial consistent cut
: non-trivial consistent cut
: join-irreducible element
: consistent cut that satisfies the predicate

Fig. 2. **a** A computation, **b** the lattice of its consistent cuts, **c** the sublattice of the consistent cuts that satisfy the regular predicate "all channels are empty", and **d** the poset induced on the set of join-irreducible elements of the sublattice

notion of *meet-irreducible element* can be similarly defined. It turns out that a distributive lattice is uniquely characterized by the set of its join-irreducible elements. In particular, every element of the lattice can be written as join of some subset of its join-irreducible elements, and vice versa. This is formally captured by the next theorem.

Theorem 4 (Birkhoff's Representation Theorem for Finite Distributive Lattices [DP90]). *Let L be a finite distributive lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible elements. Then the map $f : L \longrightarrow \mathcal{C}(\mathcal{JI}(L))$ defined by*

$$f(a) = \{\, x \in \mathcal{JI}(L) \mid x \leqslant a \,\}$$

is an isomorphism of L onto $\mathcal{C}(\mathcal{JI}(L))$. Dually, let P be a finite poset (partially ordered set). Then the map $g : P \longrightarrow \mathcal{JI}(\mathcal{C}(P))$ defined by

$$g(a) = \{\, x \in P \mid x \leqslant a \,\}$$

is an isomorphism of P onto $\mathcal{JI}(\mathcal{C}(P))$.

Note that the above theorem can also be stated in terms of meet-irreducible elements.

Example 1. Consider the computation depicted in Fig. 2a. Figure 2b depicts the lattice of consistent cuts of the computation. In the figure, the label of a consistent cut indicates the number of events that have to be executed on each process to reach the cut. For example, the label of the consistent cut C is $(3, 2, 1)$ implying that to reach C, three events have to executed on process p_1, two on p_2 and one on p_3. Mathematically, $C = \{e_1, e_2, e_3, f_1, f_2, g_1\}$.

In Fig. 2b, the consistent cuts of the computation corresponding to the join-irreducible elements of the lattice have been drawn in thick lines. There are eight join-irreducible elements which is same as the number of strongly connected components of the computation. Note that the poset induced on the set of strongly connected components of the computation is isomorphic to the poset induced on the set of join-irreducible elements of the lattice. It can be verified that every consistent cut of the computation can be expressed as the join of some subset of these join-irreducible elements. For example, the consistent cut C can be written as the join of the consistent cuts T and V. Moreover, the join of every subset of these join-irreducible elements is a consistent cut of the computation. For instance, the join of the consistent cuts T, V and W is given by the consistent cut D. □

In this paper, we are concerned with only a subset of consistent cuts and not the entire set of consistent cuts. To that end, the notion of sublattice of a lattice comes in useful [DP90]. Given a lattice, a subset of its elements forms a *sublattice* if the subset is closed under the meet and join operators of the given lattice. In our case, the meet and join operators are set intersection and set union, respectively. Clearly, the set of consistent cuts satisfying a regular predicate forms a sublattice of the lattice of consistent cuts. Finally, we make an important observation regarding a sublattice which will help us prove the desired result.

Lemma 3 ([DP90]). *A sublattice of a distributive lattice is also a distributive lattice.*

Example 2. In Fig. 2b, the consistent cuts for which the regular predicate "all channels are empty" evaluates to true have been shaded. Figure 2c depicts the poset induced on these consistent cuts. It can be verified that the poset forms a sublattice of the lattice in Fig. 2b. Moreover, the sublattice is a distributive lattice. □

Theorem 5. *The slice of a computation with respect to a regular predicate is uniquely defined.*

Proof. Consider a computation $\langle E, \rightarrow \rangle$ and a regular predicate b. Clearly, from the definition of regular predicate, $\mathcal{C}_b(E)$, is closed under set intersection and set union. As a result, $\mathcal{C}_b(E)$ forms a sublattice of $\mathcal{C}(E)$. Since $\mathcal{C}(E)$ forms a distributive lattice, from Lemma 3, $\mathcal{C}_b(E)$ also forms a distributive lattice. Thus, from Theorem 4, $\mathcal{C}_b(E)$ is uniquely characterized by the set of its join-irreducible elements $\mathcal{JI}(\mathcal{C}_b(E))$. In other words, the set of consistent cuts of the poset induced on $\mathcal{JI}(\mathcal{C}_b(E))$ is identical to $\mathcal{C}_b(E)$. Clearly, the slice of $\langle E, \rightarrow \rangle$ with respect to b is isomorphic to the poset induced on $\mathcal{JI}(\mathcal{C}_b(E))$. □

We now prove that the slice for a predicate is lean if and only if the predicate is regular.

Theorem 6. *The slice of a computation with respect to a predicate is lean if and only if the predicate is regular.*

Proof. (*if*) Assume that the predicate, say b, is regular. Thus the set of consistent cuts that satisfy the predicate, denoted by $\mathcal{C}_b$, forms a sublattice of the lattice of consistent cuts (of the computation). From Lemma 3, $\mathcal{C}_b$ is in fact a distributive lattice. Let $\mathcal{JI}(\mathcal{C}_b)$ denote the set of join-irreducible elements of $\mathcal{C}_b$. From Birkhoff's Representation Theorem, $\mathcal{C}_b$ is isomorphic to $\mathcal{C}(\mathcal{JI}(\mathcal{C}_b))$. Thus the required slice is given by the poset induced on $\mathcal{JI}(\mathcal{C}_b)$ by $\subseteq$. Moreover, every consistent cut of the slice satisfies the predicate and therefore the slice is lean.

(*only if*) Assume that the slice of a computation with respect to a predicate is lean. From the proof of Theorem 3, the set of consistent cuts of the slice is closed under set union and set intersection. This in turn implies that the set of consistent cuts that satisfy the predicate is closed under set union and set intersection. Thus the predicate is regular. □

Example 3. The sublattice shown in Fig. 2c has exactly six join-irreducible elements, namely U, V, W, X, Y and Z. These elements (or consistent cuts) have been drawn in thick lines. It can be ascertained that every consistent cut in the sublattice can be written as the join of some subset of the consistent cuts in $\mathcal{J} = \{U, V, W, X, Y, Z\}$. In other words, every consistent cut of the computation that satisfies the regular predicate "all channels are empty" can be represented as the join of some subset of the elements in $\mathcal{J}$. Moreover, the join of every subset of elements in $\mathcal{J}$ yields a consistent cut contained in the sublattice and hence a cut for which "all channels are empty". The poset induced on the elements of $\mathcal{J}$ by the relation $\subseteq$ is shown in Fig. 2d. (Recall that each join-irreducible element corresponds to a strongly connected component, that is, a meta-event.) This poset corresponds to the slice of the computation shown in Fig. 2a with respect to the regular predicate "all channels are empty". □

Theorem 6 implies that for every sublattice of the lattice of consistent cuts of a computation, there is a regular predicate, and vice versa.

5.2 Non-regular predicate

To prove that the slice exists even for a predicate that is not a regular predicate, we define a closure operator, denoted by reg, which, given a computation, converts an arbitrary predicate into a regular predicate satisfying certain properties. Given a computation $\langle E, \rightarrow \rangle$, let $\mathcal{R}(E)$ denote the set of predicates that are regular with respect to the computation ($\rightarrow$ is implicit).

Definition 5 (reg). *Given a predicate b, we define $reg\,(b)$ as the predicate that satisfies the following conditions:*

1. *it is regular, that is, $reg\,(b) \in \mathcal{R}(E)$,*
2. *it is weaker than b, that is, $b \Rightarrow reg\,(b)$, and*
3. *it is stronger than any other predicate that satisfies (1) and (2), that is,*

$$\langle \forall\, u : u \in \mathcal{R}(E) : (b \Rightarrow u) \Rightarrow (reg\,(b) \Rightarrow u) \rangle$$

Informally, $reg\,(b)$ is the *strongest regular predicate weaker* than b. In general, $reg\,(b)$ not only depends on the predicate b, but also on the computation under consideration.

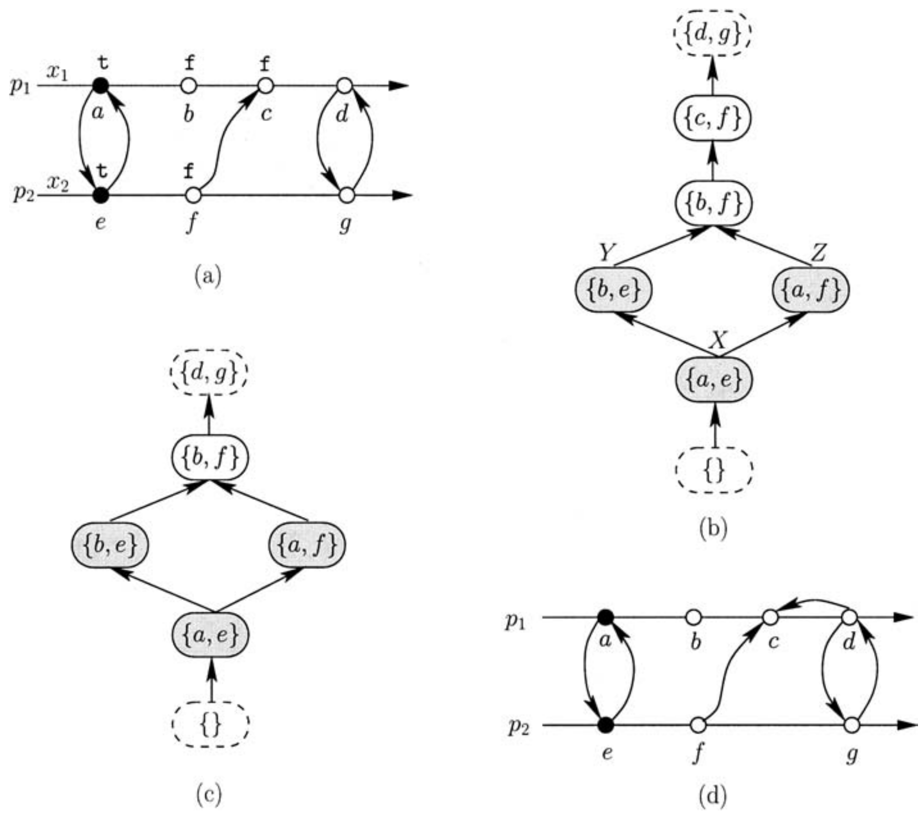


Fig. 3. a A computation, **b** the lattice of its consistent cuts, **c** the sublattice of its consistent cuts that satisfy $reg\,(x_1 \vee x_2)$, and **d** its slice with respect to $reg\,(x_1 \vee x_2)$ (and therefore also with respect to $x_1 \vee x_2$)

We assume the dependence on computation to be implicit and make it explicit only when necessary. The next theorem establishes that $reg\,(b)$ exists and is uniquely defined for every predicate b.

Theorem 7. *The predicate $reg\,(b)$ exists and is uniquely defined for every predicate b.*

Proof. Consider a predicate b. First, we show that if $reg\,(b)$ exists, then $reg\,(b)$ is uniquely defined. Then, we show that $reg\,(b)$ exists.

(uniqueness) Assume that $reg\,(b)$ exists. Suppose there are two predicates u and v that satisfy the three conditions specified in Definition 5. Since u satisfies the first two conditions and v satisfies the third condition, $v \Rightarrow u$. Likewise, $u \Rightarrow v$. Combining the two implications, $u \equiv v$.

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

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

It remains to be shown that $reg\,(b)$ as defined satisfies the three required conditions. Now, first condition holds because the class of regular predicates is closed under conjunction. The second condition holds because every predicate in $\mathcal{R}_b(E)$ is weaker than b and hence their conjunction is weaker than b. Finally, let u be a predicate that satisfies the first two conditions. Note that $u \in \mathcal{R}_b(E)$. Since conjunction of predicates is stronger than any of its conjunct, $reg\,(b)$ is stronger than u. Thus $reg\,(b)$ satisfies the third condition. □

We now prove that the slice is uniquely defined for every predicate.

Theorem 8. *The slice of a computation is uniquely defined with respect to every predicate.*

Proof. Consider a computation $\langle E, \rightarrow \rangle$ and a predicate b. For convenience, let $r = reg\,(b)$. Since $b \Rightarrow r$, $\mathcal{C}_b(E) \subseteq \mathcal{C}_r(E) \subseteq \mathcal{C}(E)$. We claim that $\mathcal{C}_r(E)$ is the *smallest* sublattice of $\mathcal{C}(E)$ that contains $\mathcal{C}_b(E)$. Let S be a sublattice of $\mathcal{C}(E)$ that contains $\mathcal{C}_b(E)$. Further, let s be the regular predicate corresponding to S. Since $\mathcal{C}_b(E) \subseteq S$, $b \Rightarrow s$. From definition of $reg\,(b)$, $reg\,(b) \Rightarrow s$, which, in turn, implies that S contains $\mathcal{C}_r(E)$. In other words, every sublattice of $\mathcal{C}(E)$ that contains $\mathcal{C}_b(E)$ also contains $\mathcal{C}_r(E)$. Clearly, the slice of $\langle E, \rightarrow \rangle$ with respect to b is given by the slice of $\langle E, \rightarrow \rangle$ with respect to $reg\,(b)$. □

Note that although the slice for b is same as the slice for $reg\,(b)$, computing a boolean formula for $reg\,(b)$ using a boolean formula for b is an intractable problem. We therefore explore alternative approaches for computing the slice for b that do not require computation of $reg\,(b)$.

