Predicate Detection on Infinite Computations

Anurag Agarwal
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-0233
anurag@cs.utexas.edu

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

Abstract

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 theory and associated algorithms for predicate detection in infinite runs. We introduce the concept of *d-diagram* which is a finite representation of infinite directed graphs. We extend the techniques of *computation slicing* to d-diagrams which allows us to efficiently detect predicates in a subclass of CTL, called RCTL. RCTL is sufficient to represent properties such as violation of liveness.

1 Introduction

Writing correct distributed programs is a difficult task. Concurrency between processes makes the programs hard to reason about, leading to errors. Testing and verification are two prominent techniques used to avoid such errors. For both testing and verification, the user provides a specification which the program should always meet. Testing techniques execute multiple runs of the program, possibly guiding the execution and check the runs for adherence to the specification. Verification techniques [EC99] build a model of the program and check the program for the specification.

Testing techniques for distributed programs generally model a finite trace as a partial order model between events, using for example Lamport's *happened-before* relation [Lam78]. The problem of testing then reduces to doing *predicate detection* on these partial order traces. The main problem in predicate detection in the partial order model is that of *state explosion*—the set of possible global states of a distributed program with N sequential processes can be of size exponential in N. Since the problem of detecting any general predicate is NP-complete [N. 01], the idea is to identify classes of predicates which can detected efficiently [Gar02, SG03, GM01, MG01]. Testing techniques have the advantage that they scale well but may not cover all possible scenarios. As a result, a program may have bugs even after successfully passing the tests.

On the other hand, if a program successfully passes through a verification procedure, it is guaranteed to be correct with respect to the properties that are verified. However, in spite of the recent advances in the formal methods, verification very quickly becomes intractable. This is especially true for verification of distributed programs although a variety of strategies for ameliorating the state explosion problem, including symbolic representation of states and partial order reduction have been explored [McM93, GW91, Val90, Pel93, Esp94].

In this work, we take a middle path; we capture a subset of infinite behaviors of a distributed program using a compact representation and extend the techniques used for testing finite traces to our representation. In particular, we draw on the work related to *computation slicing* [GM01, MG01, SG03] to perform efficient predicate detection for infinite computations. A computation slice, defined with respect to a global predicate, is the computation with the least number of global states that contains all global states of the original computation for which the predicate evaluates to true. Computation slicing can be used to throw away the *extraneous* global states of the original computation in an efficient manner, and focus on only those that are currently *relevant* for our purpose.

In this paper, we introduce the concept of *d-diagrams* (directed graph diagrams). The d-diagrams are essentially finite representations of a class of infinite directed graphs. To our knowledge, this is the first attempt to develop a finite representation for infinite directed graphs. Although in some ways d-diagrams are similar to other well known representations such as petri-nets [Pet62] and message sequence charts [ITS96], there are some

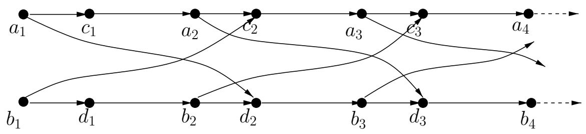

Figure 1: A poset which cannot be captured using MSC graphs or HMSC

important distinctions. Discussion of these differences is deferred to Section 2. Along with being of independent interest mathematically, d-diagrams can be used to represent a subset of possibly infinite behaviors of a distributed program.

We also show that it is possible to compute the slice of a d-diagram with respect to a predicate. The class of predicates considered in this work is a subset of CTL [CE81], called RCTL [SG03]. In RCTL, the temporal operators are **EF**, **EG**, and **AG** and the boolean operator is conjunction. Moreover, atomic propositions are regular predicates [GM01]. The class of regular predicates has the property that the slice with respect to a regular predicate contains *precisely* those global states for which the predicate evaluates to true. Some examples of regular predicates are conjunction of local predicates [Gar02] such as "all processes are in *red* state" and some relational predicates [Gar02].

As an example, the RCTL logic can be used to represent violation of progress properties. These properties are of the form $\mathbf{EF}(p \wedge \mathbf{EG}(q))$. Intuitively, this property says that there exists a global state of the system on which p is true and there is a path from that state on which q always remains true. As a concrete example, in the case of dining philosophers, p can be the predicate that "philosopher is hungry" and q can be the predicate "philosopher does not get to eat". Then the above property represents the violation of the progress property that "If a philosopher is hungry, he would eventually get to eat". In addition slicing can be used in conjunction with other general techniques to efficiently detect properties of the form $p \wedge q$ where p is a formula in the RCTL logic and q is any arbitrary formula.

Our slicing algorithms exploit the structure of the predicate and are more efficient than currently used techiques in model checking programs. Recently, a model checking algorithm was proposed [KG05] which represents a distributed program as a collection of infinite posets and checks for reachability properties in that representation. The results there show that exploiting the predicate structure can drastically improve the performance of the model checking algorithms. Our work opens up the avenues for representing a distributed program as a collection of d-diagrams and be able to check a larger set of properties on them.

In summary, this paper makes the following contributions:

- We introduce a finite representation of infinite directed graph, called d-diagrams, which can be used to model infinite distributed computations or a subset of behaviors of a distributed programs.
- We show that it is possible to perform slicing on d-diagrams for both temporal and non-temporal regular predicates. In particular, we develop techniques to compute the slice of a predicate with respect to a class of predicates called RCTL. This class is powerful enough to represent properties such as violation of liveness.

2 Related Work

A lot of work has been done in identifying the classes of predicates which can be efficiently detected. Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [Gar02, HMRS98], *disjunctive* [Gar02], *observer-independent* [CBDGF95, Gar02], *linear* [Gar02, SG02a], *non-temporal regular* [GM01, MG01] predicates and *temporal regular* [SG03, SG02b]. As mentioned earlier, the previous work in this area is mainly restricted to finite traces. Computational slicing has also been applied to only finite traces. In this paper, we address the problem of predicate detection and slicing on infinite traces.

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) [ITS96] and petri nets [Pet62]. 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 that 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 an infinite poset such as the one shown in Figure 1 is not possible to represent through MSCs. Furthermore, the problem of model checking MSCs [AY99, BAL97] is in general intractable. In this paper, we consider a weaker model but we give a provably efficient algorithm for predicate detection and slicing.

Petri nets [Pet62] are also used to model concurrent systems. Partial order semantics in petri nets are captured through net unfoldings [MNW80]. Unfortunately, unfoldings are usually infinite sets and cannot be stored directly. Instead, a finite initial part of the unfolding, called the finite complete prefix [McM93] is generally used to represent the unfolding. McMillan showed that reachability can be checked using the finite prefix itself. Later Esparza [Esp94] extended this work to use unfoldings to efficiently detect predicates from a logic involving the possibility operator. Again petri nets can used to model the behavior of a complete system whereas d-diagrams cannot model “choice”. However, along with being simpler, d-diagrams have some other advantages over petri nets. They allow some *unsafe* petri nets to be model-checked which was not possible to do efficiently using earlier techniques for petri nets. It is also possible to detect (and compute slice for) $\mathbf{EG}(p)$ with our methods. This facilitates checking violation of progress properties; violation of progress properties could not be expressed using Esparza’s logic.

3 Infinite Posets and Infinite Directed Graphs

In this section, we first introduce a finite representation for a class of (countably) infinite posets which is a special case of an infinite directed graph. The representation for infinite posets is called *p-diagram* (poset diagram).

Definition 1 (p-diagram) *We define a p-diagram (poset diagram) $Q = (V,F,R,B)$ 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*
- *B (shift edges) is a subset of $R \times R$*

with the following constraints.

