Modeling, Analyzing and Slicing Periodic Distributed Computations

Vijay K. Garg [*], Anurag Agarwal[**], and Vinit Ogale[* * *]

The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Abstract. The earlier work on predicate detection has assumed that the given computation is finite. Detecting violation of a liveness predicate requires that the predicate be evaluated on an infinite computation. In this work, we develop the theory and associated algorithms for predicate detection in infinite runs. In practice, an infinite run can be determined in finite time if it consists of a recurrent behavior with some finite prefix. Therefore, our study is restricted to such runs. We introduce the concept of *d-diagram*, which is a finite representation of infinite directed graphs. Given a d-diagram that represents an infinite distributed computation, we solve the problem of determining if a global predicate ever became true in the computation. The crucial aspect of this problem is the stopping rule that tells us when to conclude that the predicate can never become true in future. We also provide an algorithm to provide vector timestamps to events in the computation for determining the dependency relationship between any two events in the infinite run. Finally, we give an algorithm to compute a slice of a d-diagram which concisely captures all the consistent global states of the computation satisfying the given predicate.

1 Introduction

Correctness properties of distributed programs can be classified either as safety properties or liveness properties. Informally, a safety property states that the program never enters a bad (or an unsafe) state, and a liveness property states that the program eventually enters into a good state. For example, in the classical dining philosopher problem a safety property is that "two neighboring philosophers never eat concurrently" and a liveness property is that "every hungry philosopher eventually eats." Assume that a programmer is interested in monitoring for violation of a correctness property in her distributed program. It is clear how a runtime monitoring system would check for violation of a safety property. If it detects that there exists a consistent global state [1] in which two neighboring philosophers are eating then the safety property is violated. The

[*] part of the work was performed at the University of Texas at Austin supported in part by the NSF Grants CNS-0509024, CNS-0718990, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship.

[**] currently at Google

[* * *] currently at Microsoft

literature in the area of global predicate detection deals with the complexity and algorithms for such tasks [2, 3]. However, the problem of detecting violation of the liveness property is harder. At first it appears that detecting violation of a liveness property may even be impossible. After all, a liveness property requires something to be true eventually and therefore no finite observation can detect the violation. We show in this paper a technique that can be used to infer violation of a liveness property in spite of finite observations. Such a technique would be a basic requirement for detecting a temporal logic formula [4] on a computation for runtime verification.

There are three important components in our technique. First, we use the notion of a *recurrent* global state. Informally, a global state is recurrent in a computation γ if it occurs more than once in it. Existence of a recurrent global state implies that there exists an infinite computation δ in which the set of events between two occurrences of can be repeated ad infinitum. Note that γ may not even be a prefix of δ. The actual behavior of the program may not follow the execution of δ due to nondeterminism. However, we know that δ is a legal behavior of the program and therefore violation of the liveness property in δ shows a bug in the program. We note here that in this paper we are only interested in algorithms that do not use randomization to ensure liveness properties. For a randomized algorithm, existence of δ may be okay because the algorithm may guarantee liveness only with probability one.

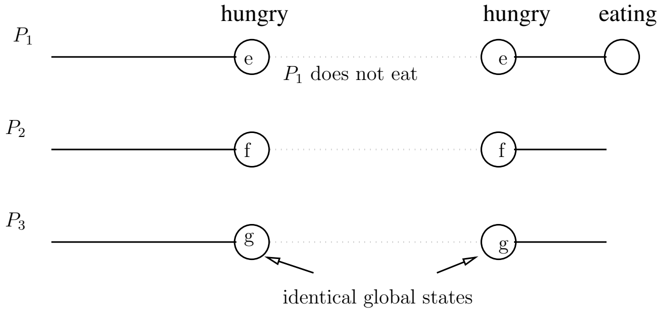

Fig. 1: A finite distributed computation C of dining philosophers

For example, in figure 1, a global state repeats where the same philosopher P_1 is hungry and has not eaten in between these occurrences. P_1 does get to eat after the second occurrence of the *recurrent* global state; and, therefore a check that "every hungry philosopher gets to eat" does not reveal the bug. It is simple to construct an infinite computation δ from the observed finite computation γ in which P_1 never eats. We simply repeat the execution between the first and the second instance of the recurrent global state. This example shows that the approach of capturing a periodic part of a computation can result in detection of bugs that may have gone undetected if the periodic behavior is not considered.

The second component of our technique is to develop a finite representation of the infinite behavior γ. Mathematically, we need a finite representation of the infinite but

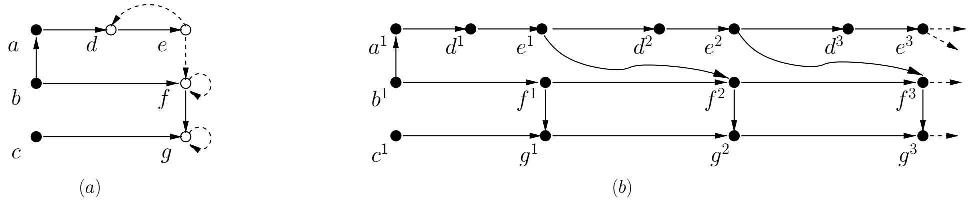

Fig. 2: (a) A d-diagram and (b) its corresponding infinite poset

periodic poset γ. In this paper, we propose the notion of d-diagram to capture infinite periodic posets. Just as finite directed acyclic graphs (dag's) have been used to represent and analyze finite computations, d-diagrams may be used for representing periodic infinite distributed computations for monitoring or logging purposes. The logging may be useful for replay or offline analysis of the computation. Figure 2 shows a d-diagram and the corresponding infinite computation. The formal semantics of a d-diagram is given in Section 4. Intuitively, the infinite poset corresponds to the infinite unrolling of the recurrent part of the d-diagram.

The third component of our technique is to develop efficient algorithms for analyzing the infinite poset given as a d-diagram. Three kinds of computation analysis have been used in past for finite computations. The first analysis is based on vector clocks which allows one to answer if two events are dependent or concurrent, for example, works by Fidge [5] and Mattern [6]. We extend the algorithm for timestamping events of a finite poset to that for the infinite poset. Of course, since the set of events is infinite, we do not give explicit timestamp for all events, but only an implicit method that allows efficient calculation of dependency information between any two events when desired. The second analysis we use is to detect a global predicate B on the infinite poset given as a d-diagram. In other words, we are interested in determining if there exists a consistent global state which satisfies B. Since the computation is infinite, we cannot employ the traditional algorithms [2, 3] for predicate detection. Because the behavior is periodic it is natural that a finite prefix of the infinite poset may be sufficient to analyze. The crucial problem is to determine the number of times the recurrent part of the d-diagram must be unrolled so that we can guarantee that B is true in the finite prefix *iff* it is true in the infinite computation. We show in this paper that it is sufficient to unroll the d-diagram N times where N is the number of processes in the system. The third analysis is based on the notion of slicing a computation introduced in [7, 8]. Informally, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. Slicing is crucial in detecting nested temporal logic predicates of a distributed computation in the partial order model. Sen and Garg use slicing in [**?**] to detect a subset of CTL called Regular CTL. Ogale and Garg use it to detect another temporal logic called BTL. All the earlier work on computing slices was based on finite computation and we extend that work for infinite computations in this paper. For example, the interpretation of "a hungry philosopher never gets to eat" was modified by earlier work to "a hungry philosopher does not eat by the end of the computation." This interpretation, although useful in some cases, is not accurate and may give false positives when employed by the programmer to detect bugs. This paper is the first one to explicitly analyze the periodic behavior to ensure that the interpretation of formulas is on the infinite computation.

In summary, this paper makes the following contributions:

- We introduce the notion of recurrent global states in a distributed computation and propose a method to detect them.
- We introduce and study a finite representation of infinite directed computations called d-diagrams.
- We provide a method of timestamping nodes of a d-diagram so that the happened-before relation can be efficiently determined between any two events in the given infinite computation.
- We define the notion of core of a d-diagram that allows us to use any predicate detection algorithm for finite computations on infinite computations as well.
- We present an algorithm to compute slice of the periodic computation represented using a d-diagram.

2 A Motivating Example: A Mutual Exclusion Algorithm

In this section we give a motivating example for the theory proposed in the paper. Consider the distributed algorithm (shown in Fig. 3) proposed by a programmer to coordinate access to the critical section by processes $P_1 ... P_N$ in a distributed system. In this algorithm, a process can enter the critical section only if it has the token. If a process is hungry and it does not have the token, it sends a request to all processes. Every process P_k maintains an array $reqFlag$ which keeps track of all processes that have outstanding requests for the token. When a process with the token is done with its critical section, it sends the token to any process that has an outstanding request. It also sends a *release* message to all other processes so that they can reset the corresponding $reqFlag$.

Now suppose that the programmer is interested in testing this algorithm. When she runs the algorithm, suppose that the following execution α takes place. All three philosophers get hungry. P_1 eats first and then sends token to P_2 who eats next. P_1 gets hungry again and gets the token from P_2. Once P_1 has finished eating, it sends the token to P_3 who eats last.

The distributed computation, α can be described in more detail by the following sequences of events at each process.

P_1: reqCS, recvReq from P_3, eats, recvReq from P_2, relCS sending token to P_2, reqCS, recvToken from P_2, eats, relCS sending token to P_3.