Example 4. Consider the computation depicted in Fig. 3a. The lattice of its consistent cuts is shown in Fig. 3b. Each consistent cut is labeled with its frontier. The consistent cuts for which the predicate $x_1 \vee x_2$ evaluates to true have been shaded in the figure. Clearly, the set of consistent cuts that satisfy $x_1 \vee x_2$ does not form a sublattice. The smallest sublattice containing the subset is shown in Fig. 3c; the sublattice corresponds to the predicate $reg\,(x_1 \vee x_2)$. The slice for the regular predicate $reg\,(x_1 \vee x_2)$ and hence for the predicate $x_1 \vee x_2$ is portrayed in Fig. 3d. □

Theorem 9. *reg is a closure operator. Formally,*

1. *$reg\,(b)$ is weaker than b, that is, $b \Rightarrow reg\,(b)$,*
2. *reg is monotonic, that is, $(b \Rightarrow u) \Rightarrow (reg\,(b) \Rightarrow reg\,(u))$, and*
3. *reg is idempotent, that is, $reg\,(reg\,(b)) \equiv reg\,(b)$.*

From the above theorem it follows that [DP90, Theorem 2.21],

Corollary 1. *$\langle \mathcal{R}(E); \Rightarrow \rangle$ forms a lattice.*

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

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

The dual notion of $reg\,(b)$, the weakest regular predicate stronger than b, is also conceivable. However, such a predicate may not always be unique.

Example 5. In the previous example, three consistent cuts satisfy the predicate $x_1 \vee x_2$, namely X, Y and Z, as shown in Fig. 3b. Two distinct subsets of the set $\mathcal{S} = \{X, Y, Z\}$, given by $\{X, Y\}$ and $\{X, Z\}$, form maximal sublattices of $\mathcal{S}$ implying that there is no weakest regular predicate that is stronger than $x_1 \vee x_2$. □

6 Representing a slice

Any directed graph that is cut-equivalent to a slice constitutes a valid representation of the slice. However, for computational purposes, it is preferable to select those graphs to represent a slice that have fewer edges and can be constructed cheaply. In this section, we show that every slice can be represented by a directed graph with $O(|E|)$ vertices and $O(n|E|)$ edges.

Recall that the set of consistent cuts of a computation $\langle E, \rightarrow \rangle$ that satisfy a regular predicate b is denoted by $\mathcal{C}_b(E)$. From Birkhoff's Representation Theorem, the poset induced on $\mathcal{JI}(\mathcal{C}_b(E))$ by the relation $\subseteq$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. It can be proved that $|\mathcal{JI}(\mathcal{C}_b(E))|$ is upper-bounded by $|E|$. Therefore the directed graph corresponding to $\langle \mathcal{JI}(\mathcal{C}_b(E)); \subseteq \rangle$ may have $\Omega(|E|^2)$ edges.

In order to reduce the number of edges, we exploit properties of join-irreducible elements. For an event e, let $J_b(e)$ denote the *least consistent cut* of $\langle E, \rightarrow \rangle$ that satisfies b and contains e. In case no consistent cut containing e that also satisfies b exists or when $e \in \top$, $J_b(e)$ is set to E – one of the trivial consistent cuts. Here, we use E as a *sentinel* cut. We first show that $J_b(e)$ is uniquely defined. Let i_e be the predicate defined as follows:

$$C \text{ satisfies } i_e \quad \triangleq \quad e \in C$$

It can be proved that i_e is a regular predicate. Next, consider the predicate b_e defined as the conjunction of b and i_e. Since the class of regular predicates is closed under conjunction, b_e is also a regular predicate. The consistent cut $J_b(e)$ can now be reinterpreted as the least consistent cut that satisfies b_e. Since b_e is regular, the notion of least consistent cut that satisfies b_e is uniquely defined, thereby implying that $J_b(e)$ is uniquely defined. For purposes of computing the slice only, we assume that both trivial consistent cuts satisfy the given regular predicate. That is, $\{\emptyset, E\} \subseteq \mathcal{C}_b(E)$. The next lemma establishes that $J_b(e)$ is a join-irreducible element of $\mathcal{C}_b(E)$.

Lemma 4. *$J_b(e)$ is a join-irreducible element of the distributive lattice $\langle \mathcal{C}_b(E); \subseteq \rangle$.*

Proof. Suppose $J_b(e)$ can be expressed as the join (in our case, set union) of two consistent cuts in $\mathcal{C}_b(E)$, say C_1 and C_2. That is, $J_b(e) = C_1 \cup C_2$, where both C_1 and C_2 satisfy b. Our obligation is to show that either $J_b(e) = C_1$ or $J_b(e) = C_2$. Since $J_b(e)$ contains e, either C_1 or C_2 contains e. Without loss of generality, assume that e belongs to C_1. By definition of set union, $C_1 \subseteq J_b(e)$. Also, since C_1 is a consistent cut containing e that satisfies b, and $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq C_1$. Combining the two, $J_b(e) = C_1$. □

It is possible that $J_b(e)$s are not all distinct. Let $\mathcal{J}_b(E)$ denote the set $\{J_b(e) \mid e \in E\}$. Does $\mathcal{J}_b(e)$ capture all join-irreducible elements of $\mathcal{C}_b(E)$? The following lemma provides the answer.

Lemma 5. *Every consistent cut in $\mathcal{C}_b(E)$ can be expressed as the join of some subset of consistent cuts in $\mathcal{J}_b(E)$.*

Proof. Consider a consistent cut C in $\mathcal{C}_b(E)$. Let $D(C)$ be the consistent cut defined as follows:

$$D(C) = \bigcup_{e \in C} J_b(e)$$

We prove that $D(C)$ is actually equal to C. Since, by definition, $e \in J_b(e)$, each event in C is also present in $D(C)$. Thus $C \subseteq D(C)$. To prove that $D(C) \subseteq C$, consider an event $e \in C$. Since C is a consistent cut containing e that satisfies b and $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq C$. More precisely, for each event $e \in C$, $J_b(e) \subseteq C$. This implies that $D(C) \subseteq C$. □

From the previous two lemmas, it follows that $\mathcal{J}_b(E) = \mathcal{JI}(\mathcal{C}_b(E))$. Combining it with Birkhoff's Representation Theorem, we can deduce that:

Theorem 10. *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate b, the poset $\langle \mathcal{J}_b(E); \subseteq \rangle$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$.*

Next, in order to reduce the number of edges, rather than constructing a poset on the set of join-irreducible elements, we construct a directed graph with events as vertices and forming a strongly connected component out of each meta-event. It can be easily verified that:

Observation 1. *The directed graph $\mathcal{G}_b(E)$ with the set of vertices as E and an edge from an event e to an event f if and only if $J_b(e) \subseteq J_b(f)$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$.*

The *poset representation* of a slice – given by a partial order on a set of meta-events – is better suited for visualization purposes. On the other hand, the *graph representation* – given by a directed graph on a set of events – is more suited for slicing algorithms. In this paper, we primarily use the graph based representation. From the way the graph $\mathcal{G}_b(E)$ is constructed, clearly, two events e and f belong to the same strongly connected component of $\mathcal{G}_b(E)$ if and only if $J_b(e) = J_b(f)$. As a result, there is a one-to-one correspondence between the strongly connected components of $\mathcal{G}_b(E)$ and the join-irreducible elements of $\mathcal{C}_b(E)$.

Now, let $F_b(e)$ be a vector whose i^{th} entry denotes the earliest event f on process p_i such that $J_b(e) \subseteq J_b(f)$. Informally, $F_b(e)[i]$ is the earliest event f on p_i such that there is a path from e to f in the slice $\langle E, \rightarrow \rangle_b$. Using F_b, we construct a directed graph we call the *skeletal representation* of the slice and denote it by $\mathcal{S}_b(E)$. The graph $\mathcal{S}_b(E)$ has E as the set of vertices and the following edges:

1. for each event $e \not\in \top$, there is an edge from e to $succ(e)$, and
2. for each event e and every process p_i, there is an edge from e to $F_b(e)[i]$.

Example 6. Consider the slice depicted in Fig. 3d of the computation shown in Fig. 3a with respect to the predicate $reg\,(x_1 \vee x_2)$. Here, $J_b(f) = \{a, e, f\}$ and $J_b(c) = \{a, b, c, d, e, f, g\} = J_b(d)$. Also, $F_b(f) = [c, f]$ and $F_b(c) = [c, g] = F_b(d)$. □

In order to prove that $\mathcal{S}_b(E)$ faithfully captures the slice $\langle E, \rightarrow \rangle_b$, we prove the following two lemmas. The first lemma establishes that J_b is order-preserving.

Lemma 6 (J_b is order-preserving). *Given events e and f, $e \rightarrow f \;\Rightarrow\; J_b(e) \subseteq J_b(f)$.*

Proof. Consider $J_b(f)$. Since $e \rightarrow f$ and $f \in J_b(f)$, $e \in J_b(f)$. Therefore $J_b(f)$ is a consistent cut that contains e and satisfies b. Since $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq J_b(f)$. □

The second lemma shows that if $J_b(e) \subseteq J_b(f)$ then there is a path from event e to event f in $\mathcal{S}_b(E)$, and vice versa.

Lemma 7. *Given events e and f, $J_b(e) \subseteq J_b(f) \;\equiv\; (e, f) \in \mathcal{P}(\mathcal{S}_b(E))$.*

Proof. ($\Rightarrow$) Assume that $J_b(e) \subseteq J_b(f)$. Let $proc(f) = p_i$ and $g = F_b(e)[i]$. Since, by definition, g is the earliest event on p_i such that $J_b(e) \subseteq J_b(g)$, $g \overset{P}{\rightarrow} f$. This implies that $(g, f) \in \mathcal{P}(\mathcal{S}_b(E))$. Further, by construction, $(e, g) \in \mathcal{P}(\mathcal{S}_b(E))$. Thus $(e, f) \in \mathcal{P}(\mathcal{S}_b(E))$.

($\Leftarrow$) It suffices to show that for each edge (u, v) in $\mathcal{S}_b(E)$, $J_b(u) \subseteq J_b(v)$. If $v = succ(u)$ then $J_b(u) \subseteq J_b(v)$ follows from Lemma 6. If $v = F_b(u)[i]$, where $p_i = proc(v)$, then $J_b(u) \subseteq J_b(v)$ follows from the definition of $F_b(u)$. □

Finally, from Observation 1 and Lemma 7, we can conclude that:

Theorem 11. *$\mathcal{S}_b(E)$ is cut-equivalent to $\langle E, \rightarrow \rangle_b$.*

It is easy to see that $\mathcal{S}_b(E)$ has $O(|E|)$ vertices and $O(n|E|)$ edges. In the next section we provide efficient polynomial-time algorithms to compute $J_b(e)$ and $F_b(e)$ for each event e when b is a regular predicate.

7 Slicing for regular predicate

In this section, we discuss our results on slicing with respect to a regular predicate. They are discussed here separately from our results on slicing for a general predicate because, as proved in Sect. 5.1, the slice for a regular predicate is lean and therefore furnishes more information than the slice for a general predicate. First, we present an efficient $O(n^2|E|)$ algorithm to compute the slice for a regular predicate. The algorithm can be optimized for the case when a regular predicate can be decomposed into a conjunction of clauses, where each clause itself is a k-local regular predicate – a regular predicate that is also k-local – with small k. We also provide an *optimal* algorithm for a special class of regular predicates, namely conjunctive predicates. (An optimal slicing algorithm for certain monotonic channel predicates can be found elsewhere [MG03b].) Next, we show how a regular predicate can be monitored under various modalities [CM91,Gar02b,MG04, SUL00], specifically $possibly$, $invariant$ and $controllable$, using slicing. Finally, we demonstrate that results pertaining to consistent global checkpoints can be derived as special cases of slicing.

7.1 Computing the slice for regular predicate

In this section, given a computation $\langle E, \rightarrow \rangle$ and a regular predicate b, we describe an efficient $O(n^2|E|)$ algorithm to compute the slice $\langle E, \rightarrow \rangle_b$. In particular, we construct $\mathcal{S}_b(E)$ – the skeletal representation of $\langle E, \rightarrow \rangle_b$. To that end, it suffices to give an algorithm to compute $F_b(e)$ for each event e.

Our approach is to first compute $J_b(e)$ for each event e. Consider the predicate b_e defined in Sect. 6. Since b_e is a regular predicate, it is also a linear predicate. (A predicate is said to be linear if, given two consistent cuts that satisfy the predicate, the consistent cut given by their set intersection also satisfies the predicate.) Chase and Garg [Gar02b] give an efficient algorithm to find the least consistent cut that satisfies a linear predicate. Their algorithm is based on the *linearity property* which is defined as follows:

Definition 6 (linearity property [CG98]). *A predicate satisfies the* linearity property *if, given a consistent cut that does not satisfy the predicate, there exists an event in its frontier, called the* forbidden event*, such that every consistent cut containing the given consistent cut either does not satisfy the predicate or does not pass through the forbidden event. Formally, given a computation $\langle E, \rightarrow \rangle$, a linear predicate b and a consistent cut C,*

$$\begin{aligned} &C \text{ does not satisfy } b \;\Rightarrow \\ &\quad \langle \exists f : f \in frontier(C) : \\ &\qquad \langle \forall\, D : D \supseteq C : D \text{ satisfies } b \;\Rightarrow\; succ(f) \in D \rangle\rangle \end{aligned}$$

We denote the forbidden event of C with respect to b by $forbidden(b, C)$.

Figure 4 describes the algorithm ComputeJ to determine $J_b(e)$ for each event e on process p_i, using the linearity property. The algorithm scans the computation once from left to right. Only a single scan is sufficient because, from Lemma 6,

```
Input: (1) a computation $\langle E, \rightarrow \rangle$, (2) a regular predicate $b$, and (3) a process $p_i$
Output: $J_b(e)$ for each event $e$ on $p_i$

1   $C$ := $\bot$;
2   for each event $e$ on $p_i$ do                      // visited in the order given by $\xrightarrow{P}$
3       $done$ := false;
4       if $C = E$ then $done$ := true;
5       while not($done$) do
6           if there exist events $f$ and $g$ in $frontier(C)$
              such that $succ(f) \rightarrow g$ then      // $C$ is not a consistent cut
7               $C$ := $C \cup \{succ(f)\}$;               // advance beyond $f$
            else                                          // $C$ is a consistent cut
8               if ($C = E$) or ($C$ satisfies $b_e$) then $done$ := true;
                else
9                   $f$ := $forbidden(b_e, C)$;           // invoke the linearity property
10                  $C$ := $C \cup \{succ(f)\}$;           // advance beyond $f$
                endif;
            endif;
        endwhile;
11      $J_b(e)$ := $C$;
    endfor;
```

Fig. 4. The algorithm ComputeJ to determine $J_b(e)$ for each event e on process p_i

once we have computed $J_b(e)$, we do not need to start all over again to determine $J_b(succ(e))$ but can rather continue on from $J_b(e)$ itself. The algorithm basically adds events one-by-one to the cut constructed so far until either all the events are exhausted or the desired consistent cut is reached.