- *(P0) F is acyclic, i.e., (V,F) is a directed acyclic graph*
- *(P1) If u is a recurrent vertex and $(u,v) \in F$ or $(u,v) \in B$, then v is also recurrent.*

Thus, a p-diagram is a finite directed acyclic graph (V,F) together with a set of *shift-edges* given by B and a subset R denoting the vertices that appear an infinite number of times.

Each p-diagram represents a possibly infinite poset which is defined as follows.

Definition 2 (poset for a p-diagram) *The poset $(X,\leq)$ for a p-diagram Q is defined as follows:*

- *$X = \{u^1 | u \in V\} \cup \{u^i | i \geq 2 \wedge u \in R\}$*
- *The relation $\leq$ is the smallest reflexive transitive relation that satisfies: (1) if $(u,v) \in F$ and $u \in R$, then $\forall i : u^i \leq v^i$, and (2) if $(u,v) \in F$ and $u \notin R$, then $u^1 \leq v^1$, and (3) if $(v,u) \in B$, then $\forall i : v^i \leq u^{i+1}$.*

In other words, X consists of two types of elements. First, for all vertices $u \in V$ we have elements u^1 in X. Further, for all vertices $u \in R$, we add an infinite number of elements $\{\ u^i | i \geq 2\}$. It is clear that when R is empty we get a finite poset and when R is nonempty we get an infinite poset. For an element $e^i \in X$, we refer to i as the *index* or *iteration* of the element and we call e^i an *instance* of e.

We now have the following theorem.

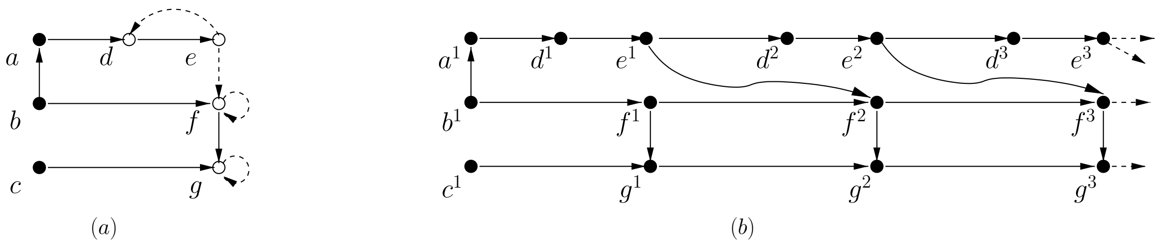

Figure 2: (a) A p-diagram and (b) its corresponding infinite poset

Theorem 1 $(X, \leq)$ *as defined above is a reflexive poset.*

Proof: We show that $\leq$ is reflexive, transitive and antisymmetric. It is reflexive and transitive by construction. We now show that $\leq$ as defined above does not create any nontrivial cycle (cycle of size greater than one). Any cycle in elements with the same index, say e^i and f^i, would imply cycle in (V,F). Thus, any cycle would have elements with at least two indices. However, $\leq$ never makes a higher indexed element smaller than a lower indexed element. ∎

Let us now consider some examples.

1. The set of natural numbers with the usual order can be modeled using the following p-diagram:
   $V = \{u\}$
   $F = \{\}$
   $B = \{(u,u)\}$
   $R = \{u\}$

   Note that this poset has infinite height and width equal to 1.

2. The set of natural numbers with no order can be modeled as:
   $V = \{u\}$
   $F = \{\}$
   $B = \{\}$
   $R = \{u\}$

3. Figure 2 shows an example of a p-diagram along with the corresponding infinite poset. The recurrent vertices in the p-diagram are represented by hollow circles and the non-recurrent vertices through filled circles. The forward edges are represented by solid arrows and the shift-edges by dashed arrows. We use the same convention throughout the paper for the figures.

We now give some examples of posets that cannot be represented using p-diagrams.

1. The set of all integers (including negative integers) under the usual order relation cannot be represented using p-diagram. The set of integers does not have any minimal element; any poset defined using p-diagram is well-founded.

2. Consider the set of all natural numbers with the order that all even numbers are less than all the odd numbers. This poset cannot be represented using p-diagram. In this poset, the upper covers [DP90] of an element may be infinite. For example, the number 2 is covered by infinite number of elements. In a p-diagram, an element can have only bounded number of lower (and upper covers).

3. Consider the poset of two-dimensional vectors with natural coordinates. Define $(x_1,y_1) \leq (x_2,y_2)$ iff $(x_1 \leq x_2) \wedge (y_1 \leq y_2)$. This poset can be visualized as the grid. It can be shown that this poset cannot be represented using a p-diagram.

The following properties of the posets generated by p-diagrams are easy to show.

Lemma 2 *A poset P defined by a p-diagram has the following properties.*

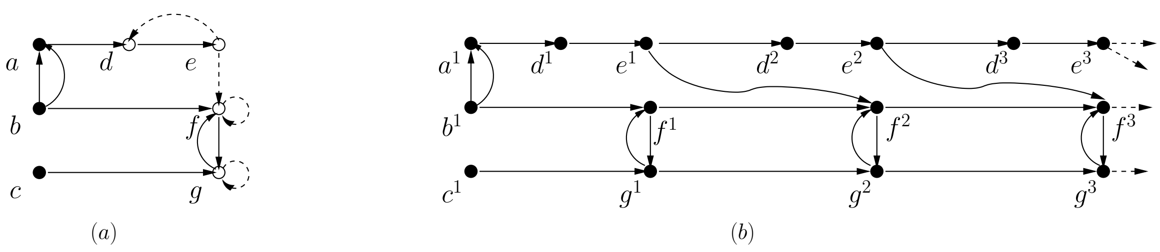

Figure 3: (a) A d-diagram and (b) its corresponding infinite directed graph

1. *P is well-founded.*
2. *There exists a constant k such that every element has the size of the set of its upper covers and lower covers bounded by k.*
3. *Every principal order ideal of P is finite.*

From distributed computing perspective, our intention is to provide a model for an infinite computation of a distributed system which is eventually periodic. Although a distributed computation in the happened-before model [Lam78] is always represented using a poset, it is useful to have a model of infinite directed graphs for the purpose of slicing [GM01, MG01]. The directed graph does not capture the order between events in the computation but it can be used to capture the set of possible consistent cuts or global states of the system. To this end we introduce a generalization of p-diagram called *d-diagram* (directed graph diagram). A d-diagram is simply a p-diagram without the constraint (P0) of F being acyclic.

Definition 3 (d-diagram) *We define a d-diagram $Q = (V, F, R, B)$ 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*
- *B (shift edges) is a subset of $R \times R$*

with the following constraint.

- *(D0) If u is a recurrent vertex and $(u,v) \in F$ or $(u,v) \in B$, then v is also recurrent.*

The directed graph generated by a d-diagram is also defined in a manner similar to the poset generated by a p-diagram.

Definition 4 (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.

Figure 3 shows a d-diagram along with a part of the infinite directed graph generated by it. In this case, the vertices $\{a,b\}$ and vertices $\{f,g\}$ form cycles in F.

Again, some properties of the directed graphs generated by d-diagrams are easy to show.