P_2: recvReq from P_3, reqCS, recvToken from P_1, recvReq from P_1, relCS sending token to P_1, recvRel from P_1

P_3: reqCS, recvReq from P_2, recvRel from P_1, recvReq from P_1, recvRel from P_2, recvToken from P_1, eat

```
$P_k$::
 var
     $state$: {thinking, hungry, eating} initially $thinking$;
     $haveToken$: boolean initially false except for $P_1$;
     $reqFlag$: array[$1..N$] of boolean initially $\forall j : reqFlag[j] = false$;

 $reqCS$: On becoming hungry (enabled if $(state = thinking)$
     $state := hungry$;
     if $(!haveToken)$ ;
         $reqFlag[k] := true$;
         send $(request)$ to others;

 $recvToken$: Upon $receive(token)$ from $P_j$:
     $haveToken := true$;
     $reqFlag[j] := false$;

 $eat$: (enabled if $haveToken \wedge (state = hungry)$ )
     $state := eating$;
     $eat$;
     $state := thinking$;
     $reqFlag[k] := false$;

 $relCS$: To release critical section: (enabled if $(state = thinking)$ and $\exists i : reqFlag[i]$)
     send token to $P_i$;
     send release to all others except $P_i$;
     $haveToken := false$;

 $recvReq$: Upon $receive(request)$ from $P_j$:
         $reqFlag[j] := true$;

 $recvRel$: Upon $receive(release)$ from $P_j$:
         $reqFlag[j] := false$;
```

Fig. 3: A (faulty) distributed mutual exclusion algorithm with FIFO channels

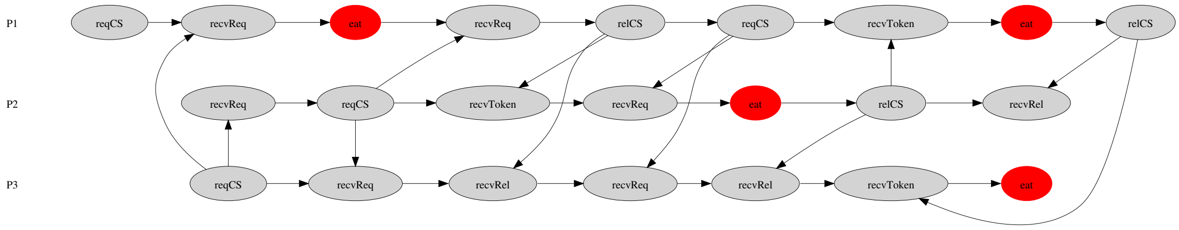

Fig. 4: A finite distributed computation α for the mutex algorithm

The distributed computation α shown in Fig. 4 satisfies the standard safety and liveness properties: two processes do not eat at the same time and every hungry process eventually eats. However, with the techniques proposed in the paper we would be able to construct an infinite computation α' that is a valid execution and violates the liveness property.

Consider the following prefix β of computation α:

P_1: reqCS, recvReq from P_3, eats
P_2: recvReq from P_3.
P_3: reqCS.

The global state after β is:
State of P_1: haveToken = true; reqFlag = (false, false, true); state=thinking
State of P_2: haveToken = false; reqFlag = (false, false, true); state=thinking
State of P_3: haveToken = false; reqFlag = (false, false, true); state=thinking

Note that the same global state occurs in α after P_1 has eaten twice in the the following prefix: (β followed by γ)

P_1: reqCS, recvReq from P_3, eats, recvReq from P_2, relCS sending token to P_2, reqCS, recvToken from P_2, eats

P_2: recvReq from P_3, reqCS, recvToken from P_1, recvReq from P_1, relCS sending token to P_1

P_3: reqCS, recvReq from P_2, recvRel from P_1, recvReq from P_1, recvRel from P_2

Since the global state is identical after executing β and β followed by γ, there exists a valid execution α' in which β is followed by γ an infinite number of times. The execution α' violates the liveness property because P_3 stays hungry forever. The infinite execution derived from α is shown as a p-diagram in Fig. 5.

Remark: The algorithm can be fixed by making two changes. First, we should maintain a *queue* of requests at all processes rather than a *set* of requests as implemented in Fig. 3 by the boolean array $reqFlag$. Second, a process with the token that has eaten last must release the token if the queue of requesting processes is nonempty.

3 Model of a Distributed Computation

We first describe our model of a distributed computation. We assume a message passing asynchronous system without any shared memory or a global clock. A distributed program consists of N sequential processes denoted by $P = \{P_1, P_2, \ldots, P_N\}$ communicating via asynchronous messages. A *local computation* of a process is a sequence of events. An event is either an internal event, a send event or a receive event. When an event is executed, it changes the state of the process (and possibly the state of incoming

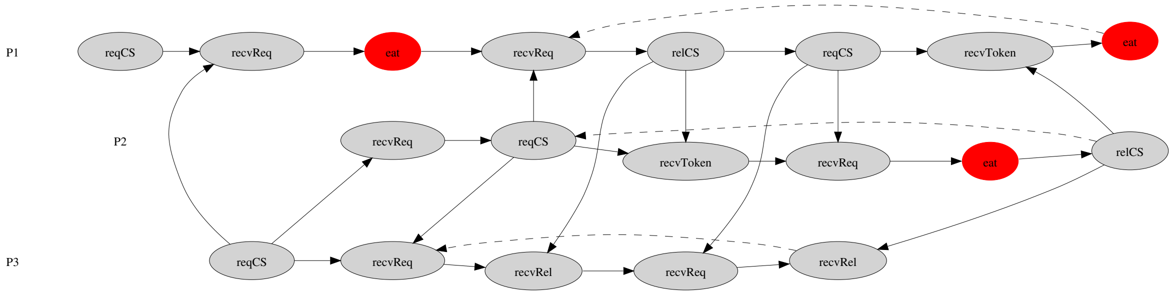

Fig. 5: A p-diagram derived from α that shows violation of a liveness property

or outgoing channels). The predecessor and successor events of e on the process on which e occurs are denoted by $pred(e)$ and $succ(e)$.

Generally a distributed computation is modeled as a partial order of a set of events, called the *happened-before relation* [12]. In this paper, we instead use directed graphs to model distributed computations as done in [8]. When the graph is acyclic, it represents a distributed computation. When the distributed computation is infinite, the directed graph that models the computation is also infinite. An infinite distributed computation is *periodic* if it consists of a subcomputation that is repeated forever. Directed graphs allow us to represent both the computation and its slice with respect to a predicate in a uniform fashion. However, as opposed to the earlier work, our computations can be infinite and as a result, the directed graphs used to model the computation can be infinite.

Given a directed graph $G = \langle E, \rightarrow \rangle$, we define a *consistent cut* as a set of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. For example, the set $C = \{a^1, b^1, c^1, d^1, e^1, f^1, g^1\}$ is a consistent cut for the graph shown in figure 8(b). The set $\{a^1, b^1, c^1, d^1, e^1, g^1\}$ is not consistent because it includes g^1, but does not include its incoming neighbor f^1. The set of finite consistent cuts for graph G is denoted by $\mathcal{C}(G)$.

In this work we focus only on finite consistent cuts (or *finite order ideals* [13]) as they are the ones of interest for distributed computing.

A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally,

$$frontier(C) = \{x \in C | succ(x) \text{ exists } \Rightarrow succ(x) \notin C\}$$

For the cut C in figure 8(b), $frontier(C) = \{e^1, f^1, g^1\}$. A consistent cut is uniquely characterized by its frontier and in this paper we always identify a consistent cut by its frontier.

Two events are said to be *consistent* iff 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 iff there is no path in the computation from $succ(e)$ to f and from $succ(f)$ to e.

4 Infinite Directed Graphs

From distributed computing perspective, our intention is to provide a model for an infinite computation of a distributed system which eventually becomes periodic. Although a distributed computation in the happened-before model [12] is always represented using a poset, it is useful to have a model of infinite directed graphs for the purpose of slicing [7, 8]. The directed graph does not capture the order between events in the computation but it is used to capture the set of possible consistent cuts or global states of the system. To this end, we introduce the notion of *d-diagram* (directed graph diagram).

Definition 1 (d-diagram). *A d-diagram Q is a tuple (V, F, R, B) where V is a set of vertices or nodes, F (forward edges) is a subset of $V \times V$, R (recurrent vertices) is a subset of V, and B (shift edges) is a subset of $R \times R$. A d-diagram must satisfy the following constraint: If u is a recurrent vertex and $(u, v) \in F$ or $(u, v) \in B$, then v is also recurrent.*

Figure 2(a) is an example of a d-diagram. The recurrent vertices and non-recurrent vertices in the d-diagram are represented by hollow circles and filled circles respectively. The forward edges are represented by solid arrows and the shift-edges by dashed arrows. The recurrent vertices model the computation that is periodic. Intuitively, a forward edge models dependency between the same instance of recurrent vertices, whereas a shift-edge (u, v) models dependency of instance $(i+1)$ of v on instance i of recurrent vertex u.

Each d-diagram generates an infinite directed graph defined as follows:

Definition 2 (directed graph for a d-diagram). *The directed graph $G = \langle E, \rightarrow \rangle$ for a d-diagram Q is defined as follows:*

- *$E = \{u^1 | u \in V\} \cup \{u^i | i \geq 2 \wedge u \in R\}$*
- *The relation $\rightarrow$ is the set of edges in E given by: (1) if $(u, v) \in F$ and $u \in R$, then $\forall i : u^i \rightarrow v^i$, and (2) if $(u, v) \in F$ and $u \notin R$, then $u^1 \rightarrow v^1$, and (3) if $(v, u) \in B$, then $\forall i : v^i \rightarrow u^{i+1}$.*

Furthermore, let $\mathcal{P}(G)$ be the set of pairs of vertices (u, v) such that there is a path from u to v in G.

The set E contains infinite instances of all recurrent vertices and single instances of non-recurrent vertices. For a vertex u^i, we define its index as i.

It can be easily shown that if the relation F is acyclic, then the resulting directed graph for the d-diagram is a poset. Figure 2 shows a d-diagram along with a part of the infinite directed graph generated by it. Two vertices in a directed graph are said to be *concurrent* if there is no path from one to other.

Note that acyclic d-diagrams cannot represent all infinite posets. For example, any poset P defined by an acyclic d-diagram is well-founded. Moreover, there exists a constant k such that every element in P has the size of the set of its upper covers and lower covers[13] bounded by k. Although acyclic d-diagrams cannot represent all infinite posets, they are sufficient for the purpose of modeling periodic distributed computations. Let the *width* of a directed graph be defined as the maximum size of a set of pairwise concurrent vertices. A distributed computation generated by a finite number of processes has finite width and hence we are interested in only those d-diagrams which generate finite width directed graphs. The following property of the posets generated by acyclic d-diagrams is easy to show.

Lemma 1. *A poset P defined by an acyclic d-diagram has finite width iff for every recurrent vertex there exists a cycle in the graph $(R, F \cup B)$ which includes a shift-edge.*

Proof. Let $k > 0$ be the number of shift-edges in the shortest cycle involving $u \in R$. By transitivity we know that there is a path from vertex u^i to u^{i+k} in R. Therefore, at most $k-1$ instances of a vertex $u \in R$ are concurrent. Since R is finite, the largest set of concurrent vertices is also finite.

Conversely, if there exists any recurrent vertex v that is not in a cycle involving a shift-edge, then v^i is concurrent with v^j for all i, j. Then, the set

$$\{v^i | i \geq 1\}$$

contains an infinite number of concurrent vertices. Thus, G has infinite width.

We assume that the process relation maps all the instances of a vertex in d-diagram to the same process i.e. $\forall e \in V : proc(e^i) = proc(e^j)$. As a result, we denote the process for an element $e^i \in E$ simply as $proc(e)$. Figure 2 shows a d-diagram and the corresponding computation. Note that for any two events x, y on a process, either there is a path from x to y or from y to x, i.e., two events on the same process are always ordered. To guarantee this condition, it can be easily shown that all the recurrent vertices in the d-diagram with the same process should form a cycle with exactly one shift-edge. We refer to these shift-edges as *process shift-edges*. We further assume that the d-diagram given to us has a non-empty recurrent part; otherwise the algorithms for the case of finite computations can be used. We augment the d-diagram with the presence of a fictitious *global initial event* denoted by $\perp$. The global initial event occurs before any other event on the processes and initializes the state of the processes. Note that for infinite computations we do not have the notion of a global final event.