The time-complexity analysis of the algorithm ComputeJ is as follows. Each iteration of the while loop at line 5 has $O(n)$ time-complexity assuming that the time-complexity of invoking $forbidden_{b_e}$ at line 9 once is $O(n)$. Moreover, the while loop is executed at most $O(|E|)$ times because in each iteration either we succeed in finding the required consistent cut or we add a new event to C. Since there are at most $|E|$ events in the computation, the while loop cannot be executed more than $O(|E|)$ times. Thus the overall time-complexity of the algorithm ComputeJ is $O(n|E|)$ implying that $J_b(e)$ for each event e can be computed in $O(n^2|E|)$ time.

Finally, we give an algorithm to compute $F_b(e)$ for each event e provided $J_b(e)$ for each event e is given to us. We first establish a lemma similar to Lemma 6 for F_b. The lemma allows us to compute the i^{th} entry of $F_b(e)$ for all events e on process p_x in a *single* scan of events on process p_i from left to right.

Lemma 8 (F_b is order-preserving). *For events e and f, and a process p_i,*

$$e \rightarrow f \;\Rightarrow\; F_b(e)[i] \xrightarrow{P} F_b(f)[i]$$

Proof. Assume that $e \rightarrow f$. Let $g = F_b(e)[i]$ and $h = F_b(f)[i]$. Note that $proc(g) = proc(h) = p_i$. By definition of $F_b(f)$, $J_b(f) \subseteq J_b(h)$. Since, from Lemma 6, $J_b(e) \subseteq J_b(f)$, $J_b(e) \subseteq J_b(h)$. Again, by definition of $F_b(e)$, g is the earliest event on p_i such that $J_b(e) \subseteq J_b(g)$. Therefore $g \xrightarrow{P} h$. □

Figure 5 depicts the algorithm ComputeF to determine $F_b(e)$ for each event e on process p_i. The algorithm is self-explanatory and its time-complexity analysis is as follows. Let E_i denote the set of events on process p_i. The outer for loop at line 1 is executed exactly n times. For the i^{th} iteration of the outer for loop, the while loop at line 4 is executed at most $O(|E_i| + |E_x|)$ times. Each iteration of the while loop has $O(1)$ time-complexity because whether $J_b(e) \subseteq J_b(f)$ can be ascertained by performing only a single comparison, namely testing whether e is contained in $J_b(f)$. More precisely, $J_b(e) \subseteq J_b(f)$ if and only if $e \in J_b(f)$. The reason is as follows. Since $e \in J_b(e)$, if $J_b(e) \subseteq J_b(f)$, then $e \in J_b(f)$. Also, if $e \in J_b(f)$, then $J_b(f)$ is a consistent cut that contains e and satisfies b. Since $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq J_b(f)$. Combining the two, we obtain the desired equivalence. The overall time-complexity of the algorithm ComputeF is, therefore, $O(|E| + n|E_x|)$. Summing over all possible values for x, $F_b(e)$ for all events e on *all* processes can be determined in $O(n|E|)$ time. The overall algorithm is summarized in Fig. 6.

7.2 Optimizing for the special case: computing the slice for decomposable regular predicate

In this section, we explore the possibility of a faster algorithm for the case when a regular predicate can be expressed as a conjunction of clauses where each clause is again a regular predicate but depends on variables of only a small number of processes. For example, consider the regular predicate "counters on all processes are approximately synchronized", denoted by b_{sync}, which can be expressed formally as:

$$b_{sync} \;\triangleq\; \bigwedge_{1 \leqslant i,j \leqslant n} (|counter_i - counter_j| \leqslant \triangle_{ij})$$

where each $counter_i$ is a monotonically non-decreasing variable on process p_i. In this example, each clause depends on

```
Input: (1) a computation $\langle E, \rightarrow \rangle$, (2) $J_b(e)$ for each event $e$, and (3) a process $p_x$
Output: $F_b(e)$ for each event $e$ on $p_x$

1  for each process $p_i$ do
2      $f$ := $\bot_i$;
3      for each event $e$ on $p_x$ do                // visited in the order given by $\xrightarrow{P}$
4          while $J_b(e) \not\subseteq J_b(f)$ do $f$ := $succ(f)$; endwhile;
5          $F_b(e)[i] := f$;
       endfor;
   endfor;
```

Fig. 5. The algorithm ComputeF to determine $F_b(e)$ for all events e on process p_x

```
Input: (1) a computation $\langle E, \rightarrow \rangle$, and (2) a regular predicate $b$
Output: the slice $\langle E, \rightarrow \rangle_b$

1  compute $J_b(e)$ for each event $e$ using the algorithm ComputeJ;
2  compute $F_b(e)$ for each event $e$ using the algorithm ComputeF;
3  construct $\mathcal{S}_b(E)$ the skeletal representation of $\langle E, \rightarrow \rangle_b$;
4  output $\mathcal{S}_b(E)$;
```

Fig. 6. The algorithm SliceForRegular to compute the slice for a regular predicate

variables of at most two processes and is therefore 2-local. Using the algorithm discussed in this section, it is possible to compute the slice for b_{sync} in $O(n|E|)$ time – a factor of n faster than using the algorithm SliceForRegular. We describe the algorithm in two steps. In the first step, we give a fast algorithm to compute the slice for each clause. In the second step, we describe how to combine slices for all clauses together in an efficient manner to obtain the slice for the given regular predicate.

7.2.1 Step 1

Consider a computation $\langle E, \rightarrow \rangle$ and a k-local regular predicate b. Let Q denote the subset of processes whose variables b depends on. *Without loss of generality, assume that $\rightarrow$ is a transitive relation.* We denote the projection of E on Q by $E(Q)$ and that of $\rightarrow$ on $Q \times Q$ by $\rightarrow(Q)$. Thus the projection of the computation $\langle E, \rightarrow \rangle$ on Q is given by $\langle E(Q), \rightarrow(Q) \rangle$.

We first show that the slice $\langle E, \rightarrow \rangle_b$ of the computation $\langle E, \rightarrow \rangle$ can be recovered exactly from the slice $\langle E(Q), \rightarrow(Q) \rangle_b$ of the projected computation $\langle E(Q), \rightarrow(Q) \rangle$. To that end, we extend the definition of $F_b(e)$ and define $F_b(e, Q)$ to be a vector whose i^{th} entry represents the earliest event on process p_i that is reachable from e in the slice $\langle E(Q), \rightarrow(Q) \rangle_b$. Thus $F_b(e) = F_b(e, P)$, where P is the entire set of processes. We next define $K_b(e)$ as follows:

$$K_b(e)[i] = \begin{cases} F_b(e, Q)[i] & : \ (e \in E(Q)) \wedge (p_i \in Q) \\ F(e)[i] & : \ \text{otherwise} \end{cases}$$

We claim that it suffices to know $K_b(e)$ for each event e to be able to compute the slice $\langle E, \rightarrow \rangle_b$. We build a graph $\mathcal{H}_b(E)$ in a similar fashion as the skeletal representation $\mathcal{S}_b(E)$ of $\langle E, \rightarrow \rangle_b$ except that we use K_b instead of F_b in its construction. The next lemma proves that every path in $\mathcal{H}_b(E)$ is also a path in $\mathcal{S}_b(E)$.

Lemma 9. *For each event e and process p_i, $F_b(e)[i] \xrightarrow{P} K_b(e)[i]$.*

Proof. Every consistent cut of the slice $\langle E, \rightarrow \rangle_b$ is a consistent cut of the computation $\langle E, \rightarrow \rangle$ as well. Therefore, by Lemma 1, every connected pair in $\langle E, \rightarrow \rangle$ is also a connected pair in $\langle E, \rightarrow \rangle_b$. This in turn implies that, for each event e and process p_i, $F_b(e)[i] \xrightarrow{P} F(e)[i]$. Our obligation is to prove that $F_b(e)[i] \xrightarrow{P} F_b(e, Q)[i]$ when $e \in E(Q)$ and $p_i \in Q$.

Consider an event $e \in E(Q)$ and process $p_i \in Q$. For convenience, let $f = F_b(e, Q)[i]$. Let C be the *least* consistent cut of the slice $\langle E, \rightarrow \rangle_b$ that contains f. Clearly, C is also a consistent cut of the computation $\langle E, \rightarrow \rangle$. We have,

$$\begin{aligned}
& \{ \text{ definition of projection } \} \\
& C(Q) \text{ is a consistent cut of } \langle E(Q), \rightarrow(Q) \rangle \\
\equiv \ & \{ \text{ predicate calculus } \} \\
& (C(Q) \text{ is a consistent cut of } \langle E(Q), \rightarrow(Q) \rangle) \wedge \\
& \Big((C = \top) \vee (C \neq \top) \Big) \\
\Rightarrow \ & \begin{Bmatrix} \text{in case } C \neq \top, \text{ by definition, } C \text{ satisfies } b \\ \text{and } b \text{ depends only on variables of } Q \end{Bmatrix} \\
& C(Q) \text{ is a consistent cut of } \langle E(Q), \rightarrow(Q) \rangle_b \\
\Rightarrow \ & \{ f \in C, proc(f) = p_i, \text{ and } p_i \in Q \ \} \\
& (f \in C(Q)) \wedge
\end{aligned}$$

```
Input: (1) a computation $\langle E, \to \rangle$,
       (2) a $k$-local regular predicate $b$ that depends only on variables of $Q \subseteq P$ with $|Q| \leqslant k$
Output: the slice $\langle E, \to \rangle_b$

1  compute $F(e)$ for each event $e$;
2  compute the projection of $\langle E, \to \rangle$ onto $Q$, say $\langle E(Q), \to(Q) \rangle$;
3  compute the slice of $\langle E(Q), \to(Q) \rangle$ with respect to $b$, say $\langle E(Q), \to(Q) \rangle_b$, using
       the algorithm SliceForRegular;
   Also, compute $F_b(e, Q)$ for each event $e$;
4  compute $K_b(e)$ for each event $e$ as follows:
       $K_b(e)[i] = \begin{cases} F_b(e,Q)[i] & : (e \in E(Q)) \wedge (p_i \in Q) \\ F(e)[i] & : \text{otherwise} \end{cases}$

5  construct the directed graph $\mathcal{H}_b(E)$ with $E$ as its set of vertices and edges as follows:
       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 $K_b(e)[i]$.
6  output $\mathcal{H}_b(E)$;
```

Fig. 7. The algorithm SliceForKLocalRegular to compute the slice for a k-local regular predicate

$(C(Q)$ is a consistent cut of $\langle E(Q), \to(Q) \rangle_b)$

$\Rightarrow$ { using definition of f, which is $F_b(e, Q)[i]$ }

$(f \in C(Q)) \wedge$

(there is a path from e to f in $\langle E(Q), \to(Q) \rangle_b) \wedge$

$(C(Q)$ is a consistent cut of $\langle E(Q), \to(Q) \rangle_b)$

$\Rightarrow$ { definition of consistent cut }

$(f \in C(Q)) \wedge (e \in C(Q))$

$\Rightarrow$ { $\{e, f\} \subseteq C(Q)$ implies $\{e, f\} \subseteq C$ }

$(f \in C) \wedge (e \in C)$

$\equiv$ { definition of C }

there is a path from e to f in $\langle E, \to \rangle_b$

$\equiv$ { definition of $F_b(e)[i]$ }

$F_b(e)[i] \xrightarrow{P} f$

Thus $F_b(e)[i] \xrightarrow{P} F_b(e, Q)[i]$. □

We now prove the converse, that is, every connected pair in $\mathcal{S}_b(E)$ is also a connected pair in $\mathcal{H}_b(E)$. To that end, by virtue of Lemma 1, it suffices to show that every consistent cut of $\mathcal{H}_b(E)$ is also a consistent cut of $\mathcal{S}_b(E)$ or, equivalently, every consistent cut of $\mathcal{H}_b(E)$ satisfies b.

Lemma 10. *Every (non-trivial) consistent cut of $\mathcal{H}_b(E)$ satisfies b.*

The proof is in the appendix. Finally, the previous two lemmas can be combined to give the following theorem:

Theorem 12. *$\mathcal{H}_b(E)$ is cut-equivalent to $\mathcal{S}_b(E)$.*

Observe that the two graphs $\mathcal{H}_b(E)$ and $\mathcal{S}_b(E)$ may actually be different. However, Theorem 12 ensures that the two will be cut-equivalent, thereby implying that $\mathcal{H}_b(E)$ captures the slice faithfully. Figure 7 describes the algorithm SliceForKLocalRegular for computing the slice for a k-local regular predicate. We assume that the computation is given to us as n queues of events – one for each process. Further, the Fidge/Mattern's timestamp $ts(e)$ for each event e [Mat89, Fid91] is also available to us, using which $J(e)$ can be computed easily. The algorithm ComputeF can be used to computer $F(e)$ for each event e in $O(n|E|)$. The projection of the computation on Q can then be computed at line 2 in a straightforward fashion – by simply ignoring events on other processes. The slice of the projected computation can be computed at line 3 in $O(|Q|^2|E(Q)|)$ time. The vector $K_b(e)$ for each event e can be determined at line 4 in $O(n|E|)$ time. Finally, the graph $\mathcal{H}_b(E)$ can be constructed at line 5 in $O(n|E|)$ time. Thus the overall time-complexity of the algorithm is $O(|Q|^2|E(Q)| + n|E|)$. If $|Q|$ is small, say at most $\sqrt{n}$, then the time-complexity of the algorithm is $O(n|E|)$ – a factor of n faster than computing the slice directly using the algorithm SliceForRegular.