Lemma 3 *A directed graph G defined by a d-diagram has the following properties.*

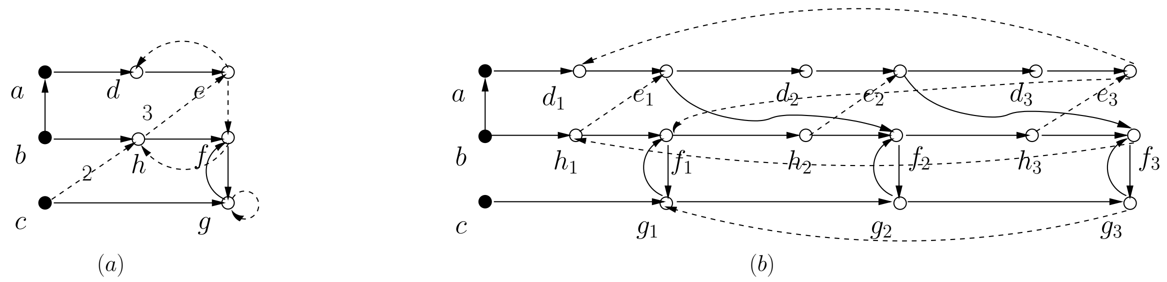

Figure 4: (a) A generalized d-diagram and (b) its equivalent d-diagram

1. *The cycles present in G only involve elements with the same index.*
2. *There exists a constant k such that every element has indegree and outdegree bounded by k.*
3. *There is no path from an element with iteration i to an element with iteration j where $j < i$.*
4. *If there is a path from element e^i to element f^j, then there is a path from e^{i+k} to f^{j+k} for all $k \geq 0$.*

Since p-diagram is a special case of d-diagram, we work with d-diagrams from now on. Moreover, a distributed computation generated by a finite number of processes has finite width and hence we are mainly interested in d-diagrams which generate finite width directed graphs. The following lemma characterizes those d-diagrams for which the corresponding directed graphs have finite width. The width of directed graph is defined as the maximum over the sizes of sets of pairwise disconnected vertices. Two vertices are said to be disconnected if there is no path from one to other.

Lemma 4 *A directed graph G defined by a 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.*

One can imagine generalizing 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 5 (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 $k-shift-edge$ is present in the d-diagram and $B_k \subseteq (V \times R)$ is the relation of $k-shift-edge$ with the property:*
If $(e,f) \in B_k$, then $e^i \rightarrow f^{i+k}$.

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 $\langle E, \rightarrow \rangle$ be the infinite graph generated by it. Then there exists a d-diagram (V',R',F',B') such that the graph $\langle E', \rightsquigarrow \rangle$ generated by the d-diagram is isomorphic to the graph $\langle E, \rightarrow \rangle$.*

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

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

2. $\mu(e^i) = e_k^l$ 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 4 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 1-shift-edges. We have essentially created multiple copies of the recurrent vertices and redrawn the k-shift-edges in terms of simple shift-edges.

4 D-diagram for distributed computing

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. Let $proc(e)$ denote the process on which event e occurs. The predecessor and successor events of e on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$.

Generally a distributed computation is modeled as a partial order of a set of events, called the *happened-before relation* [Lam78]. In this paper, we instead use directed graphs to model distributed computations as done in [MG01]. 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* or a global state as a set of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. Formally, C is a consistent cut of G, if

$$\forall x, y \in E : (x \rightarrow y) \wedge (y \in C) \Rightarrow (x \in C)$$

. We say that a strongly connected component is *non-trivial* if it has more than one element. We call the empty set $\emptyset$ and the set of vertices E as *trivial* consistent cuts. A distributed computation in our model can contain cycles. This is because whereas a computation in the happened-before model captures the *observable* order of execution of events, a computation in our model captures the set of possible consistent cuts. Intuitively, each strongly connected component of a computation can be viewed as a *meta-event*; a meta-event consists of one or more events.

Let $\mathcal{C}(G)$ denote the set of *finite* consistent cuts of the directed graph G. The following theorem is a slight generalization of the result in lattice theory that the set of *ideals* of a partially ordered set forms a finitary distributive lattice [Sta86]. A lattice is finitary if it has the bottom element and is locally finite.

Theorem 6 *Given a directed graph G, $(\mathcal{C}(G); \subseteq)$ forms a a finitary distributive lattice.*

In this work we focus only on *finite order ideals* or finite consistent cuts as they are the ones of interest for distributed computing. For an event $x \in C$, we denote by $J(x)$, the least consistent cut which includes x. The cut $J(x)$ can also be interpreted as principal ideal of the element x in the directed graph G.

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,

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

A consistent cut is uniquely characterized by its frontier and vice versa. Therefore, sometimes we specify a consistent cut by simply listing the events in its frontier instead of enumerating all its events. Two events are said to be *consistent* 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 $succ(e)$, if it exists, to f and from $succ(f)$, if it exists, to e. Note that in our model, an event can be inconsistent with itself.

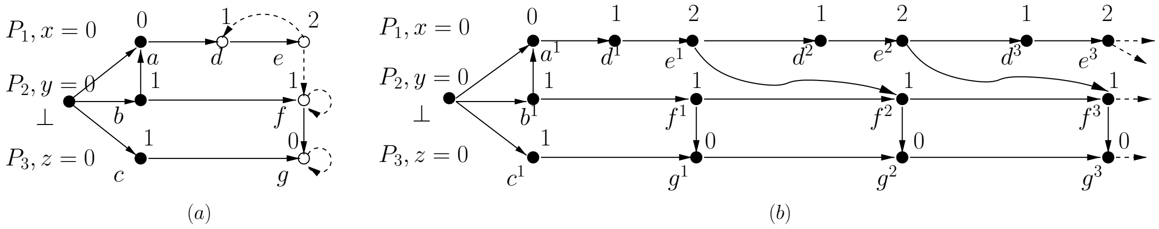

Figure 5: (a) A d-diagram with process and label information (b) The computation corresponding to the d-diagram v

We use a d-diagram (V,R,F,B) to represent the infinite directed graph of the computation. 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 $proc(e)$. For any two elements x,y on a process, either there is a path from x to y or from y to x. 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* denoted by $\perp$. The global initial event occurs before any other event on the processes and initializes the state of the processes. We do not have a notion of a global final event as we are only interested in the finite ideals of the computation.

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. We define $\mathcal{L} : G \to L$ 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 as repetition of the same state. We sometimes refer to this assumption as the *index-independence assumption*.

Figure 5 shows a computation represented as a d-diagram. Figure 5(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.

We now discuss another general 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 6 (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 $\langle E, \to \rangle$ that $\forall e^i \in frontier(C) : e \in R$. Then $\mathcal{U}(Q,C) = (V',R',F',B')$ is the d-diagram:*

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

It can be easily shown that the unrolled d-diagram generates a directed graph isomorphic to the original d-diagram. Let $\langle E, \to \rangle$ be the directed graph generated by the d-diagram Q and $\langle E', \rightsquigarrow \rangle$ be the directed graph generated by $\mathcal{U}(Q,C)$. Then we denote the isomorphism function from elements in E to elements in E' by 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 $\mathcal{L}(e^i) = \mathcal{L}(I_C(e^i))$. The following result is easy to show:

Lemma 7 *Let $\langle E', \rightsquigarrow\rangle$ be the graph generated by $\mathscr{U}(Q,C) = (V',R',F',B')$. Let e^i be such that $e^i \in C$ and $\forall j > i : e^j \notin C$. Then $I_C(e^{i+k}) = h^k$ for some $h \in R'$.*

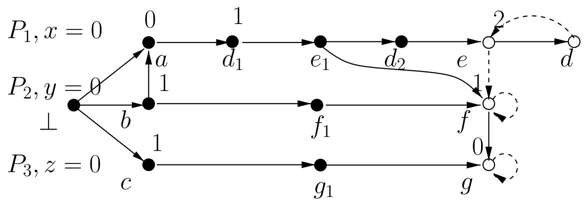

Figure 6: The d-diagram in Figure 5(a) unrolled with respect to $C = \{d^2, f^1, g^1\}$

As an example, Figure 6 shows the unrolling of the d-diagram in Figure 5(a) with respect to a cut $C = \{d^2, f^1, g^1\}$.