4.1 Index Independence Assumption

Given a consistent cut, a predicate is evaluated with respect to the values of the variables resulting after executing all the events in the cut. If a predicate p evaluates to true for a consistent cut C, we say that C satisfies p. We further assume that a predicate depends only on the *labels* of the events in the computation. This rules out predicates based on global state such as channel predicates [20]. We define $\mathcal{L} : G \rightarrow L$ to be an onto mapping from the set of vertices in d-diagram to a set of labels L with the constraint that $\forall e \in V : \mathcal{L}(e^i) = \mathcal{L}(e^j)$. In other words, the label of an element in the directed graph is independent of its index. This is in line with modeling the recurrent events

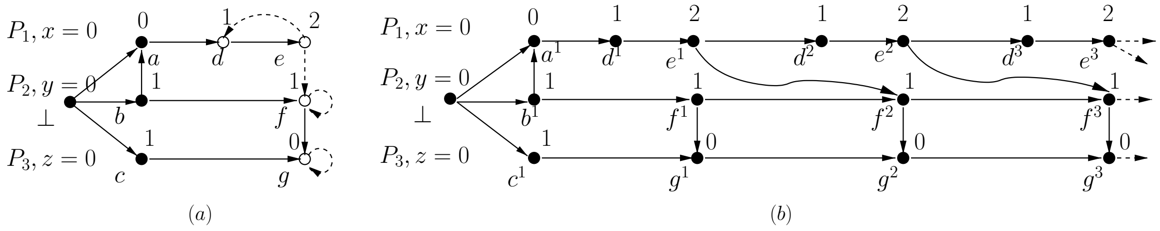

Fig. 6: (a) A d-diagram with process and label information (b) The computation corresponding to the d-diagram
v

as repetition of the same state. We refer to this assumption as the *index-independence assumption*.

Figure 6 shows a computation represented as a d-diagram. Figure 6(a) shows the d-diagram along with the process information. Along with each process, the local variables x, y, z on processes P_1, P_2 and P_3 are also listed. For each event the value of the local variable is listed. The values of local variables can be considered to be labels in this case.

4.2 Unrolling of a d-diagram

We now discuss another aspect of d-diagrams; a directed graph does not have a unique representation in terms of d-diagram. In fact, we show that there are countably infinite representations of a directed graph as a d-diagram. We define the notion of *unrolling* of a d-diagram Q with respect to a consistent cut C. Intuitively, this operation increases the non-recurrent part of the d-diagram to include the cut C and rearranges the forward and shift-edges.

Definition 3 (unrolling a d-diagram). *Let $Q = (V, R, F, B)$ be a d-diagram and C be a consistent cut in the corresponding directed graph $\langle E, \rightarrow\rangle$ such that $\forall e^i \in frontier(C) : e \in R$. Then $\mathscr{U}(Q, C) = (V', R', F', B')$ is the d-diagram:*

- $V' = \{e_k | e \in V \wedge e^k \in C\}$
- $R' = \{e_k | e \in R \wedge k - 1 = max\{i | e^i \in C\}\}$
- $F' = \{(e_i, f_j) | e_i, f_j \in V' \wedge e^i \rightarrow f^j\}$
- $B' = \{(e_i, f_j) | e_i, f_j \in R' \wedge e^i \rightarrow f^{j+1}\}$

It can be easily shown that the unrolled d-diagram generates a directed graph isomorphic to the original d-diagram. Let $\langle E, \rightarrow\rangle$ be the directed graph generated by the d-diagram Q and $\langle E', \rightsquigarrow\rangle$ be the directed graph generated by $\mathscr{U}(Q, C)$. Then we denote the isomorphism function from elements in E to elements in E' by $\mathcal{I}_C$. If the isomorphism function maps a node in the original directed graph to a node with the same label, then the two directed graphs are equivalent from the perspective of predicate detection. For this purpose, we assume that $\mathcal{L}(e^i) = \mathcal{L}(\mathcal{I}_C(e^i))$.

As an example, Figure 7 shows the unrolling of the d-diagram in Figure 6(a) with respect to a cut $C = \{d^2, f^1, g^1\}$.

4.3 Shift of a Cut

The notion of *shift* of a cut is useful for analysis of periodic infinite computations. Intuitively, the shift of a frontier C produces a new cut by moving the cut C forward or backward by a certain number of iterations along a set X of recurrent events in C. Formally,

Definition 4 (d-shift of a cut). *Given a frontier C, a set of recurrent events $X \subseteq R$ and an integer d, a* d-shift cut *of C with respect to X, is represented by the frontier $S_d(C, X)$*

$$\{e^i | e^i \in C \wedge e \notin X\} \cup \{e^m | e^i \in C \wedge e \in X \wedge m = \mathit{max}(1, i + d)\}$$

We denote $S_d(C, R)$ simply by $S_d(C)$.

Hence $S_d(C, X)$ contains all events e^i that are not in X, and the shifted events for all elements of X. Note that in the above definition d can be negative. Also, for a consistent cut C, $S_d(C, X)$ is not guaranteed to be consistent for every X.

As an example, consider the infinite directed graph for the d-diagram in figure 8. Let C be a cut given by the frontier $\{e^1, f^1, g^1\}$ and $X = \{f, g\}$. Then $S_1(C, X)$ is the cut given by $\{e^1, f^2, g^2\}$. Figure 8 shows the cut C and $S_1(C, X)$. Similarly for C given by $\{a^1, f^1, g^1\}$, $S_1(C) = \{a^1, f^2, g^2\}$. Note that in this case, a^1 remains in the frontier of $S_1(C)$ and the cut $S_1(C)$ is not consistent.

5 Vector Clock Timestamps

In this section, we present algorithms to assign vector timestamps to nodes in the infinite computation given as a d-diagram. The objective is to determine dependency between any two events, say e^i and f^j, based on the vector clocks assigned to these events rather than exploring the d-diagram. Since there are infinite instances of recurrent events, it is clear that we can only provide an implicit vector timestamp (or a function to compute vector timestamp) for the events. The explicit vector clock can be computed for any specific instance of i and j.

Timestamping events of the computation has many applications in debugging distributed programs [5]. Given the timestamps of recurrent events e and f, our algorithm enables answering queries of the form:

1. Are there any instances of e and f which are concurrent, i.e., are there indices i and j such that e^i is concurrent with f^j? For example, when e and f correspond to entering in the critical section, this query represents violation of the critical section.
2. What is the maximum value of i such that e^i happened before a given event such as f^{256}?
3. Is it true that for all i, e^i happened before f^i?

We show in this section, that there exists an efficient algorithm to timestamp events in the d-diagram. As expected, the vectors corresponding to any recurrent event e^i eventually become periodic. The difficult part is to determine the threshold after which the vector clock becomes periodic and to determine the period.

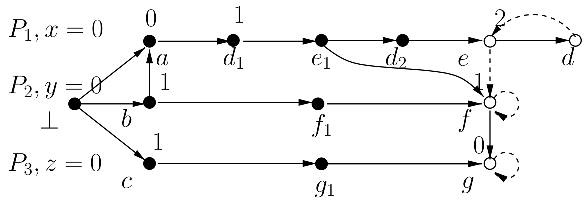

Fig. 7: The d-diagram in Figure 6(a) unrolled with respect to $C = \{d^2, f^1, g^1\}$

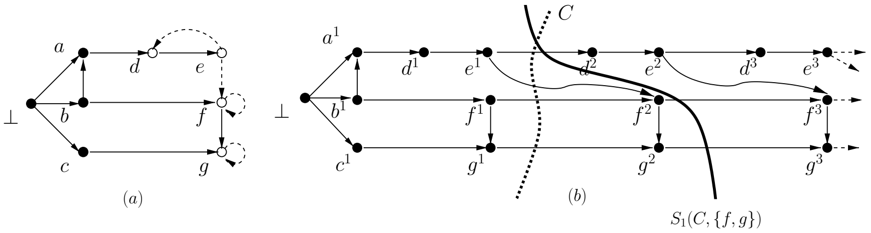

Fig. 8: (a) A d-diagram (b) The computation for the d-diagram showing a cut and the shift of a cut

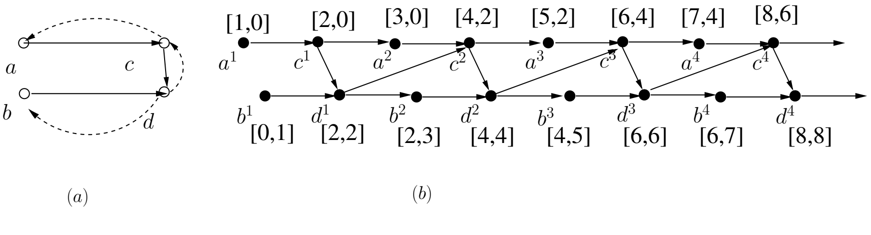

Fig. 9: (a) A d-diagram (b) The corresponding computation with vector timestamps

We first introduce the concept of *shift-diameter* of a d-diagram. The shift-diameter provides us with the threshold after which the dependency of any event becomes periodic.

Definition 5 (shortest path in a d-diagram). *For a d-diagram Q, the shortest path between any two vertices is a path with the minimum number of shift-edges.*

Definition 6 (shift-diameter of a d-diagram). *For a d-diagram Q, the shift-diameter $\eta(Q)$ is the maximum of the number of shift-edges in the shortest path between any two vertices in the d-diagram.*

When Q is clear from the context, we simply use η to denote $\eta(Q)$. For the d-diagram in Figure 8, $\eta = 1$. In figure 9, we can see that $\eta = 2$. We first give a bound on η.

Lemma 2. *For a d-diagram Q corresponding to a computation with N processes, $\eta(Q) \leq 2N$.*

Proof. Consider the shortest path between two vertices $e, f \in V$. Clearly this path does not have a cycle; otherwise, a shorter path which excludes the cycle exists. Moreover, all the elements from a process occur consecutively in this path. As a result, the shift-edges that are between events on the same process are traversed at most once in the path. Moving from one process to another can have at most one shift-edge. Hence, $\eta(Q) \leq 2N$.