A natural question to ask is: "Can this technique of taking a projection of a computation on a subset of processes, then computing the slice of the projection and finally mapping the slice back to the original set of processes be used for a non-regular predicate as well?" The answer is no in general as the following example demonstrates.

Example 7. Consider the computation depicted in Fig. 8a involving three processes p_1, p_2 and p_3. Let x_1 and x_2 be boolean variables on processes p_1 and p_2, respectively. In the figure, the solid events, namely e_3 and f_3, satisfy the respective boolean variable. The slice of the computation for the (non-regular) predicate $x_1 \vee x_2$ is depicted in Fig. 8b. Figure 8c displays the projection of the computation on processes on which the predicate $x_1 \vee x_2$ depends, namely p_1 and p_2. The slice of the projected computation is shown in Fig. 8d and its mapping back to the original set of processes is depicted in Fig. 8e. As it can be seen, the slice shown in Fig. 8e computed using the algorithm SliceForKLocalRegular is

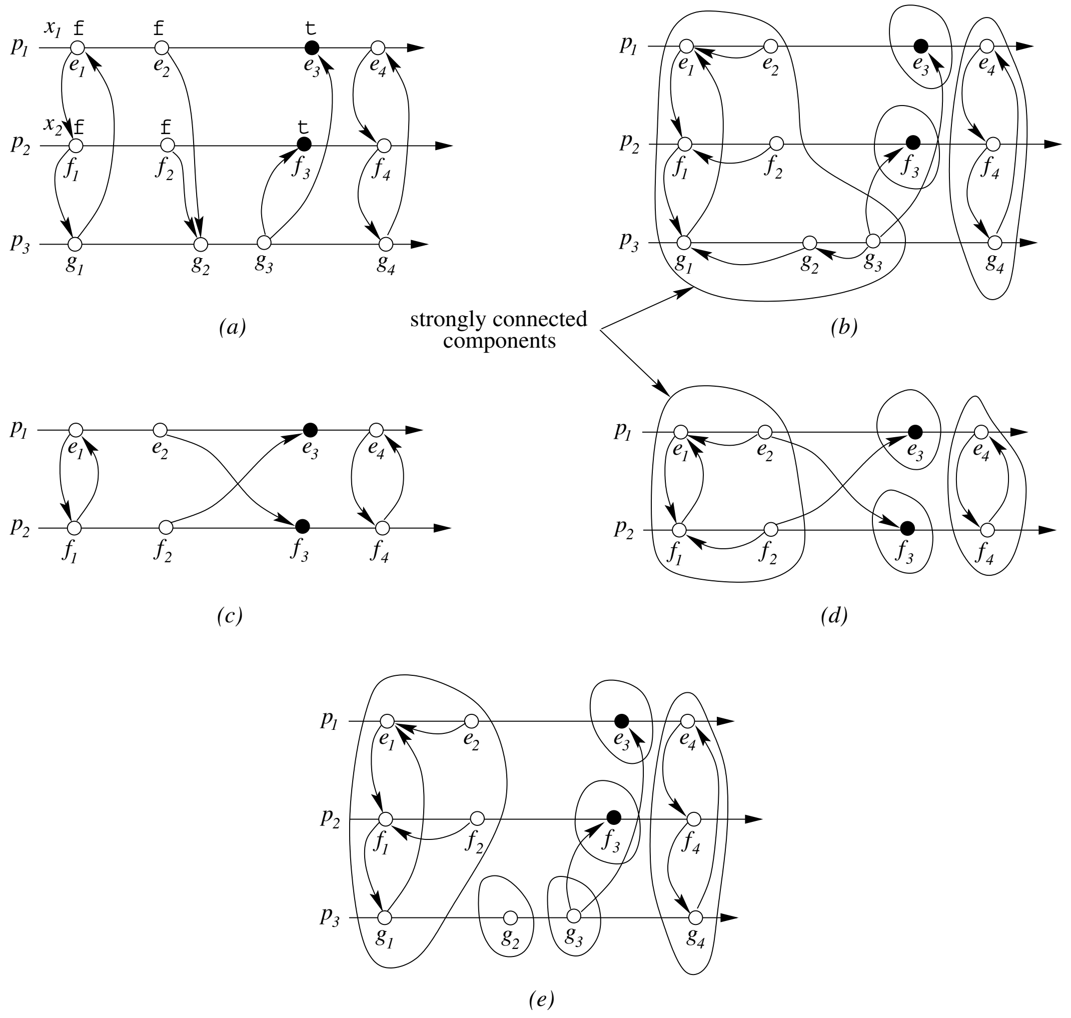


Fig. 8. **a** A computation, **b** its slice with respect to the predicate $x_1 \vee x_2$, **c** its projection on processes p_1 and p_2, **d** the slice of the projected computation with respect to the predicate $x_1 \vee x_2$, and **e** the slice computed in **d** mapped to the original set of processes

different from the actual slice shown in Fig. 8b. For instance, events g_2 and g_3 belong to the same meta-event in the actual slice but not in the slice computed using the algorithm SliceForKLocalRegular. The reason for this difference is as follows. Since the predicate $x_1 \vee x_2$ is non-regular, the slice of the projected computation shown in Fig. 8d contains the consistent cut $X = \{e_1, e_2, f_1, f_2\}$ which does not satisfy $x_1 \vee x_2$ but has to be included anyway so as to complete the sublattice. Now, on mapping this slice back to the original set of processes, the resulting slice depicted in Fig. 8e will contain *all* consistent cuts of the original computation whose projection on $\{p_1, p_2\}$ is X. There are three such consistent cuts, namely $X \cup \{g_1\}$, $X \cup \{g_1, g_2\}$ and $X \cup \{g_1, g_2, g_3\}$. However, only one of these consistent cuts, given by $X \cup \{g_1, g_2, g_3\}$, is required to complete the sublattice for the actual slice. □

It can be verified that the algorithm SliceForKLocalRegular when used for computing the slice for a non-regular predicate, in general, produces a slice bigger than the actual slice. Thus it yields a fast way to compute an approximate slice for a non-regular predicate (*e.g.*, linear predicate).

7.2.2 Step 2

Now, consider a decomposable regular predicate b expressible as conjunction of k-local regular predicates $b^{(j)}$, where $1 \leqslant j \leqslant m$. Let Q^j denote the subset of processes whose variable(s) the j^{th} clause $b^{(j)}$ depends on. For a process p_i, we define $clauses_i$ as the set of those clauses that depend on some variable of p_i, that is, $clauses_i \triangleq \{ b^{(j)} \mid p_i \in Q^j \}$. Also, let $s = \max_{1 \leqslant i \leqslant n} |clauses_i|$. For example, for the regular predicate b_{sync}, $k = 2$ and $s = n$.

To obtain the slice with respect to b, we can proceed as follows. We first compute the slice for each clause using the algorithm SliceForKLocalRegular. This will give us $K_{b^{(j)}}(e)$ for each clause $b^{(j)}$ and event e. Then, for each event e and process p_i, we simply set $K_b(e)[i]$ to the earliest event in the following set:

$$\{K_{b^{(1)}}(e)[i], K_{b^{(2)}}(e)[i], \ldots, K_{b^{(m)}}(e)[i]\}$$

However, this approach has time-complexity of $O((nm + k^2 s)|E|)$.

To reduce the time-complexity, after computing the slice $\langle E(Q^j), \rightarrow(Q^j)\rangle_{b^{(j)}}$ for each clause $b^{(j)}$, we compute K_b directly without first computing $K_{b^{(j)}}$ for each clause $b^{(j)}$. The

```
for each event $e \in E$ do
    $K_b(e)$ := $F(e)$;
endfor;

for each conjunct $b^{(j)}$ do
    for each event $e \in E(Q^j)$ do
        for each process $p_i \in Q^j$ do
            $K_b(e)[i]$ := $\min\{\ K_b(e)[i],\ F_{b^{(j)}}(e, Q^j)[i]\ \}$;
        endfor;
    endfor;
endfor;
```

Fig. 9. Computing $K_b(e)$ for each event e

algorithm is shown in Fig. 9. Intuitively, among all the slices for the clauses belonging to $clauses_i$, $K_b(e)[i]$ is the earliest event on p_i that is reachable from e. Formally,

$$K_b(e)[i] = \min_{b^{(j)} \in clauses_i} F_{b^{(j)}}(e, Q^j)[i]$$

It can be verified that the graph $\mathcal{H}_b(E)$ then constructed using $K_b(e)$ for each event e – in a similar fashion as in Step 1 – is actually cut-equivalent to the slice $\langle E, \rightarrow\rangle_b$. The proof is similar to that in Step 1 and has been omitted. The overall time-complexity of the algorithm is given by:

$$\begin{aligned}
& O(n|E|) + \sum_{j=1}^{m} O(|Q^j|^2 |E(Q^j)|) \\
= \ & \left\{ \begin{array}{l} \text{each } b^{(j)} \text{ is a } k\text{-local predicate, therefore} \\ |Q^j| \leqslant k \end{array} \right\} \\
& O(n|E|) + O(k^2 \sum_{j=1}^{m} |E(Q^j)|) \\
= \ & \{ \text{ simplifying } \} \\
& O(n|E| + k^2 s\, |E|) = O((n + k^2 s)|E|)
\end{aligned}$$

In case k is $O(1)$ and s is $O(n)$, as is the case with b_{sync}, the overall time-complexity is $O(n|E|)$, which is a factor of n less than computing the slice directly using the algorithm SliceForRegular.

7.3 Optimal algorithm for the special case

For special cases of regular predicates, namely conjunctive predicates and certain monotonic channel predicates, it is possible to give an $O(|E|)$ optimal algorithm for computing the slice. We only present the slicing algorithm for the class of conjunctive predicates here. The slicing algorithm for the class of monotonic channel predicates can be found elsewhere [MG03b].

Consider a computation $\langle E, \rightarrow\rangle$ and a conjunctive predicate b. The first step is to partition events on each process into true events and false events. Then we construct a graph $\mathcal{H}_b(E)$ with vertices as the events in E and the following edges:

1. from an event, that is not a final event, to its successor,
2. from a send event to the corresponding receive event, and
3. from the successor of a false event to the false event.

For the purpose of building the graph, we assume that all final events are true events. Thus every false event has a successor. The first two types of edges ensure that the Lamport's happened-before relation [Lam78] is contained in $\mathcal{P}(\mathcal{H}_b(E))$. Consider the computation depicted in Fig. 10a and the conjunctive predicate $x_1 \wedge x_2$. The corresponding graph constructed as described is shown in Fig. 10b. We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy the conjunctive predicate.

Lemma 11. *Every (non-trivial) consistent cut of $\mathcal{H}_b(E)$ satisfies b.*

Proof. It is sufficient to prove that no consistent cut of $\mathcal{H}_b(E)$ contains a false event in its frontier. Consider a consistent cut C of $\mathcal{H}_b(E)$. Assume, on the contrary, that C contains a false event, say e, in its frontier. Since every false event has a successor, by construction, there is an edge from the successor of e, say f, to e. Therefore f also belongs to C. This contradicts the fact that e is the last event on its process to be contained in C. □

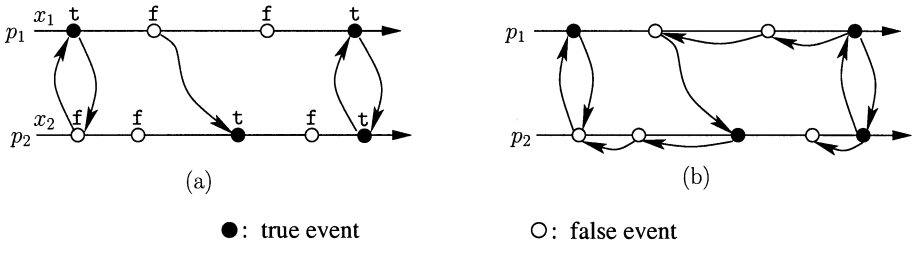


Fig. 10. a A computation, and **b** its slice with respect to the conjunctive predicate $x_1 \wedge x_2$

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

Lemma 12. *Every consistent cut of $\langle E, \rightarrow\rangle$ that satisfies b is a consistent cut of $\mathcal{H}_b(E)$.*

Proof. Consider a consistent cut C of $\langle E, \rightarrow\rangle$ that satisfies b. Assume, on the contrary, that C is not a consistent cut of $\mathcal{H}_b(E)$. Thus there exist events e and f such that there is an edge from e to f in $\mathcal{H}_b(E)$, f belongs to C but e does not. Since C is a consistent cut of $\langle E, \rightarrow\rangle$, the edge from e to f could only be of type (3). (The other two types of edges are present in $\langle E, \rightarrow\rangle$ as well.) Equivalently, e and f occur on the same process, e is the successor of f, and f is a false event. Again, since f is contained in C but its successor e is not, f belongs to the frontier of C. However, C satisfies b and hence cannot contain any false event in its frontier. □

From the previous two lemmas, it follows that:

Theorem 13. *$\mathcal{H}_b(E)$ is cut-equivalent to $\langle E, \rightarrow\rangle_b$.*

It is easy to see that the graph $\mathcal{H}_b(E)$ has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in $O(|E|)$ time. Thus the algorithm has $O(|E|)$ overall time-complexity. It also gives us an $O(|E|)$ algorithm to evaluate $possibly\colon b$ when b is a conjunctive predicate.

7.4 Applications of slicing

In this section, we show that slicing can be used to solve other problems in distributed systems.