5 Background on Slicing and Regular Predicates

In the earlier work [GM01, MG01], the notion of slicing was used for finite directed graphs and was based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [DP90]. 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 [DP90]. 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 [DP90], we have the following theorem for the case of finite order ideals of a poset:

Theorem 8 *[[Sta86], 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 7 (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 [MG01]. 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 [GM01]. Equivalently,

Definition 8 (regular predicate [GM01]) *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)

We say that a regular predicate is *non-temporal* if it does not contain temporal operators such as **EF**, **AG**, and **EG**, otherwise it is a *temporal* regular predicate.

Some examples of non-temporal 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 [GM01].

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.

6 RCTL syntax and semantics

We work with the RCTL [SG03] (Regular CTL) extended for infinite computations. We define *successor* of a cut by a relation $\triangleright \subseteq \mathcal{C}(G) \times \mathcal{C}(G)$ such that $C \triangleright D$ if and only if $D = C \cup X$, where X is the set of vertices in some strongly connected component in $\langle E, \rightarrow \rangle$ and $X \cap C = \emptyset$. We denote the reflexive closure of this relation by $\trianglerighteq$. A *consistent cut sequence* π is a possibly infinite sequence of consistent cuts $C_0, C_1, \ldots,$ of $(\mathcal{C}(G), \subseteq)$ such that for each $0 \leq i$, $C_i \triangleright C_{i+1}$. The suffix of the consistent cut sequence from the cut C_i onwards is denoted by π^i and the i^{th} element in the sequence is denoted by $\pi(i)$. We say that a cut D is *reachable* from a cut C if $C \subseteq D$.

Propositional temporal logics use a finite set of atomic propositions AP, each one of which represents some property of the global state. A labeling function $\lambda\colon \mathcal{C}(G) \rightarrow 2^{AP}$ assigns to each global state the set of predicates from AP that hold in it. In this paper we assume that atomic propositions are non-temporal regular predicates defined only in terms of the labels of the events.

The formal syntax of RCTL is given below.

- Every predicate $ap \in AP$ is an RCTL formula.
- If p and q are RCTL formulas, then so are $p \wedge q$, $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$.

Given a distributive lattice $W = (\mathcal{C}(G), \subseteq)$, the formulas of RCTL are interpreted over the consistent cuts in $\mathcal{C}(G)$. Let p be an RCTL formula and C be a consistent cut in $\mathcal{C}(G)$. Then, the satisfaction relation, $L, C \models p$ means that predicate p holds at consistent cut C in lattice $W = (\mathcal{C}(G), \subseteq)$ and is defined inductively below. We denote $C \models p$ as a short form for $W, C \models p$, when W is clear from the context.

- $C \models ap$ iff $ap \in \lambda(C)$ for an atomic proposition ap.
- $C \models p \wedge q$ iff $C \models p$ and $C \models q$.
- $C \models \mathbf{EG}(p)$ iff for some consistent cut sequence π such that $\pi(0) = C$, we have $\forall i \geq 0 : \pi(i) \models p$.
- $C \models \mathbf{AG}(p)$ iff for all consistent cut sequences π such that $\pi(0) = C$ we have $\forall i \geq 0 : \pi(i) \models p$.
- $C \models \mathbf{EF}(p)$ iff for some consistent cut sequence π such that $\pi(0) = C$, we have $\exists i \geq 0 : \pi(i) \models p$.

We define $W \models p$ if and only if $W, \{\perp\} \models p$.

The formula $C \models \mathbf{AG}(p)$ (resp. $C \models \mathbf{EG}(p)$) intuitively means that for all consistent cut sequences (resp. for some consistent cut sequence) starting at C, p holds at every cut of the sequence. The formula $C \models \mathbf{EF}(p)$ intuitively means that for some consistent cut sequence starting at C, there exists a consistent cut that satisfies p.

The *predicate detection* problem is to decide whether the initial consistent cut of a distributed computation satisfies a predicate. Note that full CTL contains other operators as well but we are not going to deal with them. This subset of CTL defined over distributive lattices has the nice property of preserving regularity i.e. if we assume that the atomic propositions are regular predicates, then all the formulas in the logic are regular predicates. This follows from the fact that conjunction [GM01] and temporal operators preserve regularity [SG03]. In other words, if p and q are regular predicates, then $p \wedge q$, $\mathbf{EF}(p)$, $\mathbf{AG}(p)$ and $\mathbf{EG}(p)$ are also regular predicates.

To continue with our example of the computation in Figure 5, we are interested in computing the slice of the computation with respect to the predicate $\mathbf{EF}(x = 1 \wedge y = 1 \wedge z = 1)$.

7 Detecting non-temporal predicates

We first consider the problem of detecting non-temporal regular predicates in d-diagrams. We show that it is sufficient to perform predicate detection on a finite part of the infinite directed graph. This result is analogous to the concept of finite prefix [McM93] in the case of petri-nets.

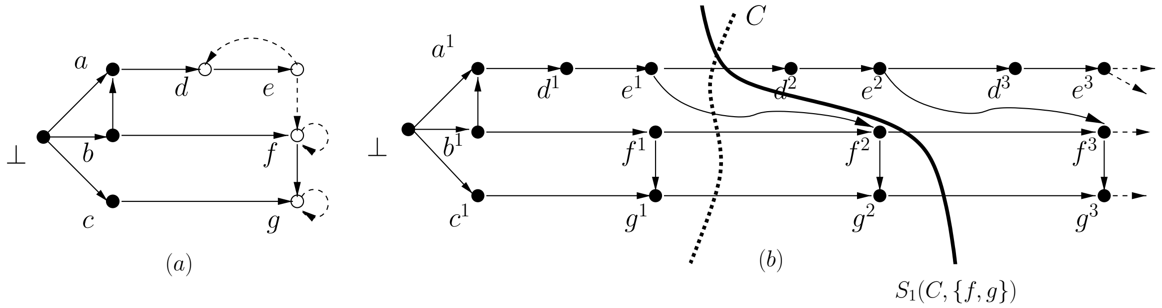

Figure 7: (a) A d-diagram (b) The computation for the d-diagram showing a cut and the shift of a cut

Before proceeding, we clarify some notation used in the paper. We always identify a cut by its frontier unless a distinction is made between the cut and its frontier. Every element $x \in G$ has the form e^i for some $e \in V$ and some $i \geq 1$. As a result, we use $e^i \in G$ as a shortcut for $x \in G : x = e^i$ with the quantification operators $\forall$ and $\exists$. In other words, $\forall e^i \in G$ is equivalent to $\forall x \in G$ and $\exists e^i \in G$ is equivalent to $\exists x \in G$.

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.

We first define the notion of a *shift* of a cut. Intuitively, the shift of a cut C produces a new cut by moving the cut C forward or backward by a certain number of iterations along a set of recurrent events in C. Formally,

Definition 9 (d-shift of a cut) *Given a cut C, a set of recurrent events $X \subseteq R$ and an integer d, a* d-shift cut *of C with respect to $X \subseteq R$, $S_d(C,X)$, is the cut*

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

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

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 2. 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 7 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.

We introduce some more concepts related to d-diagrams.

Definition 10 *For a consistent cut C and $l \geq 0$, define $C_l = \{e^i | e^i \in C \wedge i > l\}$. In other words, C_l is the part of the cut C which has indices greater than l.*

Definition 11 (indices of a cut) *For a consistent cut C, define $indices(C)$ as the sequence given by the set $\{0\} \cup \{i | e^i \in C\}$ sorted in increasing order. The j^{th} element of the sequence $indices(C)$ is referred to as $indices(C)[j]$.*

Definition 12 (gap) *For a consistent cut C and $i \in indices(C)$, define $gap(C,i) = j - i$ where $j \in indices(C), j > i$ and $\nexists l \in indices(C) : i < l < j$. If such a j does not exist, then $gap(C,i) = 0$.*

As an example, consider $C = \{e^3, f^1, g^1\}$ shown in Figure 8. Then $C_1 = \{e^3\}$, $indices(C) = \{0,1,3\}$ and $gap(C,0) = 1, gap(C,1) = 2$ and $gap(C,3) = 0$.