For an event $x \in E$, we denote by $J(x)$, the least consistent cut which includes x. The least consistent cut for $J(e^i)$ will give us the vector clock for event e^i. We first show that the cuts $J(e^i)$ *stabilize* after some iterations i.e. the cut $J(e^j)$ can be obtained from $J(e^i)$ by a simple shift for $j > i$. This allows us to *predict* the structure of $J(e^i)$ after certain iterations.

The next lemma shows that the cut $J(f^j)$ does not contain recurrent events with iterations very far from j.

Lemma 3. *If $e^i \in frontier(J(f^j))$, $e \in R$, then $0 \leq j - i \leq \eta$.*

Proof. If $e^i \in frontier(J(f^j))$, then there exists a path from e^i to f^j and $\forall k > i$ there is no path from e^k to f^j. Therefore the path from e^i to f^j corresponds to the shortest path between e and f in the d-diagram. Therefore, by the definition of η, $j - i \leq \eta$.

The following theorem proves the result regarding the stabilization of the cut $J(e^i)$. Intuitively, after a first few iterations the relationship between elements of the computation depends only on the difference between their iterations.

Theorem 1. *For a recurrent vertex $e \in R$, $J(e^{\beta+1}) = S_1(J(e^\beta))$ for all $\beta \geq \eta + 1$.*

Proof. We first show that $S_1(J(e^\beta)) \subseteq J(e^{\beta+1})$. Consider $f^j \in S_1(J(e^\beta))$. If $f \in V \setminus R$ (i.e., f is not a recurrent vertex), then $f^j \in J(e^\beta)$, because the shift operator affects only the recurrent vertices. This implies that there is a path from f^j to e^β, which in turn implies the path from f^j to $e^{\beta+1}$. Hence, $f^j \in J(e^{\beta+1})$. If f is recurrent, then

$f^j \in S_1(J(e^\beta))$ implies $f^{j-1} \in J(e^\beta)$. This implies that there is a path from f^{j-1} to e^β, which in turn implies the path from f^j to $e^{\beta+1}$, from the property of d-diagrams. Therefore, $S_1(J(e^\beta)) \subseteq J(e^{\beta+1})$.

Now we show that $J(e^{\beta+1}) \subseteq S_1(J(e^\beta))$. Consider $f^j \in J(e^{\beta+1})$. If $j > 1$, then given a path from f^j to $e^{\beta+1}$, there is a path from f^{j-1} to e^β. Hence $f^j \in S_1(J(e^\beta))$. Now, consider the case when j equals 1. $f^1 \in J(e^{\beta+1})$ implies that there is a path from f^1 to $e^{\beta+1}$. We claim that for $\beta > \eta$, there is also a path from f^1 to e^β. Otherwise, the shortest path from f to e has more than η shift-edges, a contradiction.

When d-diagram generates a poset, Theorem 1 can be used to assign timestamps to vertices in the d-diagram in a way similar to vector clocks. The difference here is that a timestamp for a recurrent vertex is a concise way of representing the timestamps of infinite instances of that vertex.

Each recurrent event, e, has a special *p-timestamp* ($PV(e)$) associated with it, which lets us compute the time stamp for any arbitrary iteration of that event. Therefore, this result gives us an algorithm for assigning p-timestamp to a recurrent event. The p-timestamp for a recurrent event e, $PV(e)$ would be a list of the form

$$(V(e^1), \ldots, V(e^\beta); I(e))$$

where $I(e) = V(e^{\beta+1}) - V(e^\beta)$ and $V(e^j)$ is the timestamp assigned by the normal vector clock algorithm to event e^j. Now for any event $e^j, j > \beta$, $V(e^j) = V(e^\beta) + (j - \beta) * I(e)$.

In figure 9, $\eta = 2$, $\beta = 3$. $V(a^3) = [5, 2]$ and $V(a^4) = [7, 4]$. $I(a) = [2, 2]$. Hence $PV(a) = ([1, 0], [3, 0], [5, 2]; [2, 2])$. Now, calculating $V(a^j)$ for an arbitrary j is trivial. For example, if $j = 6$, then $V(a^6) = [5, 2] + (6 - 3) * [2, 2] = [11, 8]$.

This algorithm requires $O(\eta n)$ space for every recurrent vertex. Once the timestamps have been assigned to the vertices, any two instances of recurrent vertices can be compared in $O(n)$ time.

The notion of vector clock also allows us to keep only the *relevant* events[14] of the d-diagram. Any dependency related question on the relevant events can be answered by simply examining the vector timestamps instead of the entire d-diagram.

6 Detecting Global Predicates

We now consider the problem of detecting predicates in d-diagrams. A predicate is a property defined on the states of the processes. An example of a predicate is "more than one philosopher is waiting." Given a consistent cut, a predicate is evaluated with respect to the values of the variables resulting after executing all the events in the cut. If a predicate p evaluates to true for a consistent cut C, we say that C satisfies p. We further assume that the truthness of a predicate on a consistent cut is governed only by the *labels* of the events in the frontier of the cut. This assumption implies that the predicates do not involve shared state such as the channel state. We define $\mathcal{L} : G \rightarrow L$ to be an onto mapping from the set of vertices in d-diagram to a set of labels L with the constraint that $\forall e \in V : \mathcal{L}(e^i) = \mathcal{L}(e^j)$. This is in agreement with modeling the recurrent events as repetition of the same event.

It is easy to see that it does not suffice to detect the predicate on the d-diagram without unrolling it. As a simple example, consider figure 9, where though $\{a^1, d^1\}$ is not a consistent cut, but $\{a^2, d^1\}$ is consistent.

In this section, we define a finite extension of our d-diagram which enables us to detect any property that could be true in the infinite poset corresponding to the d-diagram. We show that it is sufficient to perform predicate detection on that finite part.

We mainly focus on the recurrent part of the d-diagram as that is the piece which distinguishes this problem from the case of finite directed graph. We identify certain properties of the recurrent part which allows us to apply the techniques developed for finite directed graphs to d-diagrams.

Predicate detection algorithms explore the lattice of global states in BFS order as in Cooper-Marzullo [2] algorithm, or a particular order of events as in Garg-Waldecker [15] algorithm. For finite directed graphs, once the exploration reaches the final global state it signals that the predicate could never become true. In the case of infinite directed graphs, there is no final global state. So, the key problem is to determine the stopping rule that guarantees that if the predicate ever becomes true then it would be discovered before the stopping point. For this purpose, we show that for every cut in the computation, a subgraph of the computation called the *core* contains a cut with the same label. The main result of this section is that the core of the periodic infinite computation is simply the set of events in the computation with iteration less than or equal to N, the number of processes.

Definition 7 (core of a computation). *For a d-diagram Q corresponding to a computation with N processes, we define $U(Q)$, the* core *of Q, as the directed graph given by the set of events $E' = \{e^j | e \in R \wedge 2 \le j \le N\} \cup \{e^1 | e \in V\}$ and the edges are the restriction of $\rightarrow$ to set E'.*

The rest of the section is devoted to proving the completeness of the core of a computation. The intuition behind the completeness of the core is as follows: For any frontier C, we can perform a series of shift operations such that the resulting frontier is consistent and lies in the core. We refer to this operation as a *compression* operation and the resulting cut is denoted by the frontier $\mathscr{C}(C)$. Figure 10 shows the cut $C = \{e^5, f^3, g^1\}$ and the compressed cut $\mathscr{C}(C) = \{e^3, f^2, g^1\}$.

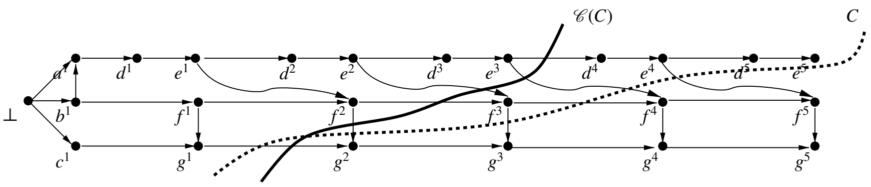

Fig. 10: Compression operation being applied on a cut

For proving the completeness of the core, we define the notion of a *compression* operation. Intuitively, compressing a consistent cut applies the shift operation multiple times such that the final cut obtained lies in the core of the computation and has the same labeling.

Definition 8 (Compression). *Given a frontier C and index i, define $\mathscr{C}(C, i)$ as shifting of all events with index greater than i by sufficient iterations such that in the shifted frontier the event with next higher index than i is $i + 1$. The cut obtained after all possible compressions is denoted as $\mathscr{C}(C)$.*

In Figure 10, consider the cut $C = \{e^5, f^3, g^1\}$. When we apply $\mathscr{C}(C, 1)$, we shift events e^5 and f^3 back by 1. This results in the cut $\{e^4, f^2, g^1\}$. The next higher index in the cut now is 2 in f^2. We now apply another compression at index 2, by shifting event e^4, and the compressed cut $\mathscr{C}(C) = \{e^3, f^2, g^1\}$. As another example, consider a cut $C = \{e^7, f^4, g^4\}$. We first apply $\mathscr{C}(C, 0)$ to get the cut $\{e^4, f^1, g^1\}$. Applying the compression at index 1, we finally get $\{e^2, f^1, g^1\}$.

Note that the cut resulting from the compression of a cut C has the same labeling as the cut C. The following lemma shows that it is safe to apply compression operation on a consistent cut i.e. compressing the gaps in a consistent cut results in another consistent cut. This is the crucial argument in proving completeness of the core.

Lemma 4. *If C is the frontier of a consistent cut, then $\mathscr{C}(C, l)$ corresponds to a consistent cut for any index l.*

Proof. Let $C' = \mathscr{C}(C, l)$ for convenience. Consider any two events $e^i, f^j \in C$. If $i \leq l, j \leq l$ or $i > l, j > l$, then the events corresponding to e^i and f^j in C' are also consistent. When $i > l$ and $j > l$, events corresponding to e^i and f^j in C' get shifted by the same number of iterations.

Now assume $i \leq l$ and $j > l$. Then e^i remains unchanged in C' and f^j is mapped to f^a such that $a \leq j$. Since $i < a$, there is no path from $succ(f^a)$ to e^i. If there is a path from $succ(e^i)$ to f^a, then there is also a path from $succ(e^i)$ to f^j as there is a path from f^a to f^j. This contradicts the fact that e^i and f^j are consistent. Hence, every pair of vertices in the cut C' is consistent.

Now we can use the compression operation to compress any consistent cut to a consistent cut in the core. Since the resulting cut has the same labeling as the original cut, it must satisfy any non-temporal predicate that the original cut satisfies. The following theorem establishes this result.