7.4.1 Monitoring Regular Predicate under Various Modalities

A predicate can be monitored under four modalities, namely $possibly$, $definitely$, $invariant$ and $controllable$ [CM91,Gar02b,MG04,SUL00]. A predicate is *possibly* true in a computation if there exists a consistent cut of the computation that satisfies the predicate. On the other hand, a predicate *definitely* holds in a computation if it eventually becomes true in all runs of the computation (a *run* is a path in the lattice of consistent cuts starting from the initial consistent cut and ending at the final consistent cut). The modalities $invariant$ and $controllable$ are duals of the modalities $possibly$ and $definitely$, respectively. Specifically, a predicate is *invariant* in a computation if the predicate stays true throughout every run of the computation. Finally, a predicate is *controllable* in a computation if there exists a run of the computation such that the predicate stays true throughout the run. Monitoring has applications in the areas of testing and debugging and software fault tolerance of distributed programs.

We show how to monitor a regular predicate under $possibly\colon b$, $invariant\colon b$ and $controllable\colon b$ modalities using slicing. Given a directed graph G, let $\mathsf{scc}(G)$ denote the number of strongly connected components of G.

Theorem 14. *A regular predicate is*

1. **possibly** *true in a computation if and only if the slice of the computation with respect to the predicate has at least one non-trivial consistent cut, that is, it has at least two strongly connected components. Formally,*
$$possibly\colon b \;\equiv\; \mathsf{scc}(\langle E, \rightarrow\rangle_b) \geqslant 2$$
2. **invariant** *in a computation if and only if the slice of the computation with respect to the predicate is cut-equivalent to the computation. Formally,*
$$invariant\colon b \;\equiv\; \langle E, \rightarrow\rangle_b \cong \langle E, \rightarrow\rangle$$
3. **controllable** *in a computation if and only if the slice of the computation with respect to the predicate has the same number of strongly connected components as the computation. Formally,*
$$controllable\colon b \;\equiv\; \mathsf{scc}(\langle E, \rightarrow\rangle_b) = \mathsf{scc}(\langle E, \rightarrow\rangle)$$

The proof of the theorem can be found in the appendix. We do not yet know how to monitor a regular predicate under $definitely$ modality.

7.4.2 Zig-zag consistency theorem: a special case of slicing

We now show how slicing relates to some of the well-known results in checkpointing. Consider a conjunctive predicate such that the local predicate for an event on a process is true if and only if the event corresponds to a local checkpoint. It can be verified that there is a *zigzag path* [NX95,Wan97] from a local checkpoint c to a local checkpoint c' in a computation if and only if there is a path from $succ(c)$, if it exists, to c' in the corresponding slice – which can be ascertained by comparing $J_b(succ(c))$ and $J_b(c')$. An alternative formulation of the consistency theorem in [NX95] can thus be obtained as follows:

Theorem 15. *A set of local checkpoints can belong to the same consistent global snapshot if and only if the local checkpoints in the set are mutually consistent (including with itself) in the slice.*

Moreover, the R-graph (rollback-dependency graph) [Wan97] is path-equivalent to the slice when each contiguous sequence of false events on a process is merged with the nearest true event that occurs later on the process. The minimum consistent global checkpoint that contains a set of local checkpoints [Wan97] can be computed by taking the set union of J_b's for each local checkpoint in the set. The maximum consistent global checkpoint can be similarly obtained by using the dual of J_b.

8 Slicing for non-regular predicate

In this section, we describe our results on slicing for general predicates. We first prove that it is in general NP-hard to compute the slice for an arbitrary predicate. Nonetheless, polynomial-time algorithms can be developed for certain special classes of predicates. In particular, we provide efficient algorithm to compute the slice for a linear predicate and its dual – a post-linear predicate [Gar02b]. We next present an efficient algorithm for composing two slices efficiently; composition can be done with respect to meet or join operator as explained later. Slice composition can be used to compute the slice for a predicate in DNF (disjunctive normal form). We further provide three more applications of composition. First, we demonstrate how composition can be employed to compute the slice for a *co-regular predicate* – complement of a regular predicate – in polynomial-time. Second, using composition, we derive a polynomial-time algorithm to the compute the slice for a k*-local predicate* for constant k. Lastly, we employ slice composition to compute an *approximate slice* – in polynomial-time – for a predicate composed from linear predicates, post-linear predicates, co-regular predicates and k-local predicates, for constant k, using $\wedge$ and $\vee$ operators.

8.1 NP-hardness result

It is evident from the definition of slice that the following is true:

Observation 2. *The necessary and sufficient condition for the slice of a computation with respect to a predicate to be non-empty is that there exists a consistent cut of the computation that satisfies the predicate.*

However, finding out whether some consistent cut of the computation satisfies a predicate is an NP-complete problem [Gar02b]. Thus it is in general NP-complete to determine whether the slice for a predicate is non-empty. This further implies that computing the slice for an arbitrary predicate is an NP-hard problem. From the results of [MG01b], it follows that this is the case even when the predicate is in 2-CNF and no two clauses contain variables from the same process.

8.2 Computing the slice for linear or post-linear predicate

Recall that a predicate is linear if given two consistent cuts that satisfy the predicate, the cut given by their set intersection also satisfies the predicate [Gar02b]. A post-linear predicate can be defined dually [Gar02b]. It turns out that the algorithm SliceForRegular described in Sect. 7.1 for computing the slice for a regular predicate can also be used to compute the slice for a linear predicate. The proof is given in the appendix. For a post-linear predicate, however, a slightly different version of the algorithm based on the notion of meet-irreducible element is applicable.

8.3 Composing two slices

Given two slices, slice composition can be used to either compute the smallest slice that contains all consistent cuts common to both slices – composing with respect to meet – or compute the smallest slice that contains consistent cuts of both slices – composing with respect to join. In other words, given slices $\langle E, \rightarrow \rangle_{b_1}$ and $\langle E, \rightarrow \rangle_{b_2}$, where b_1 and b_2 are regular predicates, composition can be used to compute the slice $\langle E, \rightarrow \rangle_b$, where b is either $b_1 \sqcap b_2 = b_1 \wedge b_2$ or $b_1 \sqcup b_2 = reg\,(b_1 \vee b_2)$. Slice composition enables us to compute the slice for an arbitrary boolean expression of local predicates – by rewriting it in DNF – although it may require exponential time in the worst case.

8.3.1 Composing with respect to Meet: $b \equiv b_1 \sqcap b_2 \equiv b_1 \wedge b_2$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if and only if the cut satisfies b_1 as well as b_2. Given an event e, let $F_{\min}(e)$ denote the vector obtained by taking componentwise minimum of $F_{b_1}(e)$ and $F_{b_2}(e)$. We first prove that no component of $F_{\min}(e)$ is less than (or occurs before) the corresponding component of $F_b(e)$.

Lemma 13. *For each event* e *and process* p_i,

$$F_b(e)[i] \overset{P}{\rightarrow} F_{\min}(e)[i].$$

Proof. For convenience, let $f = F_{b_1}(e)[i]$. Let C be the *least* consistent cut of the slice $\langle E, \rightarrow \rangle_b$ that contains f. Clearly, C is also a consistent cut of the computation $\langle E, \rightarrow \rangle$. We have,

$$\begin{array}{cl}
 & (C \text{ is a consistent cut of } \langle E, \rightarrow \rangle) \wedge \\
 & \Big((C = \top) \vee (C \neq \top)\Big) \\
\Rightarrow & \left\{ \begin{array}{l} \text{in case } C \neq \top, \text{ by definition, } C \text{ satisfies } b \\ \text{and therefore satisfies } b_1 \text{ as well} \end{array} \right\} \\
 & C \text{ is a consistent cut of } \langle E, \rightarrow \rangle_{b_1} \\
\Rightarrow & \{ \text{ by definition, } C \text{ contains } f \ \} \\
 & (f \in C) \wedge (C \text{ is a consistent cut of } \langle E, \rightarrow \rangle_{b_1}) \\
\Rightarrow & \{ \text{ using definition of } f, \text{ which is } F_{b_1}(e)[i] \ \} \\
 & (f \in C) \wedge \\
 & (\text{there is a path from } e \text{ to } f \text{ in } \langle E, \rightarrow \rangle_{b_1}) \wedge \\
 & (C \text{ is a consistent cut of } \langle E, \rightarrow \rangle_{b_1}) \\
\Rightarrow & \{ \text{ definition of consistent cut } \} \\
 & (f \in C) \wedge (e \in C) \\
\equiv & \left\{ \begin{array}{l} \text{by definition, } C \text{ is the } \textit{least} \text{ consistent cut of} \\ \langle E, \rightarrow \rangle_b \text{ that contains } f \end{array} \right\} \\
 & \text{there is a path from } e \text{ to } f \text{ in } \langle E, \rightarrow \rangle_b \\
\equiv & \{ \text{ definition of } F_b(e)[i] \ \} \\
 & F_b(e)[i] \overset{P}{\rightarrow} f
\end{array}$$

Therefore $F_b(e)[i] \overset{P}{\rightarrow} F_{b_1}(e)[i]$. Likewise, $F_b(e)[i] \overset{P}{\rightarrow} F_{b_2}(e)[i]$. □

We now construct a directed graph $\mathcal{S}_{\min}(E)$ that is similar to $\mathcal{S}_b(E)$ except that we use $F_{\min}$ instead of F_b in its construction. We show that $\mathcal{S}_{\min}(E)$ is in fact cut-equivalent to $\mathcal{S}_b(E)$.

Theorem 16. $\mathcal{S}_{\min}(E)$ *is cut-equivalent to* $\mathcal{S}_b(E)$.

Proof. We have,

$$
\begin{aligned}
&\{ \text{ definition of } F_{\min} \} \\
&\Big(\mathcal{P}(\mathcal{S}_{b_1}(E)) \subseteq \mathcal{P}(\mathcal{S}_{\min}(E))\Big) \wedge \\
&\Big(\mathcal{P}(\mathcal{S}_{b_2}(E)) \subseteq \mathcal{P}(\mathcal{S}_{\min}(E))\Big) \\
\equiv\ &\{ \text{ using Lemma 1 } \} \\
&\Big(\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_{b_1}(E))\Big) \wedge \\
&\Big(\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_{b_2}(E))\Big) \\
\equiv\ &\{ \text{ set calculus } \} \\
&\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \Big(\mathcal{C}(\mathcal{S}_{b_1}(E)) \cap \mathcal{C}(\mathcal{S}_{b_2}(E))\Big) \\
\equiv\ &\{ b \equiv b_1 \wedge b_2 \} \\
&\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_b(E))
\end{aligned}
$$

Also, we have,

$$
\begin{aligned}
&\{ \text{ using Lemma 13 } \} \\
&\mathcal{P}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{P}(\mathcal{S}_b(E)) \\
\equiv\ &\{ \text{ using Lemma 1 } \} \\
&\mathcal{C}(\mathcal{S}_b(E)) \subseteq \mathcal{C}(\mathcal{S}_{\min}(E))
\end{aligned}
$$

Thus $\mathcal{C}(\mathcal{S}_{\min}(E)) = \mathcal{C}(\mathcal{S}_b(E))$. □

Roughly speaking, the aforementioned algorithm computes the union of the sets of edges of each slice. Note that, in general, $F_b(e)[i]$ need not be same as $F_{\min}(e)[i]$. This algorithm can be generalized to conjunction of an arbitrary number of regular predicates.

8.3.2 Composing with respect to Join: $b \equiv b_1 \sqcup b_2 \equiv reg\,(b_1 \vee b_2)$

In this case, the slice $\langle E, \rightarrow\rangle_b$ contains a consistent cut of $\langle E, \rightarrow\rangle$ if the cut satisfies either b_1 or b_2. Given an event e, let $F_{\max}(e)$ denote the vector obtained by taking componentwise maximum of $F_{b_1}(e)$ and $F_{b_2}(e)$. We first prove that no component of $F_b(e)$ is less than (or occurs before) the corresponding component of $F_{\max}(e)$.

Lemma 14. *For each event e and process p_i,*

$$F_{\max}(e)[i] \overset{P}{\underline{\rightarrow}} F_b(e)[i]$$

The proof of Lemma 14 is similar to that of Lemma 13 and therefore has been omitted. We now construct a directed graph $\mathcal{S}_{\max}(E)$ that is similar to $\mathcal{S}_b(E)$ except that we use $F_{\max}$ instead of F_b in its construction. We show that $\mathcal{S}_{\max}(E)$ is in fact cut-equivalent to $\mathcal{S}_b(E)$.

Theorem 17. *$\mathcal{S}_{\max}(E)$ is cut-equivalent to $\mathcal{S}_b(E)$.*

Again, the proof of Theorem 17 is similar to that of Theorem 16 and hence has been omitted. Intuitively, the above-mentioned algorithm computes the intersection of the sets of edges of each slice. In this case, in contrast to the former case, $F_b(e)[i]$ is identical to $F_{\max}(e)[i]$. The reason is as follows. Recall that $F_b(e)[i]$ is the earliest event on p_i that is reachable from e in $\langle E, \rightarrow\rangle_b$. From Theorem 17, at least $F_{\max}(e)[i]$ is reachable from e in $\langle E, \rightarrow\rangle_b$. Thus $F_b(e)[i] \overset{P}{\underline{\rightarrow}} F_{\max}(e)[i]$. Combining it with Lemma 14, we obtain,

Observation 3. *For each event e and process p_i,*

$$F_b(e)[i] = F_{\max}(e)[i]$$

This algorithm can be generalized to disjunction of an arbitrary number of regular predicates.

8.4 Computing the slice for co-regular predicate

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

Consider a computation $\langle E, \rightarrow\rangle$ and a regular predicate b. For convenience, let $\rightarrow_b$ be the edge relation for the slice $\langle E, \rightarrow\rangle_b$. *Without loss of generality, assume that both $\rightarrow$ and $\rightarrow_b$ are transitive relations*. Our objective is to find a property that *distinguishes* the consistent cuts that belong to the slice from the consistent cuts that do not. Consider events e and f such that $e \not\rightarrow f$ but $e \rightarrow_b f$. Then, clearly, a consistent cut that contains f but does not contain e cannot belong to the slice. On the other hand, every consistent cut of the slice that contains f also contains e. This motivates us to define a predicate $prevents(f, e)$ as follows:

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

We now show that the predicate $prevents(f, e)$ is actually a regular predicate. Specifically, we establish that $prevents(f, e)$ is a conjunctive predicate.