The regularity of a predicate allows one to explore the lattice of ideals in a fixed order of events such that if the predicate becomes true then the least ideal in which it becomes true will be explored. 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 here 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 core of the computation is simply the set of events in the computation with iteration less than or equal to N, the number of processes.

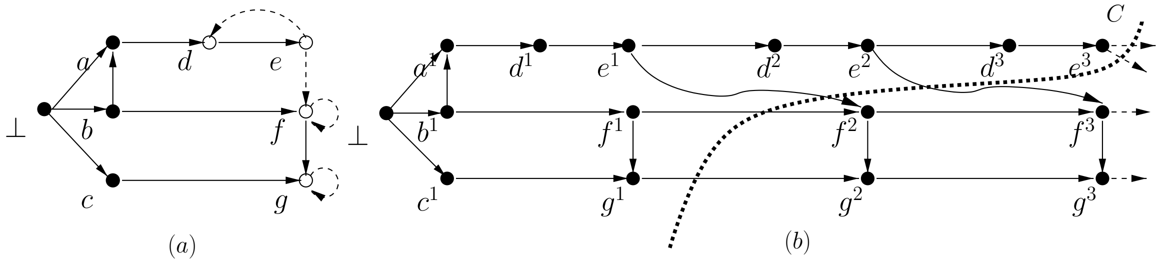

Figure 8: (a) A d-diagram and (b) its corresponding computation with a cut C

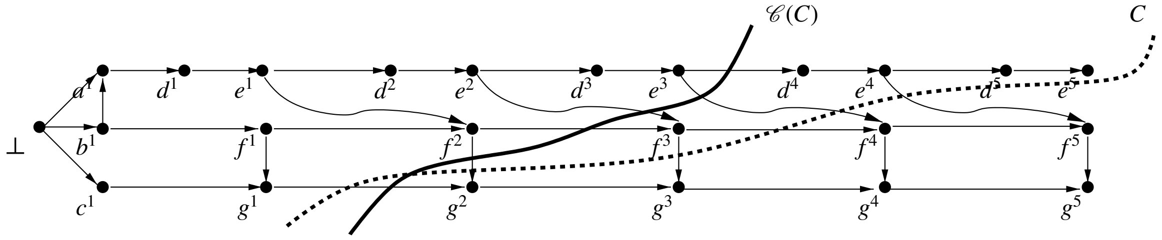

Figure 9: Compression operation being applied on a cut

Definition 13 (core of a computation) *For a d-diagram Q, we define $U(Q)$, the* core *of Q, as the directed graph given by the set of events $E' = \{e^j | e \in R \wedge 2 \leq j \leq N\} \cup \{e^1 | e \in V\}$ and the edges are the restriction of $\rightarrow$ to set E'.*

For the d-diagram in Figure 7(a), the part of the poset shown in Figure 7(b) is the core of the d-diagram.

For proving the completeness of the core, we define the notion of a *compression* operation. Intuitively, compressing a consistent cut removes all the gaps between iterations and results in a smaller consistent cut with the same labeling.

Definition 14 (compression) *Given a cut C and $i \in indices(C)$, define $\mathscr{C}(C,i,j) = S_{-j}(C, C_i)$. In other words, the operation $\mathscr{C}$ shifts all the events in C with iteration greater than i back by j iterations.*
We use $\mathscr{C}(C,i)$ to refer to $\mathscr{C}(C,i,gap(C,i)-1)$. In other words, $\mathscr{C}(C,i)$ compresses the gap at position i to 1.
We also define the cut obtained after all possible compressions as $\mathscr{C}(C)$. Let $m = |indices(C)|$. Then $\mathscr{C}(C) = \mathscr{C}(\mathscr{C}(\ldots\mathscr{C}(C, indices(C)[m]), indices(C)[m-1]), \ldots)$.

As an example, consider the cut $C = \{e^5, f^3, g^1\}$ shown in Figure 9. Then $\mathscr{C}(C,3,3) = \{e^2, f^3, g^1\}$, $\mathscr{C}(C,3) = \{e^4, f^3, g^1\}$ and $(C)(C) = \{e^3, f^2, g^1\}$.

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.

Lemma 9 *If C is a consistent cut, then $\mathscr{C}(C,l)$ is a consistent cut for any $l \in indices(C)$.*

Proof: Let $C' = \mathscr{C}(C,l)$ for some $l \in indices(C)$. 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.

Now wlog 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. ∎

Repeated application of the above lemma to compress all the gaps leads to the following result.

Corollary 10 *If G is a consistent cut, then $\mathscr{C}(G)$ is also a consistent cut.*

Now we can use the compression operation to compress any consistent cut to a 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 11 *If there is a cut $C \in \langle E, \rightarrow \rangle$, then there exists a cut $C' \in U(Q)$ such that $\mathcal{L}(C) = \mathcal{L}(C')$.*

Proof: Let $C' = \mathscr{C}(C)$. By Corollary 10, C' is a consistent cut and $\mathcal{L}(C) = \mathcal{L}(C')$. Moreover, $\forall i \in indices(C'), i \leq |P|$ as for any index $j \in indices(C')$, $gap(G', i) = 1$. Therefore, $C' \in U(Q)$. ■

The above result shows that if a predicate is not true in $U(Q)$, then it is not true anywhere in the d-diagram. Hence, to check if a predicate ever becomes true, it suffices to check the truthness of the predicate in the core. Now the algorithms for predicate detection on finite directed graphs can be used for d-diagrams as well. Note that here we did not use the regularity of the predicate and hence, this result holds for any predicate which is based on the label of the consistent cut.

8 Slicing with non-temporal regular predicates

In this section, we consider the problem of slicing a d-diagram with respect to a non-temporal regular predicate. It was shown [GM01] that for computing the slice of a finite computation with respect to a regular predicate p, it is sufficient to compute $J_p(x)$ for every element x in the computation. The result can be easily extended for the case of infinite computation when we are dealing with finite consistent cuts. However, the problem is that since the computation is infinite, it is not feasible to compute $J_p(x)$ for every element. Instead, we show that we can directly operate on the d-diagram and compute the slice of the infinite computation as another d-diagram. In this section, the examples work on the d-diagram in Figure 5 and assume that the predicate p is $(x = 1) \wedge (y = 1) \wedge (z = 1)$.

We first introduce the concept of *shift-diameter* of a d-diagram.

Definition 15 (shift-diameter of 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. By the shortest path here we mean the path with minimum number of shift-edges. When Q is clear from the context, we simply use η to denote $\eta(Q)$.*

For the d-diagram in Figure 7, $\eta = 1$. We prove a bound on the shift-diameter in terms of the number of processes N in the system.