Theorem 2. *If there is a cut $C \in \mathcal{C}(\langle E, \rightarrow \rangle)$, then there exists a cut $C' \in \mathcal{C}(U(Q))$ such that $\mathcal{L}(C) = \mathcal{L}(C')$.*

Proof. Let $C' = \mathscr{C}(C)$. By repeated application of the lemma 4, we get that C' is a consistent cut and $\mathcal{L}(C) = \mathcal{L}(C')$. Moreover, by repeated compression, no event in C' has index greater than N. Therefore, $C' \in U(Q)$.

The completeness of the core implies that the algorithms for predicate detection on finite directed graphs can be used for d-diagrams as well after unrolling the recurrent events N times. This result holds for any global predicate that is non-temporal (i.e., defined on a single global state). Suppose that the global predicate B never becomes true in the core of the computation, then we can assert that there exists an infinite computation in which B never becomes true (i.e., the program does not satisfy that eventually B becomes true). Similarly, if a global predicate B is true in the recurrent part of the computation, it verifies truthness of the temporal predicate that B becomes true infinitely often.

7 Recurrent Global State Detection Algorithm

We now briefly discuss a method to obtain a d-diagram from a finite distributed computation. The *local state* of a process is the value of all the variables of the process including the program counter. The *channel* state between two processes is the sequence of messages that have been sent on the channel but not received. A *global state* of a computation is defined to be the cross product of local states of all processes and all the channel states at any cut. Any consistent cut of the computation determines a unique consistent global state. A global state is *recurrent* in a computation, if there exist consistent cuts Y and Z such that the global states for Y and Z are identical and Y is a proper subset of Z. Informally, a global state is recurrent if there are at least two distinct instances of that global state in the computation.

We now give an algorithm to detect recurrent global states of a computation. We assume that the system logs the message order and nondeterministic events so that the distributed computation can be exactly replayed. We also assume that the system supports a vector clock mechanism.

The first step of our recurrent global state detection (RGSD) algorithm consists of computing the global state of a distributed system. Assuming FIFO, we could use the classical Chandy and Lamport's algorithm[1] for this purpose. Otherwise, we can use any of the algorithms, such as [16–18]. Let the computed global snapshot be G. Let Z be the vector clock for the global state G.

The second step consists of replaying the distributed computation while monitoring the computation to determine the least consistent cut that matches G. We are guaranteed to hit such a global state because there exists at least one such global state (at vector time Z) in the computation. Suppose that the vector clock of the detected global state is Y. We now have two vector clocks Y and Z corresponding to the global state G. If Y equals Z, we continue with our computation. Otherwise, we have succeeded in finding a recurrent global state G.

Note that replaying a distributed computation requires that all nondeterministic events (including the message order) be recorded during the initial execution [19]. Monitoring the computation to determine the least consistent cut that matches G can be done using algorithms for conjunctive predicate detection [3, 20].

When the second step fails to find a recurrent global state, the first step of the algorithm is invoked again after certain time interval. We make the following observation about the recurrent global state detection algorithm.

Theorem 3. *If the distributed computation is periodic then the algorithm will detect a recurrent global state. Conversely, if the algorithm returns a recurrent global state G, then there exists an infinite computation in which G appears infinitely often.*

Proof. The RGSD algorithm is invoked periodically and therefore it will be invoked at least once in repetitive part of the computation. This invocation will compute a global state G. Since the computation is now in repetitive mode, the global state G must have occurred earlier and the RGSD algorithm will declare G as a recurrent global state.

We prove the converse by constructing the infinite computation explicitly. Let Y and Z be the vector clocks corresponding to the recurrent global state G. Our infinite computation will first execute all events till Y. After that it will execute the computation that corresponds to events executed between Y and Z. Since Y and Z have identical global state, the computation after Y is also a legal computation after Z. By repeatedly executing this computation, we get an infinite legal computation in which G appears infinitely often.

The proof of Theorem 3 also shows how we can construct a p-diagram given the recurrent global state G. Let Y and Z be the vector clocks corresponding to the global state G. All the events between Y and Z are modeled as recurrent vertices. For every process, we add shift-edges from the last event in Z to the first event after Y on that process. In addition, for every message that is in channel at the global state in Z, we add a shift edge from the send event on the sending process to the receive event. An example of this construction is Fig. 5. The global state after β and β followed by γ are recurrent. The computation in Fig: 9(a) shows an example in which the shift edge in p-diagram is from one process to the other.

It is important to note that our algorithm does not guarantee that if there exists any recurrent global state, it will be detected by the algorithm. It only guarantees that if the computation is periodic, then it will be detected.

We note here that RGSD algorithm is also useful in debugging applications in which the distributed program is supposed to be terminating and presence of a recurrent global state itself indicates a bug.

8 Computing Slice of a Computation

In this section we present an algorithm to compute slice of a periodic computation. We first provide a brief introduction to slicing, then we generalize the notion of d-diagrams by introducing *k-shift-edges* because the slice is more conveniently expressed using generalized d-diagram. Then, we give an algorithm to compute the slice.

8.1 Background on Slicing

In the earlier work [7, 8], the notion of slicing was used for finite directed graphs and was based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [21]. Informally, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate.

In this work, we extend the notion of slicing for infinite directed graphs by focusing only on finite order ideals [21]. As mentioned earlier as well, we deal only with finite consistent cuts and refer to them simply as consistent cuts. Similar to the the Birkhoff's representation theorem [21], the following theorem is known for the case of finite order ideals of an infinite poset:

Theorem 4. *[[22], Proposition 3.4.3] Let P be a poset such that every principal order ideal is finite. Then the poset $J_f(P)$ of finite order ideals of P, ordered by inclusion, is a finitary distributive lattice. Conversely, if L is a finitary distributive lattice and P is its subposet of join-irreducibles, then every principal order ideal of P is finite and $L = J_f(P)$.*

We can use the above theorem instead of the Birkhoff's theorem and generalize the notion of slice for consistent cuts to the following definition.

Definition 9 (slice). *The slice of a directed graph G with respect to a predicate p is the directed graph whose consistent cuts form the smallest sublattice that contains all the consistent cuts satisfying the predicate p.*

This definition is equivalent to the definition of slice in the earlier work for finite directed graphs. It can be easily shown that the slice for a predicate always exists and is unique. Since the slice of an infinite directed graph can again be an infinite directed graph, we use d-diagram to represent the slice of a computation as well. We later show that if the original computation was representable using d-diagram, then the slice of the computation is also representable by d-diagram.

We denote the slice of the computation $\langle E, \rightarrow \rangle$ with respect to a predicate p by $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$. Note that $\langle E, \rightarrow \rangle = \mathsf{slice}(\langle E, \rightarrow \rangle, true)$. Every slice derived from the computation $\langle E, \rightarrow \rangle$ has the trivial consistent cut $\emptyset$ among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts [8]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut.

In general, a slice contains consistent cuts that do not satisfy the predicate (besides trivial consistent cut $\emptyset$). In case a slice does not contain any such cut, it is called *lean*. The slice for the class of predicates called *regular predicates* is always lean. Given a computation, the set of consistent cuts satisfying a regular predicate forms a sublattice of the set of consistent cuts of the computation [7]. Equivalently,

Definition 10 (regular predicate [7]). *A predicate is regular if given two consistent cuts that satisfy the predicate, the consistent cuts obtained by their set union and set intersection also satisfy the predicate. Formally, given a regular predicate p,*
(C satisfies p) $\wedge$ (D satisfies p) $\implies$ ($C \cap D$ satisfies p) $\wedge$ ($C \cup D$ satisfies p)

Some examples of regular predicates are conjunction of local predicates such as "P_i and P_j are in critical section", and relational predicates such as $x_1 - x_2 \leq 5$, where x_i is a monotonically non-decreasing integer variable on process P_i. From the definition of a regular predicate we deduce that a regular predicate has a least satisfying cut as the lattice of order ideals has a bottom element. Furthermore, the class of regular predicates is closed under conjunction [7].

For a regular predicate p and an element $x \in G$, we define $J_p(x)$ as the least consistent cut which satisfies p and includes x. If there is no cut that satisfies p and includes x, $J_p(x)$ is defined to be $NULL$; otherwise $J_p(x)$ is well-defined as the regular predicates are closed under intersection.

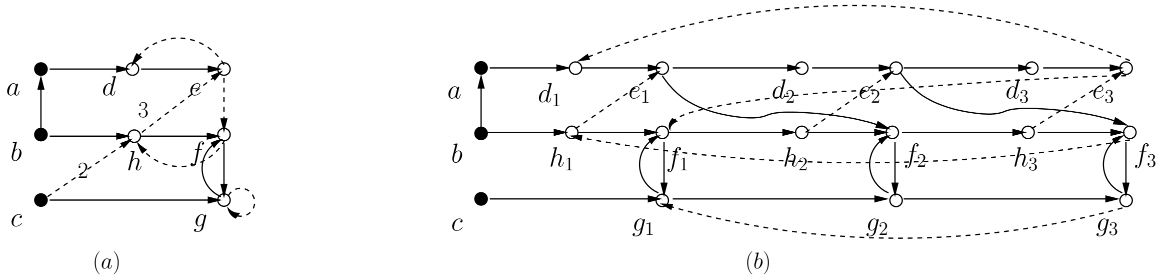

Fig. 11: (a) A generalized d-diagram and (b) its equivalent d-diagram

8.2 Generalized d-diagram

For computing slice of a d-diagram it is covenient to first generalize the d-diagrams by having *k-shift-edges* which increase the iteration of a vertex by k for any arbitrary k instead of a simple shift-edge. We call such a d-diagram as the *generalized d-diagram.*

Definition 11 (generalized d-diagram). *A generalized d-diagram is a set* $(V, R, F, B_1, B_2, \ldots, B_m)$ *where* m *is the maximum* k *such that a* k-shift-edges *is present in the d-diagram and* $B_k \subseteq (V \times R)$ *is the relation of* k-shift-edges *with the property:*
If $(e, f) \in B_k$*, then* $e^i \to f^{i+k}$*.*

Note that in the above definition, we allow the shift edges to be present between non-recurrent and recurrent vertices as well. Clearly d-diagrams are a special case of the generalized d-diagrams where $m = 1$ and $B_1 \subseteq R \times R$. The following theorem further shows that the generalized d-diagram are not any more expressive than the d-diagrams.

Theorem 5. *Let* $(V, R, F, B_1, B_2, \ldots, B_m)$ *be a generalized d-diagram and* $\langle E, \to\rangle$ *be the infinite graph generated by it. Then there exists a d-diagram* (V', R', F', B') *such that the graph* $\langle E', \leadsto\rangle$ *generated by the d-diagram is isomorphic to the graph* $\langle E, \to\rangle$*.*

Proof. We construct the d-diagram corresponding to the generalized d-diagram as follows.