Lemma 15. *$prevents(f, e)$ is a conjunctive predicate.*

Proof. Let $proc(e) = p_i$ and $proc(f) = p_j$. We define a local predicate $l_i(e)$ to be true for an event g on process p_i if $g \overset{P}{\rightarrow} e$. Similarly, we define a local predicate $m_j(f)$ to be true for an event h on process p_j if $f \overset{P}{\underline{\rightarrow}} h$. Clearly, $prevents(f, e)$ is equivalent to $l_i(e) \wedge m_j(f)$. □

It turns out that every consistent cut that does not belong to the slice satisfies $prevents(f, e)$ for some pair of events (e, f) such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds. Formally,

Theorem 18. *Let C be a consistent cut of $\langle E, \rightarrow\rangle$. Then,*

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

Proof. We have,

$$
\begin{aligned}
&C \text{ satisfies } \neg b \\
\equiv\ &\{ b \text{ is a regular predicate } \} \\
&\neg\Big(C \in \mathcal{C}(\langle E, \rightarrow\rangle_b)\Big) \\
\equiv\ &\{ \text{ definition of } \mathcal{C}(\langle E, \rightarrow\rangle_b) \} \\
&\neg\langle \forall\, e, f : e \rightarrow_b f : f \in C \Rightarrow e \in C\rangle \\
\equiv\ &\{ \text{ predicate calculus } \}
\end{aligned}
$$

$$
\begin{aligned}
& \langle \exists\, e, f : e \rightarrow_b f : (f \in C) \wedge (e \notin C) \rangle \\
\equiv\ & \{ \text{ definition of } prevents(f,e) \ \} \\
& \langle \exists\, e, f : e \rightarrow_b f : C \ \text{ satisfies } \ prevents(f,e) \rangle \\
\equiv\ & \{ \text{ predicate calculus } \} \\
& \langle \exists\, e, f : (e \rightarrow_b f) \bigwedge \Big((e \rightarrow f) \ \vee \ (e \not\rightarrow f) \Big) : \\
& \qquad\qquad C \text{ satisfies } prevents(f,e) \rangle \\
\equiv\ & \{ \ e \rightarrow f \ \text{ implies } \ e \rightarrow_b f \ \} \\
& \langle \exists\, e, f : (e \rightarrow f) \bigvee \Big((e \rightarrow_b f) \ \wedge \ (e \not\rightarrow f) \Big) : \\
& \qquad\qquad C \text{ satisfies } prevents(f,e) \rangle \\
\equiv\ & \begin{Bmatrix} \text{since } C \text{ is a consistent cut of } \langle E, \rightarrow \rangle, \ C \\ \text{satisfies } prevents(f,e) \text{ implies } e \not\rightarrow f \end{Bmatrix} \\
& \langle \exists\, e, f : (e \rightarrow_b f) \ \wedge \ (e \not\rightarrow f) : \\
& \qquad\qquad C \text{ satisfies } prevents(f,e) \rangle
\end{aligned}
$$

This establishes the theorem. □

Theorem 18 can also be derived using the results in lattice theory [Riv74]. We now give the time-complexity of the algorithm. We start by making the following observation.

Observation 4. *Let e, f and g be events such that $f \rightarrow g$. Then,*

$$prevents(g,e) \ \Rightarrow \ prevents(f,e)$$

Let $K_b(e)$ denote the vector whose i^{th} entry denotes the earliest event f on process p_i, if it exists, such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds. Observation 4 implies that $prevents(K_b(e)[i], e)$, whenever $K_b(e)[i]$ exists, is the weakest predicate among all predicates $prevents(f,e)$, where $proc(f) = p_i$ and $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$. Thus we can ignore all other events on p_i for the purpose of computing the slice for a co-regular predicate. More precisely, Theorem 18 can be restated as:

Theorem 19. *Let C be a consistent cut of $\langle E, \rightarrow \rangle$. Then,*

$$
\begin{aligned}
& C \ \textit{satisfies} \ \neg b \ \equiv \\
& \qquad \langle \exists\, e, p_i : K_b(e)[i] \ \textit{exists} : \\
& \qquad\qquad C \ \textit{satisfies} \ prevents(K_b(e)[i], e) \rangle
\end{aligned}
$$

It turns out that $K_b(e)[i]$ and $F_b(e)[i]$ are closely related.

Observation 5. *$K_b(e)[i]$ exists if and only if $e \not\rightarrow F_b(e)[i]$. Moreover, whenever $K_b(e)[i]$ does exist, it is identical to $F_b(e)[i]$.*

Theorem 19 implies that the number of disjuncts in the predicate equivalent to the negation of a regular predicate is at most $O(n|E|)$. Further, using Observation 5, these disjuncts can be determined in $O(n^2|E|)$ time using the algorithms ComputeJ and ComputeF discussed in Sect. 7.1. The slice with respect to each disjunct can be computed in $O(|E|)$ time using the slicing algorithm for conjunctive predicate. Moreover, for a disjunct $b^{(i)}$, $J_{b^{(i)}}(e)$ for each event e can be computed in $O(n|E|)$ time from its slice $\langle E, \rightarrow \rangle_{b^{(i)}}$ (by topologically sorting the strongly connected components). Using $J_{b^{(i)}}$, it is possible to determine $F_{b^{(i)}}(e)$ for each event e in $O(n|E|)$ time using the algorithm ComputeF. Finally, these slices can be composed together to produce the slice for a co-regular predicate in $O(n|E| \times n|E|) = O(n^2|E|^2)$ time. This is because, given an event e, computing each entry of $F_r(e)$, where $r = reg\,(\neg b)$, using Observation 3 requires $O(n|E|)$ time. Thus the overall time-complexity of the algorithm is $O(n^2|E| + n^2|E|^2) = O(n^2|E|^2)$.

8.5 Computing the slice for k-local predicate for constant k

In case the predicate is regular, we can simply use the algorithm SliceForKLocalRegular to compute the slice in $O(n|E|)$ time. However, if the predicate is not regular, then the slice produced will only be an approximate one. To compute the slice for a k-local predicate, which is not regular, we use the technique developed by Stoller and Schneider [SS95]. For a given computation, their technique can be used to transform a k-local predicate into a predicate in DNF such that each clause has at most k conjuncts. (Such a predicate is referred to as k-DNF predicate.) The resulting predicate has at most m^{k-1} clauses, where m is the maximum number of events on a process. For example, consider the predicate $x_1 \neq x_2$. Let V denote the set of values that x_1 can take in the given computation. Then $x_1 \neq x_2$ can be rewritten as:

$$x_1 \neq x_2 \ \equiv \ \bigvee_{v \in V} \Big((x_1 = v) \wedge (x_2 \neq v) \Big)$$

Note that $|V| \leqslant m$. Thus the resultant predicate, in the above case, consists of at most m clauses where each clause is a conjunctive predicate [Gar02b]). In general, the resultant k-DNF predicate will consist of at most m^{k-1} clauses. To compute the slice for each clause, we use the optimal $O(|E|)$ algorithm given in Sect. 7.3. We then compose these slices together with respect to disjunction to obtain the slice for the given k-local predicate. The overall time-complexity of the algorithm is given by $O(nm^{k-1}|E|)$.

8.6 Computing an approximate slice

Even though it is, in general, NP-hard to compute the slice for an arbitrary predicate, it is still possible to compute an *approximate slice* in many cases. The slice is “approximate” in the sense that it is bigger than the actual slice for the predicate. Nonetheless, it still contains all consistent cuts of the computation that satisfy the predicate. In many cases, the approximate slice that we obtain is much smaller than the computation itself and therefore can be used to prune the search-space for many intractable problems such as monitoring predicates under various modalities.

In particular, using slice composition and slicing algorithms for various classes of predicates, it is possible to compute an approximate slice in an efficient manner for a large class of predicates – namely those derived from other predicates for which it is possible to compute the “exact” slice efficiently using $\wedge$ and $\vee$ operators.

To compute an approximate slice for such a predicate, we first construct the parse tree for the corresponding boolean expression; all predicates occupy leaf nodes whereas all operators occupy non-leaf nodes. We then recursively compute the slice by starting with leaf-nodes and moving up, level by level, until we reach the root. For a leaf node, we use the slicing algorithm appropriate for the predicate contained in the node. For example, if the leaf node contains to a linear predicate, we use the algorithm described in Sect. 8.2. For a non-leaf node, we use the suitable composition algorithm depending on the operator.

Example 8. Suppose we wish to compute an approximate slice for the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$, where each x_i is a regular predicate. First, we compute slices for regular predicates x_1, x_2, x_3 and x_4. Next, we compose the first two and the last two slices together with respect to join to obtain slices for the clauses $x_1 \vee x_2$ and $x_3 \vee x_4$, respectively. Finally, we compose the slices for both clauses together with respect to meet. The slice obtained will contain all consistent cuts that satisfy the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$. □

8.7 Experimental evaluation

In this section, we evaluate the effectiveness of slicing in pruning the search-space for detecting a predicate under *possibly* modality. We compare our approach with that of Stoller, Unnikrishnan and Liu [SUL00], which is based on *partial-order methods* [God96]. Intuitively, when searching the state-space, at each consistent cut, partial-order methods allow only a small subset of enabled transitions to be explored. In particular, we use partial-order methods employing both persistent and sleep sets for comparison. We consider two examples that were also used by Stoller, Unnikrishnan and Liu to evaluate their approach [SUL00].

The first example, called *primary-secondary*, concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The invariant for the algorithm requires that there is a pair of processes p_i and p_j such that (1) p_i is acting as a primary and correctly thinks that p_j is its secondary, and (2) p_j is acting as a secondary and correctly thinks that p_i is its primary. Both the primary and the secondary may choose new processes as their successor at any time; the algorithm must ensure that the invariant is never falsified. A global fault, therefore, corresponds to the complement of the invariant which can be expressed as:

$$\neg I_{ps} = \bigwedge_{i,j \in [1\ldots n],\, i \neq j} \Big(\neg isPrimary_i \vee \neg isSecondary_j \vee (secondary_i \neq p_j) \vee (primary_j \neq p_i) \Big)$$

Note that $\neg I_{ps}$ is a predicate in CNF where each clause is a disjunction of two local predicates. An approximate slice for $\neg I_{ps}$ can be computed in $O(n^3|E|)$ time. In the second example, called *database partitioning*, a database is partitioned among processes p_2 through p_n, while process p_1 assigns tasks to these processes based on the current partition. A process p_i, $i \in [2 \ldots n]$, can suggest a new partition at any time by setting variable $change_i$ to true and broadcasting a message containing the proposed partition. An invariant that should be maintained is: if no process is changing the partition, then all processes agree on the partition. Its complement, corresponding to a global fault, can be expressed as:

$$\neg I_{db} = \neg change_2 \wedge \neg change_3 \wedge \cdots \wedge \neg change_n \wedge \Big(\bigvee_{i,j \in [1\ldots n],\, i \neq j} (partition_i \neq partition_j) \Big) \quad (1)$$

Note that the first $n-1$ clauses of $\neg I_{db}$ are local predicates and the last clause, say LC, is a disjunction of 2-local predicates. Thus, using the technique described in Sect. 8.5, LC can be rewritten as a predicate in DNF with $O(n|E|)$ clauses. To reduce the number of clauses, we proceed as follows. Let V denote the set of values that $partition_1$ assumes in the given computation. Then it can be verified that LC is logically equivalent to:

$$\bigvee_{v \in V} \Big((partition_1 = v) \wedge \Big((partition_2 \neq v) \vee (partition_3 \neq v) \vee \cdots \vee (partition_n \neq v) \Big) \Big) \quad (2)$$

This decreases the number of clauses to $O(n|V|)$. Note that $|V|$ is bounded by the number of events on the first process, and therefore we expect $n|V|$ to be $O(|E|)$. Also, note that rewriting LC does not have any impact on the performance of the partial order methods approach because the set of processes on whose variables the clause LC depends stay the same in both cases.

We use the simulator implemented in Java by Stoller, Unnikrishnan and Liu to generate computations of these protocols. Message latencies and other delays (e.g., how long to wait before looking for a new successor) are selected randomly using the distribution $1 + \exp(x)$, where $\exp(x)$ is the exponential distribution with mean x. Further details of the two protocols and the simulator can be found elsewhere [SUL00]. We consider two different scenarios: *fault-free* and *faulty*. The simulator always produces fault-free computations. A faulty computation is generated by randomly injecting faults into a fault-free computation. Note that in the first (fault-free) scenario, we know *a priori* that the computation does not contain a faulty consistent cut. We cannot, however, assume the availability of such knowledge in general. Thus it is important to study the behavior of the two predicate detection techniques in the fault-free scenario as well. We implement the algorithm for slicing a computation in Java. We compare the two predicate detection techniques with respect to two metrics: amount of time spent and amount of memory used. In the case of the former technique, both metrics also include the overhead of computing the slice. We run our experiments on a machine with Pentium 4 processor operating at 1.8 GHz clock frequency and 512 MB of physical memory.