Lemma 12 *For a d-diagram, $\eta(Q) \leq 2N$.*

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, process shift-edges are traversed at most once in the path and each process is visited at most once in the path. Moving from one process to another requires at most one shift-edge. Hence, $\eta(Q) \leq 2N$. ■

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 13 *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 $(e^i, f^j) \in \mathcal{P}(\langle E, \rightarrow \rangle)$ and $\forall k > i : (e^k, f^j) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$. 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 14 *For a node $e \in R$, there exists j such that $J(e^{j+1}) = S_1(J(e^j))$.*

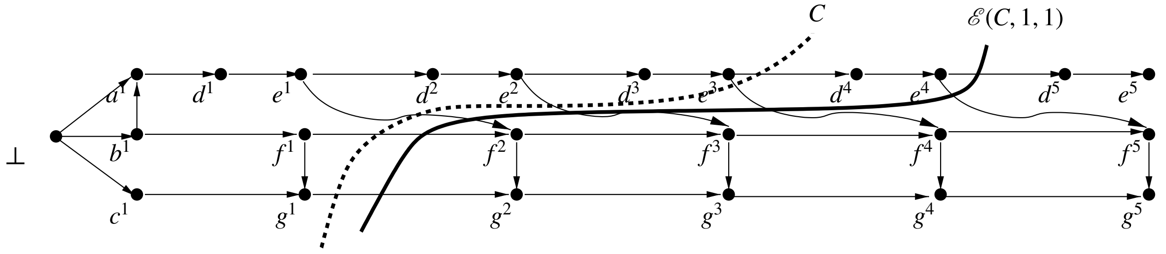

Figure 10: Expansion operation being applied on a cut

Proof: For proving the theorem, we show a stronger result. Let $\beta = \eta + 1$. We show that for a recurrent node e, $J(e^{\beta+1}) = S_1(J(e^\beta))$.

Consider $f^j \in frontier(S_1(J(e^\beta)))$. If $f \in V \setminus R$, then $f^j \in frontier(J(e^\beta))$ and hence (f^j, e^β) and so $(f^j, e^{\beta+1})$ are in $\mathcal{P}(\langle E, \rightarrow \rangle)$. If $f \in R$, then f^{j-1} exists as $j > 1$ by Lemma 13. Moreover, $f^{j-1} \in frontier(J(e^\beta))$ and hence $(f^{j-1}, e^\beta) \in \mathcal{P}(\langle E, \rightarrow \rangle)$. By the property of d-diagram, this implies that $(f^j, e^{\beta+1}) \in \mathcal{P}(\langle E, \rightarrow \rangle)$. Therefore, $S_1(J(e^\beta)) \subseteq J(e^{\beta+1})$.

Now we show that $J(e^{\beta+1}) \subseteq S_1(J(e^\beta))$. In this case, it is equivalent to showing $S_{-1}(J(e^{\beta+1}) \subseteq J(e^\beta)$. Consider $f^j \in frontier(S_{-1}(J(e^{\beta+1})))$. If $f \in R$, then an argument similar to the previous case suffices. If $f \in V \setminus R$, then $j = 1$ and $f^j \in J(e^{\beta+1})$. Since $\beta > \eta$, therefore $f^j \in J(e^\beta)$; otherwise the shortest path from f to e has more than η shift-edges.

Hence $J(e^{j+1}) = S_1(J(e^j))$. ■

In case of a p-diagram, this result can be used to assign timestamps to vertices in the p-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. We refer to the timestamps assigned to nodes in the p-diagram as p-timestamps. This result gives us an algorithm for assigning p-timestamp to a recurrent event. The p-timestamp for a recurrent event e, $PV(e)$ is a vector of the form $(V(e^1), \ldots, V(e^\alpha); I(e))$ where $I(e) = V(e^{\alpha+1}) - V(e^\alpha)$ and $V(e^j)$ is the timestamp assigned by the normal vector clock algorithm to event e^j. Now for any event $e^j, j > \alpha$, $V(e^j) = V(e^\alpha) + (j - \alpha + 1) * I(e)$.

This algorithm requires $O(\eta N)$ space for every recurrent vertex. Once the p-timestamps have been assigned to the vertices, any two instances of recurrent vertices can be compared using the following property of the vector clocks.

Lemma 15 *Given two events e^i and f^j, $V(e^i) < V(f^j)$ iff $e^i < f^j$.*

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

Definition 16 (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 17 (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 $i \leq \rho$ and $j > \rho$, $(e^i, f^k) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$ for any $k \geq j$.*

As an example, let $C = \{e^3, f^1, g^1\}$ as shown in Figure 7. 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 16 *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 $l \geq 0$.*

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 18 (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 7, $H(f^2) = \{P_1, P_2\}$. Some properties of helper processes are easy to show.

Lemma 17 *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 18 *For $i > N - |H(e)| + \eta$, $frontier(J_p(e^i))$ can be written as $C \cup D$, where*

1. *$f^j \in C \Rightarrow (proc(f) \notin H(e)) \wedge (j \leq N - |H(e)|)$*
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 $C = \mathcal{C}(C')$ 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 17, 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 11 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 18 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 18 to $i > N + \eta$. This makes the bound independent of e and allows us to deal with all vertices uniformly.

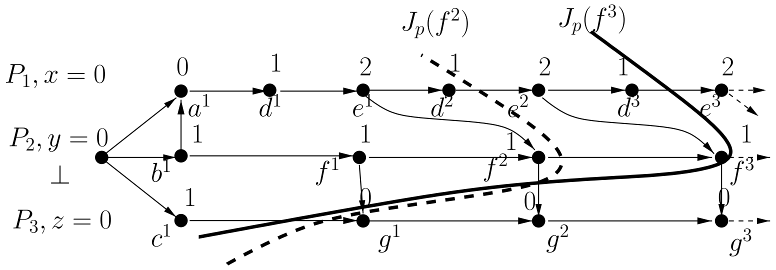

Figure 11: The cuts $J_p(f^2)$ and $J_p(f^3)$ where p is $(x = 1) \wedge (y = 1) \wedge (z = 1)$

Corollary 19 *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 20 *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 16 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 11, 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 19 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 21 *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 18, $J_p(e^i)$ 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 20, $C_1 = C_3$ 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, $J_p(e^{i+1}) \subseteq J_p(f^{j+1})$. ■

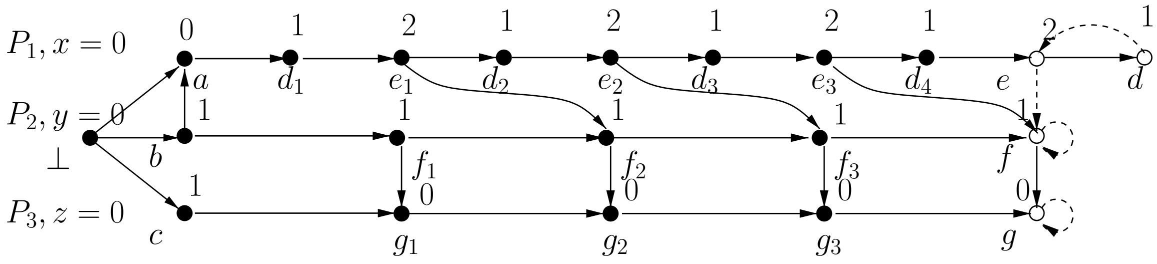

Figure 12: The unrolled d-diagram $\mathscr{U}(Q, \mathcal{R}(p))$

A slice of a computation can have multiple representations. The *skeletal representation* [MG01] 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 19 (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 22 *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 $I_{\mathcal{R}(p)}(h^l) = a^1$ and $I_{\mathcal{R}(p)}(g^k) = b^1$ 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 $I_{\mathcal{R}(p)}(h^l) = a^i$ and $I_{\mathcal{R}(p)}(g^k) = b^j$, 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 12 shows the unrolled d-diagram