1. $V' = \{e_1 | e \in V\} \cup \{e_i | e \in R \text{ and } 2 \leq i \leq m + 1\}$
2. $R' = \{e_i | e \in R \text{ and } 1 \leq i \leq m + 1\}$
3. $F' = \{(e_1, f_1) | (e, f) \in F\}$
   $\cup \{(e_i, f_{i+k}) | (e, f) \in B_k \text{ and } 1 \leq i \leq m + 1 - k\}$
4. $B' = \{(e_i, f_j) | (e, f) \in B_k \text{ and } m{+}1{-}k < i \leq m{+}1 \text{ and } j = (i{+}k) \bmod m{+}1)\}$

Now we show that the graph $\langle E', \leadsto\rangle$ generated by (V', R', F', B') is isomorphic to the graph $\langle E, \to\rangle$. Let $\mu : E \to E'$ be the isomorphism function between the elements in E and E' defined as follows:

1. $\mu(e^1) = e_1^1$ if $e \in V \setminus R$.

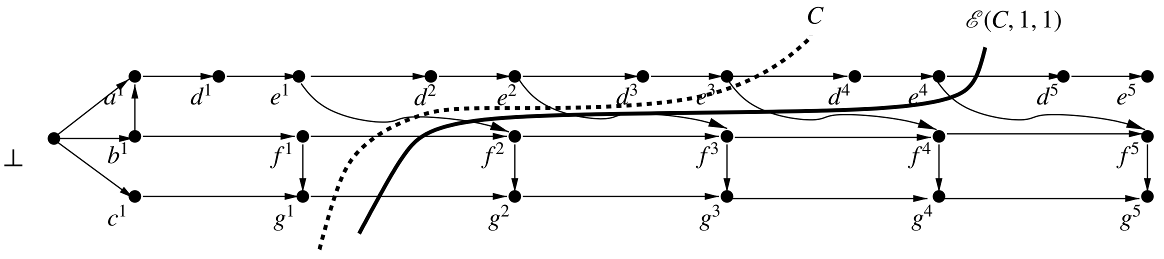

Fig. 12: Expansion operation being applied on a cut

2. $\mu(e^i) = e^l_k$ if $e \in R$. Here $k = 1 + (i-1) \bmod m+1$ and $l = 1 + (i-1)/(m+1)$.

With this isomorphism function, it is easy to show that $(e^i, f^j) \in \mathcal{P}(\langle E', \rightsquigarrow \rangle)$ iff $(\mu(e^i), \mu(f^j)) \in \mathcal{P}(\langle E, \rightarrow \rangle)$.

■

Figure 11 shows an example of this conversion. The dotted edges labeled with a number k denote the *k-shift-edges* and the unlabeled dotted edges denote the simple shift edges. We have essentially created multiple copies of the recurrent vertices and redrawn the *k-shift-edges* in terms of simple shift-edges.

8.3 Computing Join-Irreducibles of the Lattice Satisying p

We first define an expansion function analogous to the compression operation. The expansion function allows expansion of a cut while maintaining the labeling.

Definition 12 (expansion of a cut). *Given a cut G and i $\in indices(G)$, define $\mathscr{E}(G,i,j) = S_j(G, G_i)$. In other words, the operation $\mathscr{E}$ shifts all the events in G with iteration greater than i forward by j iterations.*

As opposed to the compression operation, expansion cannot be applied to any event in the frontier of a cut while maintaining its consistency. We define the notion of an *expansion point* in a cut which gives a safe way of expanding a cut.

Definition 13 (expansion point). *We define $\rho \in indices(G)$ to be an expansion point for a consistent cut G if $\forall e^i, f^j \in G$ with $i \le \rho$ and $j > \rho$, $(e^i, f^k) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$ for any $k \ge j$.*

As an example, let $C = \{e^3, f^1, g^1\}$ as shown in Figure 8. Then $\rho = 1$ is an expansion point as $f^1 \parallel e^3$ and $g^1 \parallel e^3$. The expansion operation $\mathscr{E}(G, \rho, 1)$ results in the frontier $\{e^4, f^1, g^1\}$ which is also consistent. The next lemma shows that this is true in general;a cut always allows expansion around the expansion point.

Lemma 5. *Let G be a consistent cut such that it has an expansion point ρ. Then the cut $G' = \mathscr{E}(G, \rho, l)$ is a consistent cut for any $l \ge 0$.*

Proof. We show that events in the frontier of the cut G' are consistent. Any two events $e^i, f^j \in G$ with $i \leq \rho, j \leq \rho$ or $i > \rho, j > \rho$ are still consistent in G' as the relationship between these events remains the same as in G. When $i \leq \rho$ and $j > \rho$, e^i is mapped to e^i and f^j is mapped to f^{j+l} in G'. By definition of an expansion point, $(e^i, f^{j+l}) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$. Hence, all the pairs of events in frontier of G' are consistent.

■

The above result provides a way to expand a cut while maintaining the consistency of the cut. Using the expansion and compression operations, we can move back and forth in the computation while maintaining the labeling of the cut.

While the compression and expansion operations are general operations which are applicable to any cut in the poset, we now examine the structure of the cuts $J_p(e^i)$. For this purpose, we define the notion of helper processes for an event e^i. Intuitively, there is an event f^j, $f \in R$, in every helper process such that there is a path from the event f^j to e^i. Therefore, as a cut advances along $proc(e)$, it must advance along the helper processes as well.

Definition 14 (helper processes). *For an event e^i, we define the helper processes for e^i as $H(e^i) = \{P_k | \exists f \in R \wedge f^j \in J(e^i) \wedge proc(f) = P_k\}$.*

For our example d-diagram in Figure 8, $H(f^2) = \{P_1, P_2\}$. Some properties of helper processes are easy to show.

Lemma 6. *The following properties relating to helper processes hold:*

- *For all $i, j > \eta$, $H(e^i) = H(e^j)$. For $i > \eta$, $H(e^i)$ is denoted simply by $H(e)$.*
- *For event e^i, f^i with $i > \eta$ and $proc(e) = proc(f)$, $H(e) = H(f)$.*
- *Let f^j be an event such that $proc(f) \notin H(e)$ and g^l be an event such that $proc(g) \in H(e)$ with $j < l$ and $l > 1$. Then $(f^j, g^l) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$.*

The first property says that the set of helper processes becomes fixed for different instances of an event after a certain number of iterations. This follows from the fact that the cut $J(e^i)$ stabilizes after some iterations. The second property says that after certain number of iterations, the set of helper processes becomes the same for every event on a process. This follows from the first property and the fact that there is a path between every two events in a process. The third property establishes the transitivity of the relation of helper processes.

Now we can give a characterization of the cut $J_p(e^i)$ in terms of the helper processes. In all the following results, we assume that $J_p(e^i)$ exists for the event e^i under consideration. We deal with the case when $J_p(e^i)$ does not exist later. The following lemma shows that after certain iterations only the events from helper processes in $frontier(J_p(e^i))$ have iterations "close" to i and other events always have iterations less than $N - |H(e)|$. This follows our intuition that helper processes advance along with the event e^i.

Lemma 7. *For $i > N - |H(e)| + \eta$, $frontier(J_p(e^i))$ can be written as $C \cup D$, where*

1. *$f^j \in C \Rightarrow (proc(f) \notin H(e)) \wedge (j \leq N - |H(e)|)$*

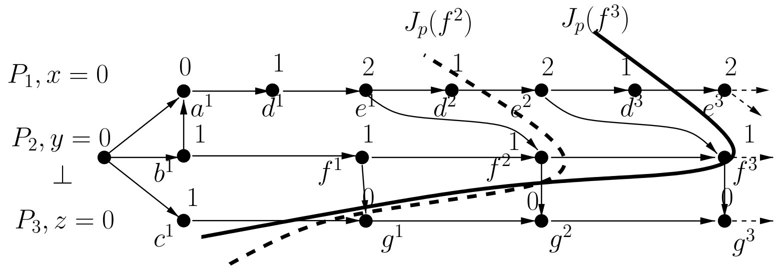

Fig. 13: The cuts $J_p(f^2)$ and $J_p(f^3)$ where p is $(x = 1) \wedge (y = 1) \wedge (z = 1)$

2. $f^j \in D \Rightarrow (proc(f) \in H(e)) \wedge (j > N - |H(e)|)$

Proof Sketch: Let C' be the projection of $frontier(J_p(e^i))$ on the set of processes $P \setminus H(e)$. Then let $C = \mathscr{C}(C')$ and D be the projection of $frontier(J_p(e^i))$ on the processes $H(e)$. Since $J(e^i) \subseteq J_p(e^i)$, for an event $f^j \in D$, $i - j \leq \eta$. Therefore, if $f^j \in D$, $j > N - |H(e)|$. On the other hand, C must belong to the core $U(Q)$ and therefore, if $f^j \in C$, then $j \leq N - |H(e)|$. This further implies that for all $f^j \in C$ and $g^l \in D$, $j < l$. By Lemma 6, this implies that f^j and g^l are consistent. Therefore, every pair of vertices in the set $C \cup D$ is consistent and so it forms the frontier of a consistent cut. Let the cut given by $C \cup D$ be C''. Then, $\mathcal{L}(J_p(e^i)) = \mathcal{L}(C'')$ and so C'' also satisfies the predicate p. It includes e^i as $e^i \in D$. Therefore, C'' is a consistent cut which includes e^i and satisfies p. By definition of $J_p(e^i)$, $J_p(e^i) \subseteq C''$. However, $C'' \subseteq J_p(e^i)$ as C'' was obtained by applying compression on the cut $J_p(e^i)$. Therefore, $J_p(e^i) = C''$. ■

Figure 13 shows the cuts $J_p(f^2)$ and $J_p(f^3)$ where p is $(x = 1) \wedge (y = 1) \wedge (z = 1)$. Here $H(f^3) = \{P_1, P_2\}$ and so we can decompose $J_p(f^3)$ into subsets C and D of Lemma 7 as $C = \{c^1\}$ and $D = \{d^3, f^3\}$.

The above result can also be interpreted in terms of the presence of an expansion point in the cut $J_p(e^i)$ as the events in $J_p(e^i)$ from the helper processes lie after iteration N and are disconnected from the rest of the vertices in $J_p(e^i)$. Henceforth, we relax the condition on i in Lemma 7 to $i > N + \eta$. This makes the bound independent of e and allows us to deal with all vertices uniformly.

Corollary 1. *If $i > N + \eta$, then $J_p(e^i)$ contains an expansion point $\rho \leq N$.*

The next theorem shows that the presence of an expansion point in the cut $J_p(e^i)$ is a sufficient condition for $J_p(e^i)$ to acquire a repeating structure.