For primary-secondary example, the simulator is run until the number of events on some process reaches 90. The measurements averaged over 300 computations are displayed in Table 1. With computation slicing, for fault-free computations, the slice is always empty. As the number of processes is increased from 6 to 12, the amount of time spent increases from 0.36s to 2.85s, whereas the amount of memory used increases from 1.21 M to 2.16 M. On the other hand, with partial-order methods, they increase, almost exponentially, from 0.07s to

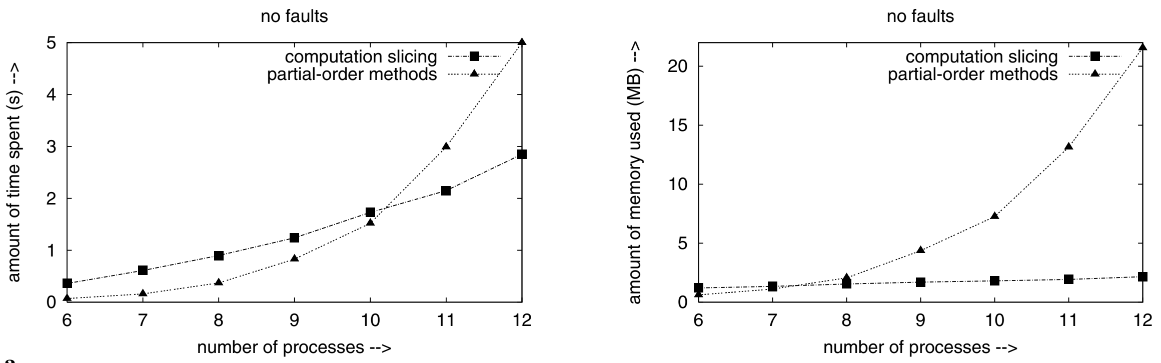
Table 1. Primary-Secondary example with the number of events on a process bounded by 90

	No faults				One injected fault			
Number of processes	Partial-order methods		Computation slicing		Partial-order methods		Computation slicing	
n	T	M	T	M	T	M	T	M
6	0.07	0.62	0.36	1.21	0.05	0.41	0.37	1.38
7	0.16	1.11	0.61	1.34	0.11	0.81	0.59	1.41
8	0.37	2.06	0.90	1.54	0.31	1.79	0.91	1.61
9	0.83	4.37	1.24	1.70	0.59	3.05	1.21	1.77
10	1.52	7.26	1.73	1.81	1.12	5.54	1.70	2.00
11	2.99*	13.14*	2.15	1.93	2.09*	9.50*	2.13	2.27
12	5.0*	21.56*	2.85	2.16	3.51*	14.13*	2.77	2.43

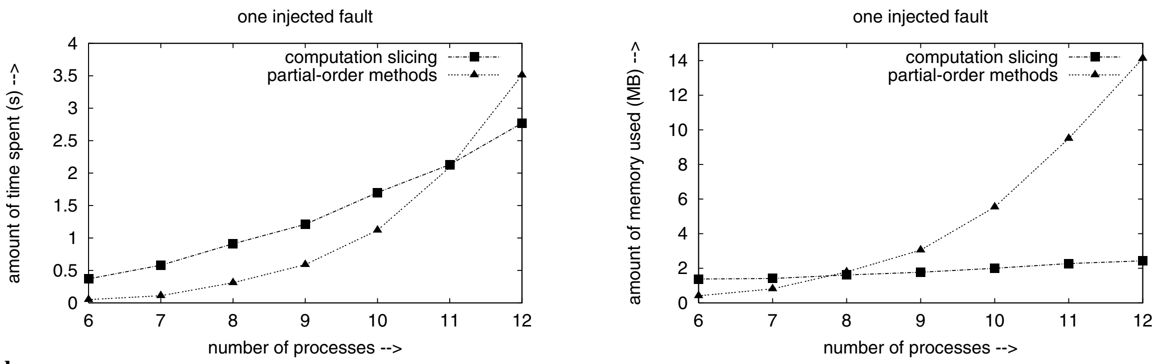
n: number of processes T: amount of time spent (in s)

M: amount of memory used (in MB)

*: does not include the cases in which the technique runs out of memory



a



b

Fig. 11. Primary-Secondary example with the number of events on a process bounded by 90 for **a** no faults and **b** one injected fault

5.0s and 0.62 M to 21.56 M, respectively. Even on injecting a fault, the slice stays quite small. After computing the slice, in our experiments, we only need to examine at the most 13 consistent cuts to locate a faulty consistent cut, if any. The amount of time spent and the amount of memory used, with computation slicing, increase from 0.37s to 2.77s and 1.38 M to 2.43 M, respectively, as the number of processes is increased from 6 to 12. However, with partial-order methods, they again increase almost exponentially from 0.05s to 3.51s and 0.41 M to 14.13 M, respectively. Clearly, with slicing, both time and space complexities for detecting a global fault, if it exists, in primary-secondary example are polynomial in input size for the specified range of parameters. In contrast, with partial-order methods, they are exponential in input size. Figure 11a and Fig. 11b plot the variation in the two metrics with the number of processes for the two approaches.

The worst-case performance of the partial-order methods approach is quite bad. With 12 processes in the system and the limit on the memory set to 100 MB, the approach runs out of memory in approximately 6% of the cases. In around two-thirds of such cases, the computation actually contains a consistent cut that does not satisfy the invariant. It may be noted that we do not include the above-mentioned cases in computing the average amount of time spent and memory

Table 2. Database partitioning example with the number of events on a process bounded by 80

	No faults				One injected fault			
Number of Processes	Partial-order methods		Computation slicing		Partial-order methods		Computation slicing	
n	T	M	T	M	T	M	T	M
4	0.05	0.07	0.24	1.06	0.03	0.05	0.24	0.95
5	0.05	0.09	0.34	1.13	0.03	0.08	0.36	0.99
6	0.05	0.13	0.50	1.22	0.03	0.10	0.48	1.13
7	0.05	0.22	0.59	1.33	0.04	0.16	0.62	1.25
8	0.07	0.31	0.76	1.41	0.04	0.23	0.73	1.57
9	0.07*	0.36*	0.89	1.56	0.05	0.31	0.92	1.69
10	0.08*	0.40*	1.09	1.80	0.05*	0.42*	1.07	1.80

n: number of processes T: amount of time spent (in s)

M: amount of memory used (in MB)

*: does not include the cases in which the technique runs out of memory

used. Including them will only make the average performance of the partial-order methods approach worse. Further, the performance of the partial-order methods approach appears to be very sensitive to the location of the fault, in particular, whether it occurs earlier during the search or much later or perhaps does not occur at all. Consequently, the variation or standard deviation in the two metrics is very large. This has implications when predicate detection is employed for achieving software fault tolerance. Specifically, it becomes hard to provision resources (in our case, memory) when using partial-order methods approach. If too little memory is reserved, then, in many cases, the predicate detection algorithm will not be able to run successfully to completion. On the other hand, if too much memory is reserved, the memory utilization will be sub-optimal.

For database partitioning example, the simulator is run until the number of events on some process reaches 80. (This also implies that V in the alternative formulation of LC has at most 80 values.) The measurements averaged over 300 computations are shown in Table 2. FiguresFigure 12c and d plot the variation in the two metrics with the number of processes for the two approaches. As it can be seen, the average performance of partial-order methods is much better than computation slicing. This is because substantial overhead is incurred in computing the slice even after changing the formulation of LC from (1) to (2). (The alternative formulation of LC decreases the time-complexity of computing the slice by a factor of four.) The slice itself is quite small. Specifically, for the fault-free scenario, the slice is always empty. On the other hand, for the faulty scenario, only at most 4 transitions need to be explored after computing the slice to locate a faulty consistent cut, if any.

Even for database partitioning example, for 10 processes, the partial-order methods approach runs out of memory in a small fraction – approximately 1% – of the cases. Therefore the worst-case performance of computation slicing is better than partial-order methods. To get the best of both worlds, predicate detection can be first done using the partial-order methods approach. In case it turns out that the approach is using too much memory, say more than $cn|E|$ for some small constant c, and still has not terminated, it can be aborted and the computation slicing approach can then be used for predicate detection.

9 Discussion

In this paper, we introduce the notion of computation slice and prove its usefulness in evaluating global properties in distributed computations. We provide efficient polynomial-time algorithms for computing the slice for several useful classes of predicates. For many other classes predicates for which it is otherwise provably NP-complete to compute the slice, we present efficient heuristic algorithms for computing an approximate slice. Our experimental results demonstrate that slicing can lead to an exponential improvement over existing techniques in terms of time and space for intractable problems such as predicate detection.

We prove elsewhere [MSGA04] 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. At first glance it may seem that we are not any better off than we were before. After all, predicate detection is "equivalent" to computation slicing. Then, how can slicing be used to improve the complexity of predicate detection? In fact, slicing can indeed be used to facilitate predicate detection as illustrated by the following example. Consider a predicate b that is a conjunction of two clauses b_1 and b_2. Now, assume that b_1 can be detected efficiently but b_2 has no structural property that can be exploited for efficient detection. To detect b, without computation slicing, we are forced to use techniques [CM91,AV01,SUL00] which do not take advantage of the fact that b_1 can be detected efficiently. With computation slicing, however, we can first compute the slice for b_1. If only a small fraction of consistent cuts satisfy b_1, then, instead of detecting b in the computation, it is much more efficient detect b in the slice. Therefore by spending only polynomial amount of time in computing the slice we can throw away exponential number of consistent cuts, thereby obtaining an exponential speedup overall. Consequently, by virtue of the equivalence result, it is possible to compute the slice efficiently for many more classes

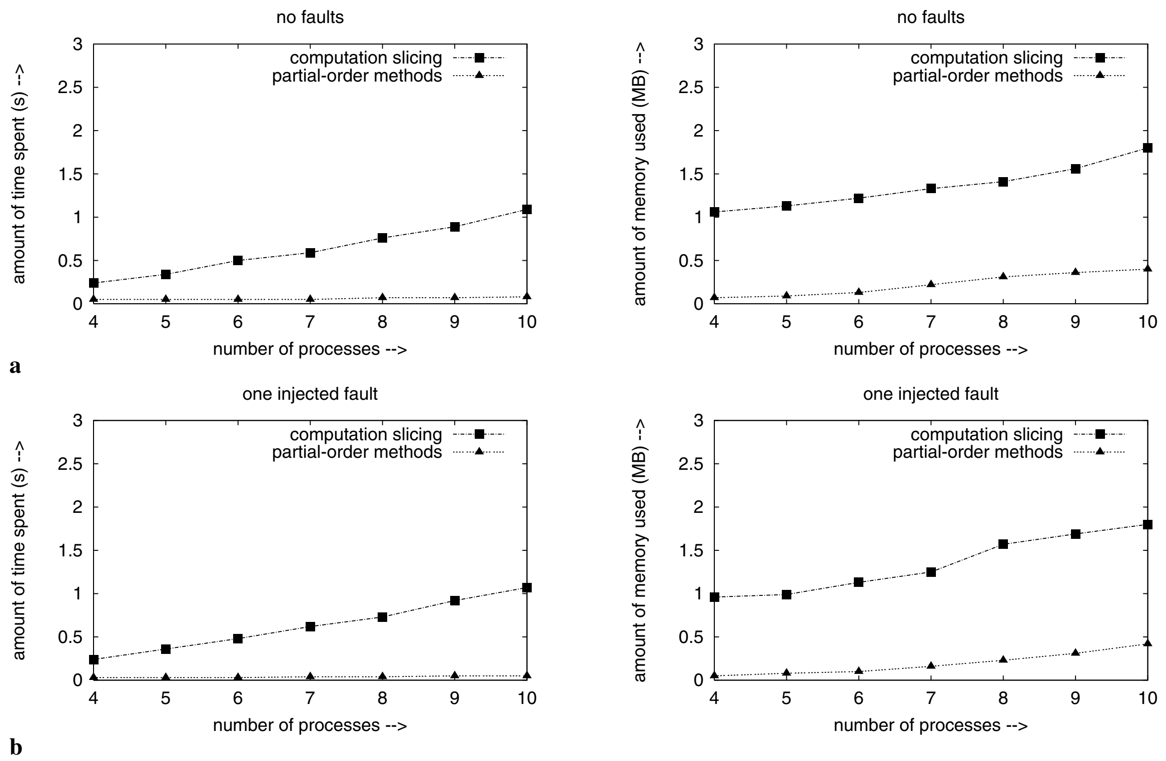


Fig. 12. Database partitioning example with the number of events on a process bounded by 80 for **c**) no faults and **d** one injected fault

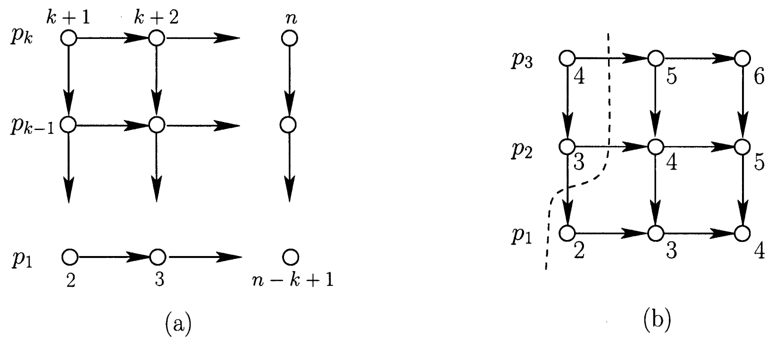


Fig. 13. a Computation for subsets of $[n]$ of size k, and **b** example when $n = 6$ and $k = 3$

predicates including relational predicates [CG95], stable and co-stable predicates, and co-linear predicates [CG98].

Although in this paper our focus is on distributed systems, slicing has applications in other areas as well, such as combinatorics [Gar02a]. A combinatorial problem usually requires counting, enumerating or ascertaining the existence of structures that satisfy a given property. 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 global states (or 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 global states that satisfy b. This gives us an efficient algorithm to count, enumerate or detect structures that satisfy b when the total set of structures is large but the set of structures satisfying b is small.

For example, consider the following problem in combinatorics: *Count the number of subsets of size k of the set $\{1, 2, \ldots, n\}$ (hereafter denoted by $[n]$) which do not contain any consecutive numbers (for given values of n and k).* To solve this problem, we first come up with a distributed computation such that there is a one-to-one correspondence between global states and subsets of size k. Figure 13a depicts a computation such that all subsets of $[n]$ of size k are its global states. There are k processes in this computation and each process executes exactly $n - k$ events. By the structure of the computation, if, in a global state, process p_i has executed j events, then process p_{i+1} must have also executed at least j events. The correspondence between subsets of $[n]$ and global states can be understood as follows. If a process p_i has executed m events in a global state, then the element $m + i$ belongs to the corresponding subset. Thus process p_1 chooses a number from $1 \ldots (n - k + 1)$ (because there are $n - k$ events); process p_2 chooses the next larger number and so on. Figure 13b gives an example of the computation for subsets of size 3 of the set $[6]$. The global state shown corresponds to the subset $\{1, 3, 4\}$.