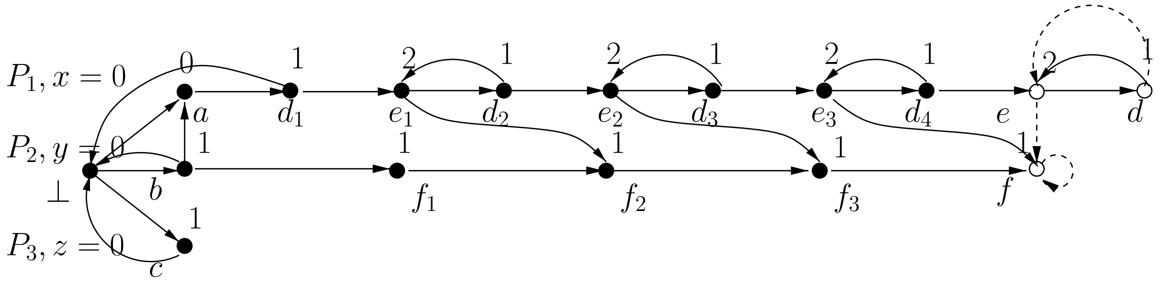

Figure 13: The slice of the d-diagram in Figure 5 with respect to $(x = 1) \wedge (y = 1) \wedge (z = 1)$

$\mathcal{U}(Q, \mathcal{R}(p))$. Note that the process shift-edge on the process P_1 is now going from d to e instead of from e to d. 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 are now in a position to construct the slice of the original d-diagram Q using the unrolled d-diagram $\mathcal{U}(Q, \mathcal{R}(p))$ as both of them generate the same computation. We slice the computation $\mathcal{U}(Q, \mathcal{R}(p))$ 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 $\mathcal{U}(Q, \mathcal{R}(p))$ 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 $\mathcal{U}(Q, \mathcal{R}(p))$ 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 $\mathcal{U}(Q, \mathcal{R}(p))$. 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 $\mathcal{U}(Q, \mathcal{R}(p))$.

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

Proof: Follows from the construction and Lemma 21. ■

Figure 13 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 non-temporal predicate only adds edges to the slice.

9 Slicing with respect to temporal predicates

Now we turn to the problem of slicing a computation with respect to temporal predicates. The algorithms presented here are generalizations of the work in [SG03]. We assume that we are given the original computation $\langle E, \rightarrow \rangle$ as d-diagram $Q = (V, R, F, B)$ and $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ as d-diagram $Q' = (V', R', F', B')$ and we are required to compute a d-diagram Q'' to represent the slice with respect to $\mathbf{EF}(p)$, $\mathbf{AG}(p)$ and $\mathbf{EG}(p)$. We further assume that $\langle E', \rightsquigarrow \rangle$ is the computation generated by the d-diagram Q'. Let the one-to-one relation M from the set of vertices E' to the set of vertices E denote the correspondance between the elements in E and E'. In the following, we use $x \in E'$ interchangeably with $M(x)$ unless otherwise stated. For a cut $C \in \mathcal{C}(\langle E, \rightarrow \rangle)$ with $\forall x \in frontier(C) \in \mathcal{C}(\mathsf{slice}(\langle E, \rightarrow \rangle,))$ Then the algorithms for computing slice for $\mathbf{EF}(p)$, $\mathbf{AG}(p)$ and $\mathbf{EG}(p)$ are given in the Figures 14, 16 and 17 respectively.

Algorithm A1

Input: A computation $\langle E, \rightarrow \rangle$ as d-diagram Q and
slice$(\langle E, \rightarrow \rangle, p) = \langle E', \rightsquigarrow \rangle$ as d-diagram Q'
Output: slice$(\langle E, \rightarrow \rangle, \mathbf{EF}(p))$ as d-diagram Q''
1. Initialize Q'' to Q'
2. For each pair of vertices $e^1, f^1 \in E'$ such that,
   (i) $(M(e^1), M(f^1))$ is not an edge in $\langle E, \rightarrow \rangle$, and
   (ii) (e^1, f^1) is an edge in $\langle E', \rightsquigarrow \rangle$
   remove the edge (e, f) in Q''
3. **return** Q''

Figure 14: Algorithm for generating a slice with respect to $\mathbf{EF}(p)$.

First consider the algorithm A1 for computing the slice with respect to $\mathbf{EF}(p)$. The algorithm simply removes all the edges in slice$(\langle E, \rightarrow \rangle, p)$ which were not present in the original computation. The intuition behind this algorithm is very simple: For an element e^i, $J_p(e^i)$ exists iff $J(e^i)$ satisfies $\mathbf{EF}(p)$. If the element e^i exists in the slice, then $J_p(e^i)$ exists for that element. Note that the slice is not the same as the original computation as we do not restore the vertices which were removed during the process of slicing. This algorithm is equivalent to the algorithm in [SG03] for finite directed graphs if we remove elements for which $J_p(e^i)$ does not exist.

Theorem 24 *Algorithm A1 outputs* slice$(\langle E, \rightarrow \rangle, \mathbf{EF}(p))$.

Proof: Let $G = \langle E, \rightarrow \rangle$ and $G' = \langle E', \rightsquigarrow \rangle$ be the computation corresponding to the d-diagram output by Algorithm A1. We show that a consistent cut $D \in \mathcal{C}(G)$ satisfies $\mathbf{EF}(p)$ iff $D \in \mathcal{C}(G')$. Observe that a cut $D \in \mathcal{C}(G)$ exists in $\mathcal{C}(G')$ iff $\forall e^i \in frontier(D) : e^i \in G'$ as the order between elements is retained.

Assume that $\mathbf{EF}(p)$ holds at a consistent cut $D \in \mathcal{C}(G)$. Then,

$\{$Definition of $\mathbf{EF}(p)\}$
$D \models \mathbf{EF}(p)$ iff $\exists C : C \in \mathcal{C}(G) : D \subseteq C \wedge C \models p$
$\{$Definition of $J_p\}$
$\equiv$ $\forall e^i \in frontier(D) : J_p(e^i)$ exists.
$\{$Definition of slice$(\langle E, \rightarrow \rangle, p)$ $\}$
$\equiv$ $\forall e^i \in frontier(D) : e^i \in$ slice$(\langle E, \rightarrow \rangle, p)$,
$\{$Observation made earlier$\}$
$\equiv$ D is in the computation output by A1. ■

Continuing with our example from the previous section, the slice for $\mathbf{EF}((x = 1) \wedge (y = 1) \wedge (z = 1))$ is shown in Figure 15. We have essentially removed the additional edges that were introduced during slicing but have not restored the deleted vertex g.

Algorithm A2 in Figure 16 generates the slice for $\mathbf{AG}(p)$. The algorithm first checks if some recurrent vertex has been removed or if an edge has been added between recurrent vertices in slice$(\langle E, \rightarrow \rangle, p)$ which was not present in the original computation. If some vertex was removed or an additional edge was introduced then the slice of $\mathbf{AG}(p)$ is set to the empty slice. Intuitively, if a vertex e was removed, then for any consistent cut C, there is a consistent cut reachable from C which includes some instance e^i of e. If an edge (e, f) was introduced between two recurrent vertices e and f, then for any finite consistent cut C, there is a consistent cut reachable from C which includes f^i and not e^i (or includes f^i and not e^{i-1} in case a shift-edge was added) for a large enough i. Therefore no cut C satisfies $\mathbf{AG}(p)$ and so the slice$(\langle E, \rightarrow \rangle, \mathbf{AG}(p))$ is empty. If this is not the case, then we can apply the algorithm for the finite case directly on the d-diagram. In this case, for any additional edge (e, f),

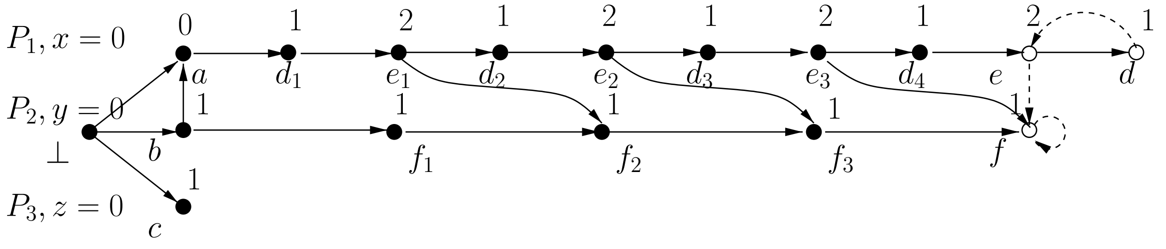

Figure 15: The slice of the d-diagram in Figure 5 with respect to $\mathbf{EF}((x=1)\wedge(y=1)\wedge(z=1))$

an edge is added from e to $\perp$. Intuitively, in this case the recurrent part is equivalent to having a final cut which satisfies the predicate and thus the algorithm for the finite case can be used.