Theorem 6. *Let i be the smallest iteration such that $J_p(e^i)$ has an expansion point ρ. Then for all $j \geq i$, $J_p(e^{j+1}) = \mathscr{E}(J_p(e^j), \rho, 1)$.*

Proof. Consider $C = \mathscr{E}(J_p(e^i), \rho, 1)$. By Lemma 5, C is consistent. Since C includes e^{i+1} and satisfies B, $J_p(e^{i+1}) \subseteq C$. Moreover, $J_p(e^i) \subseteq J_p(e^{i+1})$. As a result, $f^j \in frontier(J_p(e^{i+1}))$ can be characterized as follows:

1. If $j \leq \rho$, then $f^j \in frontier(J_p(e^i))$
2. If $j > \rho$, then $\exists g^k \in frontier(J_p(e^i)) : g^k \leq f^j \leq g^{k+1}$ and $proc(g) = proc(f)$. This also implies that $j = k$ or $j = k + 1$.

Now consider $D = \mathscr{C}(J_p(e^{i+1}, \rho, 1)$. D is a consistent cut, includes e^i and satisfies B. Therefore, $J_p(e^i) \subseteq D$. Based on the operation $\mathscr{C}$, an event $f^j \in frontier(D)$, satisfies the following:

1. If $j \leq \rho$, then $f^j \in frontier(J_p(e^{i+1}))$
2. If $j > \rho$, then $f^{j+1} \in frontier(J_p(e^{i+1}))$

From the characterization of $J_p(e^{i+1})$, it is clear that for $j \leq \rho$, $f^j \in frontier(J_p(e^i))$. For $j > \rho$, we have $g^k \leq f^j \leq f^{j+1} \leq g^{k+1}$ for some $g^k \in frontier(J_p(e^i))$. This implies that $f^j = g^k$ and so $f^j \in frontier(J_p(e^i))$. Therefore, $D = J_p(e^i)$ and also, $C = J_p(e^{i+1})$. Inductively using the above argument for $j > i$, we get that for all $j \geq i$, $J_p(e^{j+1}) = \mathscr{E}(J_p(e^j), \rho, 1)$.

■

For the computation in Figure 13, it can be seen that $J_p(f^2)$ has an expansion point at $\rho = 1$. Therefore, we can write $J_p(f^3) = \mathscr{E}(J_p(f^2), 1, 1)$.

The above result essentially establishes the correspondence between the cuts $J_p(e^i)$ and $J_p(e^{i+1})$ for some large enough i. It says that the cuts have the same iteration of some elements and for the others, the iterations differ by exactly one. Hence the structure of $J_p(e^i)$ becomes repetitive once $J_p(e^i)$ has an expansion point. Corollary 1 gives an upper bound $(N + \eta)$ on the expansion point for $J_p(e^i)$. Also, note that this upper bound is independent of the predicate p and just depends on the d-diagram. For a d-diagram, let γ be the maximum over the expansion points of all the recurrent events. Again, γ is bounded by $N + \eta$.

Lemma 8. *Let $J_p(e^i) \subseteq J_p(f^j)$ with $i, j > \gamma$. Then $J_p(e^{i+1}) \subseteq J_p(f^{j+1})$.*

Proof. By Lemma 7, $J_p(e^i)$ can be written as $C_1 \cup D_1$ such that $g^k \in C_1 \Rightarrow (proc(g) \notin H(e)) \wedge (k \leq N - |H(e)|)$ and $g^k \in D_1 \Rightarrow (proc(g) \in H(e)) \wedge (k > N - |H(e)|)$. Similarly, $J_p(f^j)$ can be written as $C_2 \cup D_2$ with similar constraints. Since $J_p(e^i) \subseteq J_p(f^j)$, the above constraints imply that $C_1 \subseteq C_2$ and $D_1 \subseteq D_2$. Furthermore, $J_p(e^{i+1})$ and $J_p(f^{j+1})$ can be decomposed as $C_3 \cup D_3$ and $C_4 \cup D_4$ respectively. By theorem 6, $C_1 = C_3$ and $C_2 = C_4$. Moreover, if $g^k \in D_1 \Rightarrow g^{k+1} \in D_3$ and similarly, $g^k \in D_2 \Rightarrow g^{k+1} \in D_4$. Therfore, $D_1 \subseteq D_2 \Rightarrow D_3 \subseteq D_4$. Therefore, $J_p(e^{i+1}) \subseteq J_p(f^{j+1})$.

■

8.4 A Slicing Algorithm

A slice of a computation can have multiple representations. The *skeletal representation* [8] of the slice has the advantage of having at most N edges per event in the graph and provides us a compact representation. Let $F_p(e^i, k)$ denote the earliest event f^k on P_j such that $J_p(e^i) \subseteq J_p(f^j)$. If no such event exists, then $F_p(e^i, k)$ is considered to be $NULL$. Then the skeletal representation of $\mathsf{slice}(\langle E, \rightarrow\rangle, p)$ can be obtained by constructing a graph $\langle E', \rightsquigarrow\rangle$ as follows: (1) $E' = E \setminus D$ where D is the set of events e^i for which $J_p(e^i)$ does not exist (2) For an event $e^i \in E'$, edges are added are from e^i to $succ(e^i)$ and to $F_p(e^i, k)$ for every process P_k. Again we want to represent $\mathsf{slice}(\langle E, \rightarrow\rangle, p)$ in terms of a d-diagram as $\mathsf{slice}(\langle E, \rightarrow\rangle, p)$ may be an infinite graph.

We now show the procedure to obtain the slice of the computation $\langle E, \rightarrow\rangle$ with respect to the predicate p. For this purpose, we define the notion of *reset cut*.

Definition 15 (reset cut). *Let $T \subseteq R$ be the set of recurrent vertices e for which $J_p(e^\gamma)$ exists. Then the reset cut, $\mathcal{R}(p)$, for a predicate p is the consistent cut given by $\{\bigcup_{e \in T} J_p(e^\gamma)\} \cup \{\bigcup_{e \in R \setminus T} J(e^\gamma)\}$.*

Note that in the definition of the reset cut we use $J(e^\gamma)$ if $J_p(e^\gamma)$ does not exist for some e^γ. This is done to ensure that the frontier of the reset cut always contains elements corresponding to recurrent vertices. The reset cut is a consistent cut as it is a union of consistent cuts. For our example computation, it can be shown that $\gamma = 3$. The following cuts can also be computed easily:

- $J_p(d^3) = \{d^3, b^1, c^1\}$
- $J_p(e^3) = \{d^4, b^1, c^1\}$
- $J_p(f^3) = \{d^3, f^3, c^1\}$
- $J_p(g^3) = \emptyset$, $J(g^3) = \{e^2, f^3, g^3\}$
- $J_p(a^1) = \{d^1, b^1, c^1\}$
- $J_p(b^1) = \{d^1, b^1, c^1\}$
- $J_p(c^1) = \{d^1, b^1, c^1\}$

Then the reset cut $\mathcal{R}(p)$ is given by $\{d^4, f^3, g^3\}$.

The d-diagram obtained by unrolling Q about $\mathcal{R}(p)$, $\mathscr{U}(Q, \mathcal{R}(p))$, has a crucial property:

Lemma 9. *Let $\langle E', \rightsquigarrow\rangle$ be the directed graph generated by $\mathscr{U}(Q, \mathcal{R}(p))$. Let $e^i \in \langle E', \rightsquigarrow\rangle$. Then $J_p(f^j) \not\subseteq J_p(e^i)$ for all $f^j \in \langle E', \rightsquigarrow\rangle$ with $j > i$.*

Proof. Let the d-diagram $\mathscr{U}(Q, \mathcal{R}(p))$ be (V', R', F', B'). Consider $g^k, h^l \in \langle E, \rightarrow\rangle$ such that $J_p(g^k) \subseteq J_p(h^l)$. If $h^l \in \mathcal{R}(p)$, then by definition of $\mathcal{R}(p)$, $g^k \in \mathcal{R}(p)$ and hence $\mathcal{I}_{\mathcal{R}(p)}(h^l) = a^1$ and $\mathcal{I}_{\mathcal{R}(p)}(g^k) = b^1$ for some $a, b \in V'$. Now consider the case when $h^l \notin \mathcal{R}(p)$. Let $m = \max\{i | h^i \in \mathcal{R}(p)\}$ and $n = \max\{i | g^i \in \mathcal{R}(p)\}$. First assume that $l - m < k - n$. Then $J_p(g^r) \subseteq J_p(h^m)$ where $r = k - (l - m)$. However, in this case $h^m \in \mathcal{R}(p)$ and $g^r \notin \mathcal{R}(p)$ which violates the definition of $\mathcal{R}(p)$. Therefore, $l - m \geq k - n$. For this case, if $\mathcal{I}_{\mathcal{R}(p)}(h^l) = a^i$ and $\mathcal{I}_{\mathcal{R}(p)}(g^k) = b^j$, then $i = l - m$ and $j = k - n$. Therefore, $\forall e^i, f^j \in \langle E', \rightsquigarrow\rangle$, $J_p(f^j) \subseteq J_p(e^i) \Rightarrow j \leq i$.

■

This property is similar to the property of the d-diagram where it is required that there should not be any edge from an element with higher iteration to an element with lower iteration. Figure 14 shows the unrolled d-diagram $\mathscr{U}(Q, \mathcal{R}(p))$. Note that the process shift-edge on the process P_1 is now going from d to e instead of from e to d.

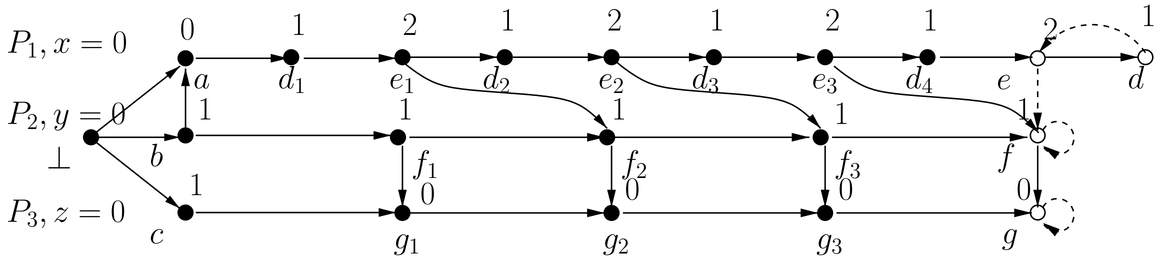

Fig. 14: The unrolled d-diagram $\mathscr{U}(Q, \mathcal{R}(p))$

This change has essentially renumbered the indices of elements corresponding to d and e such that in the new d-diagram, $J_p(e^1) = \{d^1, b^1, c^1\}$.