Now we define predicate b to be "the global state does not contain any consecutive numbers". For the computation we have constructed, it can be easily verified that the predicate

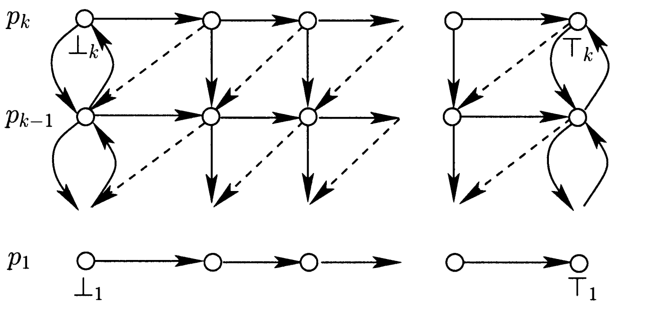


Fig. 14. Slice with respect to the predicate "the global state does not contain any consecutive numbers"

b is regular. Therefore one can mechanically and efficiently compute the slice of the computation with respect to b. Figure 14 shows the slice which includes precisely such subsets. Clearly, if the event labeled m on process p_i has been executed, then the event labeled $m+2$ on process p_{i+1} should also have been executed. This can be accomplished by adding the dotted arrows to the computation as depicted in the figure. By collapsing all strongly connected components and by removing the transitively implied edges, we obtain a graph that is isomorphic to the graph shown in Fig. 13a, with k processes and in which each process executes $n-k-(k-1)$ events. Therefore the total number of such sets is $\binom{n-k+1}{k}$. [Gar02a] gives several other applications of slicing for analyzing problems in integer partitions, set families, and set of permutations.

At present, all our algorithms for computing a slice are centralized in nature. They assume that there is a designated process that is responsible for collecting all the events that have been generated and constructing a trace using them. Slicing algorithms use this trace to compute the slice. While the centralized approach is quite adequate for applications such as testing and debugging, for other applications including software fault tolerance, a more distributed approach is desirable. Moreover, all slicing algorithms presented in this paper are off-line in nature. To detect a software fault in a more timely manner, however, it is desirable and sometimes essential that its slice be computed and analyzed for any possible fault in an incremental fashion. As the execution of the system progresses and more and more events in the trace become available, the current slice is updated to reflect the newly generated events. In the future, we plan to develop slicing algorithms that are incremental and more distributed in nature.

References

[AV01] Alagar S, Venkatesan S: Techniques to Tackle State Explosion in Global Predicate Detection. IEEE Trans Software Eng 27(8):704–714 (2001)

[BFR96] Babaoğlu Ö, Fromentin E, Raynal M: A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations. J Syst Software 33(3):287–298 (1996)

[Bou87] Bouge L: Repeated Snapshots in Distributed Systems with Synchronous Communication and their Implementation in CSP. Theor Comput Sci 49:145–169 (1987)

[CG95] Chase C, Garg VK: On Techniques and their Limitations for the Global Predicate Detection Problem. In: Proceedings of the Workshop on Distributed Algorithms (WDAG), pp 303–317, France, 1995

[CG98] Chase C, Garg VK: Detection of Global Predicates: Techniques and their Limitations. Distrib Comput (DC) 11(4):191–201 (1998)

[Che93] Cheng J: Slicing Concurrent Programs – A Graph-Theoretical Approach. In: Proceedings of the International Workshop on Automated Debugging (AADEBUG), pp 223–240, 1993

[CL85] Chandy KM, Lamport L: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Trans Comput Syst 3(1):63–75 (1985)

[CM91] Cooper R, Marzullo K: Consistent Detection of Global Predicates. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp 163–173, Santa Cruz, California, 1991

[DP90] Davey BA, Priestley HA: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK, 1990

[Fid91] Fidge C: Logical Time in Distributed Computing Systems. IEEE Comput 24(8):28–33 (1991)

[Gar02a] Garg VK: 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

[Gar02b] Garg VK: Elements of Distributed Computing. Wiley, New York, NY, 2002

[GM01] Garg VK, Mittal N: On Slicing a Distributed Computation. In: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), pp 322–329, Phoenix, Arizona, April 2001

[God96] Godefroid P: Partial-Order Methods for the Verification of Concurrent Systems, Lect Notes Comput Sci, vol 1032, Springer, 1996

[HMSR98] Hurfin M, Mizuno M, Singhal M, Raynal M: Efficient distributed detection of conjunctions of local predicates. IEEE Trans Software Eng 24(8):664–677 (1998)

[JMN95] Jegou R, Medina R, Nourine L: Linear Space Algorithm for On-line Detection of Global Predicates. In: Desel J (ed) Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), pp 175–189. Springer, 1995

[JZ88] Johnson DB, Zwaenepoel W: Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In: Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC), pp 171–181, August 1988

[KR97] Korel B, Rilling J: Application of Dynamic Slicing in Program Debugging. In: Kamkar M (ed) Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG), pp 43–57, Linköping, Sweden, May 1997

[Ksh98] Kshemkalyani AD: A Framework for Viewing Atomic Events in Distributed Computations. Theor Comput Sci 196(1–2):45–70 (1998)

[Lam78] Lamport L: Time, Clocks, and the Ordering of Events in a Distributed System. Commun ACM (CACM) 21(7):558–565 (1978)

[Mat89] Mattern F: Virtual Time and Global States of Distributed Systems. In: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG), pp 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989

[MG01a] Mittal N, Garg VK: Computation Slicing: Techniques and Theory. In: Proceedings of the Symposium on Distributed Computing (DISC), pp 78–92, Lisbon, Portugal, October 2001

[MG01b] Mittal N, Garg VK: On Detecting Global Predicates in Distributed Computations. In: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), pp 3–10, Phoenix, Arizona, April 2001

[MG03a] Mittal N, Garg VK: Software Fault Tolerance of Distributed Programs using Computation Slicing. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS), pp 105–113, Providence, Rhode Island, May 2003

[MG03b] Mittal N, Garg VK: Techniques and Applications of Computation Slicing. Technical Report UTDCS-15-03, Department of Computer Science, The University of Texas at Dallas, April 2003

[MG04] Mittal N, Garg VK: Finding Missing Synchronization in a Distributed Computation using Controlled Re-execution. Distrib Comput (DC), March 2004. Online First

[MSGA04] Mittal N, Sen A, Garg VK, Atreya R: Finding Satisfying Global States: All for One and One for All. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), April 2004

[NX95] Netzer RHB, Xu J: Necessary and Sufficient Conditions for Consistent Global Snapshots. IEEE Trans Parallel Distrib Syst (TPDS) 6(2):165–169 (1995)

[Riv74] Rival I: Maximal Sublattices of Finite Distributive Lattices II. Proc Amer Math Soci 44(2):263–268 (1974)

[SK86] Spezialetti M, Kearns P: Efficient Distributed Snapshots. In: Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS), pp 382–388, 1986

[SS95] Stoller SD, Schneider F: Faster Possibility Detection by Combining Two Approaches. In: Proceedings of the Workshop on Distributed Algorithms (WDAG), Lect Notes Comput Sci (LNCS), vol 972 pp 318–332, France, September 1995

[SUL00] Stoller SD, Unnikrishnan L, Liu YA: Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In: Proceedings of the 12th International Conference on Computer-Aided Verification (CAV), Lect Notes Comput Sci (LNCS), vol 1855 pp 264–279. Springer, July 2000

[TG98] Tarafdar A, Garg V:. Predicate Control for Active Debugging of Distributed Programs. In: Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP), pp 763–769, Orlando, 1998

[Tip95] Tip F: A Survey of Program Slicing Techniques. J Programm Lang 3(3):121–189 (1995)

[Ven95] Venkatesh G: Experimental Results from Dynamic Slicing of C Programs. ACM Trans Programm Lang Syst (TOPLAS) 17(2):197–216 (1995)

[Wan97] Wang Y-M: Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. IEEE Trans Comput 46(4):456–468 (1997)

[Wei82] Weiser M: Programmers Use Slices when Debugging. Commun ACM (CACM) 25(7):446–452 (1982)

Appendix A: Omitted proofs

Proof (of Theorem 2). We have to prove that if b_1 and b_2 are regular predicates then so is $b_1 \wedge b_2$. Consider consistent cuts C_1 and C_2 that satisfy $b_1 \wedge b_2$. By semantics of conjunction, both C_1 and C_2 satisfy b_1 as well as b_2. Since b_1 and b_2 are regular predicates, $C_1 \cap C_2$ satisfies b_1 and b_2. Again, by semantics of conjunction, $C_1 \cap C_2$ satisfies $b_1 \wedge b_2$. Likewise, $C_1 \cup C_2$ satisfies $b_1 \wedge b_2$. Thus $b_1 \wedge b_2$ is a regular predicate. □

Proof (of Theorem 9). ($reg\,(b)$ *is weaker than* b) Follows from the definition.

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

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

Proof (of Lemma 10). It suffices to prove that if C is a consistent cut of $\mathcal{H}_b(E)$, then $C(Q)$ is a consistent cut of $\langle E(Q), \rightarrow(Q)\rangle_b$. We prove the contrapositive. We have,

$$
\begin{aligned}
& C(Q) \text{ is not a consistent cut of } \langle E(Q), \rightarrow(Q)\rangle_b \\
\Rightarrow\ & \{ \text{ definition of consistent cut } \} \\
& \langle \exists\, e, f \in E(Q) : \\
& \text{there is a path from } e \text{ to } f \text{ in } \langle E(Q), \rightarrow(Q)\rangle_b : \\
& (f \in C(Q)) \wedge (e \notin C(Q) \\
\Rightarrow\ & \begin{Bmatrix} \text{using definition of } F_b(e,Q)[i] \text{ where} \\ p_i = proc(f)\rangle \end{Bmatrix} \\
& \langle \exists\, e, f \in E(Q) : F_b(e,Q)[i] \xrightarrow{P} f : \\
& \qquad\qquad (f \in C(Q)) \wedge (e \notin C(Q))\rangle \\
\Rightarrow\ & \{ \text{ using definition of } K_b(e)[i] \} \\
& \langle \exists\, e, f \in E(Q) : K_b(e)[i] \xrightarrow{P} f : \\
& \qquad\qquad (f \in C(Q)) \wedge (e \notin C(Q))\rangle \\
\Rightarrow\ & \{ \text{ using definition of } \mathcal{H}_b(E) \} \\
& \langle \exists\, e, f \in E(Q) : \\
& \text{there is a path from } e \text{ to } f \text{ in } \mathcal{H}_b(E) : \\
& (f \in C(Q)) \wedge (e \notin C(Q))\rangle \\
\Rightarrow\ & \begin{Bmatrix} f \in C(Q) \Rightarrow f \in C \text{ and} \\ (e \notin C(Q)) \wedge (e \in E(Q)) \Rightarrow e \notin C \end{Bmatrix} \\
& \langle \exists\, e, f \in E : \\
& \text{there is a path from } e \text{ to } f \text{ in } \mathcal{H}_b(E) : \\
& (f \in C) \wedge (e \notin C)\rangle \\
\Rightarrow\ & \{ \text{ definition of consistent cut } \} \\
& C \text{ is not a consistent cut of } \mathcal{H}_b(E)
\end{aligned}
$$

This establishes the lemma. □

Proof (of Theorem 14). The first two propositions are easy to verify. We only prove the last proposition. As for the last proposition, it can be verified that a regular predicate is controllable in a computation if and only if there exists a path from the initial to the final consistent cut in the lattice (of consistent cuts) such that every consistent cut along the path satisfies the predicate [TG98]. Note that the path from the initial to the final consistent cut actually corresponds to a longest chain in the lattice of consistent cuts. For a lattice L, let $\mathsf{height}(L)$ denote the length of a longest chain in L. Therefore if b is controllable in $\langle E, \rightarrow \rangle$, then a longest chain in $\mathcal{C}(E)$ is contained in $\mathcal{C}_b(E)$ as well and vice versa. This implies that $\mathsf{height}(\mathcal{C}(E)) \leqslant \mathsf{height}(\mathcal{C}_b(E))$. However, $\mathcal{C}_b(E) \subseteq \mathcal{C}(E)$ implying that $\mathsf{height}(\mathcal{C}_b(E)) \leqslant \mathsf{height}(\mathcal{C}(E))$. Therefore we have:

$$controllable : b \ \equiv\ \mathsf{height}(\mathcal{C}(E)) = \mathsf{height}(\mathcal{C}_b(E))$$

For a finite distributive lattice L, the length of its longest chain is equal to the number of its join-irreducible elements [DP90]. In other words, $\mathsf{height}(L) = \mathcal{JI}(L)$. Also, as observed before in Sect. 6, for a directed graph, the number of join-irreducible elements of the lattice generated by its set of consistent cuts – including trivial consistent cuts – is same as the number of its strongly connected components. As a result,

$$\mathsf{height}(\mathcal{C}(E)) = \mathcal{JI}(\mathcal{C}(E)) = \mathsf{scc}(\langle E, \rightarrow \rangle), \text{ and}$$
$$\mathsf{height}(\mathcal{C}_b(E)) = \mathcal{JI}(\mathcal{C}_b(E)) = \mathsf{scc}(\langle E, \rightarrow \rangle_b)$$

This establishes the theorem. □

Neeraj Mittal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. 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 co-director 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.

Vijay K. Garg received his B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur in 1984 and M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at 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, 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* (Wiley & Sons, 2002), *Principles of Distributed Systems* (Kluwer, 1996) and a co-author of the book *Modeling and Control of Logical Discrete Event Systems* (Kluwer, 1995).