We can now construct the slice of the original d-diagram Q using the unrolled d-diagram $\mathscr{U}(Q, \mathcal{R}(p))$ as both of them generate the same computation. We slice the computation $\mathscr{U}(Q, \mathcal{R}(p))$ by introducing additional edges in the computation and removing vertices for which $J_p(e^i)$ does not exist. Intuitively, the additional edges make the cuts which do not satisfy the predicate, inconsistent in the new computation and removing the elements e^i for which $J_p(e^i)$ does not exist is equivalent to removing any consistent cut which includes the element e^i. Note that if an element e^i is removed from the slice, then any element f^j which has a path from e^i must be removed as well.

Using these results, we can construct a *generalized* d-diagram for the slice of $\mathscr{U}(Q, \mathcal{R}(p))$ as follows. The set of vertices V' in the d-diagram for $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ is the set $V \setminus D$ where V is the set of vertices for $\mathscr{U}(Q, \mathcal{R}(p))$ and D is the set of vertices for which $J(e^1)$ does not exist. The set R' is also constructed similarly using the set of recurrent vertices in $\mathscr{U}(Q, \mathcal{R}(p))$. Note that if e was a recurrent vertex, then by removing e we are not only removing the element e^1 from the computation but all $e^i, i \geq 1$. However, if $J_p(e^1)$ does not exist then $J_p(e^i)$ does not exist as e^i is reachable from e^1. On the other hand, if $J_p(e^1)$ exists, then $J_p(e^i)$ exists for all e^i as the cut $J_p(e^1)$ has stabilized in $\mathscr{U}(Q, \mathcal{R}(p))$.

Edges are added between the vertices in V' based on the skeletal representation of $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ as follows:
(1) If there is an edge from $J_p(e^1)$ to $J_p(f^1)$, then add a forward edge from e to f.
(2) If there is an edge from $J_p(e^1)$ to $J_p(f^j)$, then add a $j-1$ shift edge from e to f.

Intuitively, it is safe to add these edges in the recurrent part of the d-diagram as the cuts $J_p(e^i)$ have stabilized and so the relationships between the cuts acquires a repeating structure. Note that we would not need to add an edge from a recurrent element to $\perp$ as the cuts $J_p(e^i)$ have stabilized and therefore for any $e^i, i \geq 2$, e^{i-1} can serve as the lower cover. The following result is a direct consequence of the construction.

Theorem 7. *The generalized d-diagram as constructed above represents* $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$

Proof. Follows from the construction and Lemma 8.

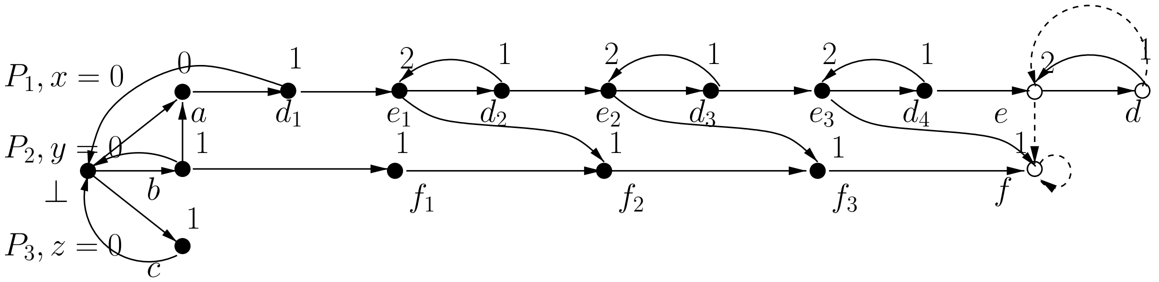

Fig. 15: The slice of the d-diagram in Figure 6 with respect to $(x = 1) \wedge (y = 1) \wedge (z = 1)$

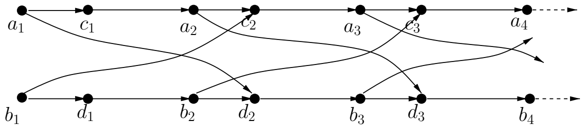

Fig. 16: A poset which cannot be captured using MSC graphs or HMSC

■

Figure 15 shows the slice of the original d-diagram with respect to the predicate $p = (x = 1) \wedge (y = 1) \wedge (z = 1)$. Note that in the slice, we have removed the recurrent vertex g and added some edges according to the above rules.

The above construction can be done in time polynomial in the size of the d-diagram as it requires unrolling the d-diagram for a polynomial number of iterations and then computing the $J_p(e^i)$ for a polynomial set of elements. Furthermore, if the slice of the computation becomes finite during the processing of a predicate, then we can simply use the algorithms for the finite directed graphs directly on the new poset. This is safe to do as slicing with respect to a regular predicate only adds edges to the slice.

9 Related Work

A lot of work has been done in identifying the classes of predicates which can be efficiently detected [9, 8]. However, most of the previous work in this area is mainly restricted to finite traces.

Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [9, 23], *disjunctive* [9], *observer-independent* [24, 9], *linear* [9], *non-temporal regular* [7, 8] predicates and *temporal* [10, 25, 26].

Some representations used in verification explicitly model concurrency in the system using a partial order semantics. Two such prominent models are message sequence charts (MSCs) [27] and petri nets [28]. MSCs and related formalisms such as time sequence diagrams, message flow diagrams, and object interaction diagrams are often used to specify design requirements for concurrent systems. An MSC represents one (finite) execution scenario of a protocol; multiple MSCs can be composed to depict more complex scenarios in representations such as MSC graphs and high-level MSCs (HMSC). These representations capture multiple posets but they cannot be used to model all the posets (and directed graphs) that can be represented by d-diagrams. In particular, a message sent in a MSC node must be received in the same node in MSC graph or HMSC. Therefore, some infinite posets which can be represented through d-diagrams cannot be represented through MSCs. Therefore, an infinite poset such as the one shown in figure 16 is not possible to represent through MSCs.

Petri nets [28] are also used to model concurrent systems. Partial order semantics in petri nets are captured through net unfoldings [29]. Unfortunately, unfoldings are usually infinite sets and cannot be stored directly. Instead, a finite initial part of the unfolding, called the finite complete prefix [30] is generally used to represent the unfolding. McMillan showed that reachability can be checked using the finite prefix itself. Later Esparza [31] extended this work to use unfoldings to efficiently detect predicates from a logic involving the $\mathbf{EF}$ and $\mathbf{AG}$ operators. Petri nets are more suitable to model the behavior of a complete system whereas d-diagrams are more suitable for modeling distributing computations in which the set of events executed by a process forms a total order. They are a simple extension of process-time diagrams[12] which have been used extensively in distributed computing literature.

10 Conclusions

In this paper, we introduce a method for detecting violation of liveness properties in spite of observing a finite behavior of the system. Our method is based on (1) determining recurrent global states, (2) representing the infinite computation by a d-diagram, (3) computing vector timestamps for determining dependency and (4) computing the core of the computation for predicate detection. We note here that intermediate steps are of independent interest. Determining recurrent global states can be used to detect if a terminating system has an infinite trace. Representing an infinite poset with d-diagram is useful in storing and replaying an infinite computation.

Our method requires that the recurrent events be unrolled N times. For certain computations, it may not be necessary to unroll recurrent event N times. It would be interesting to develop a method which unrolls each recurrent event just the minimum number of times required for that prefix of the computation to be core.

11 Acknowledgments

We are grateful to the reviewers of this paper for many helpful comments and suggestions.

References

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

2. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, ACM/ONR (1991) 163–173
3. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. on Parallel and Distributed Systems **5**(3) (1994) 299–307
4. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual IEEE-ACM Symposium on Foundations of Computer Science. (1977) 46–57
5. Fidge, C.J.: Partial orders for parallel debugging. Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices **24**(1) (1989) 183–194
6. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms. (1989)
7. Garg, V.K., Mittal, N.: On Slicing a Distributed Computation. In: Proc. of the 15^{th} Int'l. Conference on Distributed Computing Systems (ICDCS). (2001)
8. Mittal, N., Garg, V.K.: Computation Slicing: Techniques and Theory. In: In Proc. of the 15^{th} Int'l. Symposium on Distributed Computing (DISC). (2001)
9. Garg, V.K.: Elements of Distributed Computing. John Wiley & Sons (2002)
10. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: 7th International Conference on Principles of Distributed Systems, La Martinique, France (2003)
11. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In Pelc, A., ed.: Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings. Volume 4731 of Lecture Notes in Computer Science., Springer (2007) 420–434
12. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7) (1978) 558–565
13. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK (1990)
14. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in shared-memory systems. In Aguilera, M.K., Aspnes, J., eds.: PODC, ACM (2005) 19–28
15. Garg, V.K., Waldecker, B.: Detection of unstable predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, ACM/ONR (1991)
16. Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel and Distributed Computing (1993) 423–434
17. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: Proceedings of the ACM Conference on Supercomputing, 2006, ACM (2006)
18. Kshemkalyani, A.D.: A symmetric o(n log n) message distributed snapshot algorithm for large-scale systems. In: CLUSTER, IEEE (2009) 1–4
19. LeBlanc, Mellor-Crummey: Debugging parallel programs with instant replay. IEEETC: IEEE Transactions on Computers **36** (1987)
20. Garg, V.K., Chase, C.M., Kilgore, R.B., Mitchell, J.R.: Efficient detection of channel predicates in distributed systems. J. Parallel Distrib. Comput. **45**(2) (1997) 134–147
21. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK (1990)
22. Stanley, R.: Enumerative Combinatorics. Wadsworth and Brookes/Cole (1986)
23. Hurfin, M., Mizuno, M., Raynal, M., Singhal, M.: Efficient detection of conjunctions of local predicates. IEEE Transactions on Software Engineering **24**(8) (1998) 664–677
24. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and temporal predicates in distributed systems. ACM Transactions on Programming Languages and Systems **17**(1) (1995) 157–179
25. Sen, A., Garg, V.K.: Detecting temporal logic predicates in the happened before model. In: International Parallel and Distributed Processing Symposium (IPDPS), Florida (2002)

26. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In Pelc, A., ed.: DISC. Volume 4731 of Lecture Notes in Computer Science., Springer (2007) 420–434
27. : Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC). (1996)
28. Petri, C.A.: Kommunikation mit Auto-maten. PhD thesis, Bonn: Institut fuer Instru- mentelle Mathematik (1962)
29. M. Nielsen, G.P., Winskel, G.: Petri nets, event structures and domains. Theoretical Computer Science **13**(1) (1980) 85–108
30. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
31. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming **23**(2) (1994) 151–195