Theorem 25 *Algorithm A2 outputs* $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$.

Proof: Let $G = \langle E, \rightarrow\rangle$ and $G' = \langle E', \rightsquigarrow\rangle$ be the computation corresponding to the d-diagram output by Algorithm A2. We show that a consistent cut $D \in \mathcal{C}(G)$ satisfies $\mathbf{AG}(p)$ iff $D \in C(G')$. First suppose that there exists a vertex $e \in R$ and a k such that for all $f \in R'$ and for all i, $M(f^i) \neq e^k$. As a result, for all $i \geq k$, there does not exist a vertex $x \in E'$ such that $M(x) = e^i$. In this case, we show that $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ is empty. Consider a cut $D \in \mathcal{C}(G)$. Since D is finite, there exists an instance e^l of e such that $e^l \notin D$ with $l \geq k$. Then consider the cut $D \cup J(e^l)$. This cut is reachable from D and does not satisfy p as $J_p(e^l)$ does not exist. Therefore, D does not satisfy $\mathbf{AG}(p)$ and hence, $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ is empty.

Now consider the case, when there exists a pair of vertices $e, f \in R'$ and i such that,
(i) $(M(e^1), M(f^i)) \ni n\mathcal{P}(\langle E, \rightarrow\rangle)$, and
(ii) $(e^1, f^i) \in \mathcal{P}(\langle E', \rightsquigarrow\rangle)$

Note that the above condition implies that $\forall j$
(i) $(M(e^{1+j}), M(f^{i+j})) \notin \mathcal{P}(\langle E, \rightarrow\rangle)$, and
(ii) $(e^{1+j}, f^{i+j}) \notin \mathcal{P}(\langle E', \rightsquigarrow\rangle)$

We use an argument similar to the first case. Consider a cut $D \in \mathcal{C}(G)$. Since D is finite, there exists an instance $M(f^l)$ with $l > i$ such that $M(f^l), M(e^{1+l-i}) \notin D$. Then consider the cut $D \cup J(M(f^l))$. This cut is reachable from D and does not satisfy p as it does not include $M(e^{1+l-i})$. Therefore, D does not satisfy $\mathbf{AG}(p)$ and hence, $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ is empty.

Now assume that neither of the conditions in line 2 of the algorithm for $\mathbf{AG}(p)$ are true. Then consider the cut $D = \cup_{e \in R'} J_p(e^1)$. It can be easily shown that every cut reachable from D satisfies p and hence $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ is not empty. Therefore, $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ is empty iff one of the condition in line 2 holds.

The rest of the proof of correctness of the algorithm follows from the proof of correctness of the algorithm for computing $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$ when $\langle E, \rightarrow\rangle$ is finite [SG03]. ■

For slicing with respect to $\mathbf{EG}(p)$, we introduce the notion of fairness.

Definition 20 (fair sequence of consistent cuts) *Let $C(i)$ denote the event $x \in frontier(C)$ such that $proc(x) = P_i$. We say that a process P_i is enabled in a consistent cut C if $succ(x)$ exists. Then, an infinite sequence of consistent cuts π is called fair if there is no π^j such that there exists a P_i such that $\forall C, D \in \pi^j$, $C(i) = D(i)$ and $succ(C(i))$ exists.*

Intuitively, a sequence of consistent cuts is said to be fair if it executes events from every process infinitely often. In terms of distributed systems, only fair paths are of interest to us as the unfair paths would not be observed in a system's behavior except in the case of failures.

Hence, we modify the interpretation of the operator **EG** as follows:
$C \models \mathbf{EG}(p)$ iff for some *fair* consistent cut sequence π such that $\pi(0) = C$, we have $\exists i \geq 0 : \pi(i) \models p$.

With fairness, the algorithm A3 for slicing with respect to $\mathbf{EG}(p)$ is similar to the one for $\mathbf{AG}(p)$. Here instead of simply checking for the introduction of an edge, we further check if the introduced edge forms a

Algorithm A2

Input: A computation $\langle E, \rightarrow \rangle$ as d-diagram Q and
slice$(\langle E, \rightarrow \rangle, p) = \langle E', \rightsquigarrow \rangle$ as d-diagram Q'

Output: slice$(\langle E, \rightarrow \rangle, \mathbf{AG}(p))$ as d-diagram Q''

1. Initialize Q'' to Q'
2. **If** there exists a vertex $e \in R$ and a j such that for all $f \in R'$ and for all i, $M(f^i) \neq e^j$
   or there exists a pair of vertices $e, f \in R'$ and i such that,
   (i) $(M(e^1), M(f^i)) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$, and
   (ii) $(e^1, f^i) \in \mathcal{P}(\langle E', \rightsquigarrow \rangle)$
   then
3. **return** empty slice
4. **else**
5. For each pair of vertices $e, f \in V'$ such that,
   (i) $(M(e^1), M(f^1)) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$, and
   (ii) $(e^1, f^1) \in \mathcal{P}(\langle E', \rightsquigarrow \rangle)$
   add an edge from vertex e to the vertex $\perp$ in Q''
6. **return** Q''

Figure 16: Algorithm for generating a slice with respect to $\mathbf{AG}(p)$.

strongly connected component (SCC). Intuitively, if there is a strongly connected component in the recurrent part, then for any finite consistent cut C, every fair consistent cut sequence starting from C would have to move across the strongly connected component. As a result, every fair path would include a consistent cut which does not satisfy the predicate p. The case when no strongly connected component was formed between recurrent vertices and no recurrent vertex was removed, we use a similar algorithm for the finite part.

The proof of correctness of the algorithm for $\mathbf{EG}(p)$ is similar to that of $\mathbf{AG}(p)$ and is omitted here.

10 Grafting two slices

In this section, we present algorithm to *graft* two slices with respect to the $\wedge$ operator. Formally, the problem of grafting with respect to $\wedge$ is defined as follows: Given d-diagrams corresponding to slice$(\langle E, \rightarrow \rangle, p)$ and slice$(\langle E, \rightarrow \rangle, q)$ where p and q are regular predicates, compute the d-diagram corresponding to slice$(\langle E, \rightarrow \rangle, p \wedge q)$.

Clearly, slice$(\langle E, \rightarrow \rangle, p \wedge q)$ contains a consistent cut of $\langle E, \rightarrow \rangle$ iff the cut satisfies p as well as q. Let $F_{min}(x,k)$ denote the earlier of events $F_p(x,k)$ and $F_q(x,k)$, that is, $F_{min}(x,k) = \min\{F_p(x,k), F_q(x,k)\}$. Let $\langle E', \rightsquigarrow \rangle$ be the directed graph similar to the skeletal representation for slice$(\langle E, \rightarrow \rangle, p \wedge q)$ except that (1) The set of elements $E' = (E_1 \cap E_2) \setminus D$ where E_1, E_2 are the sets of elements in slice$(\langle E, \rightarrow \rangle, p)$ and slice$(\langle E, \rightarrow \rangle, q)$ respectively and D is the set of elements reachable from an element in $E \setminus (E_1 \cap E_2)$ in either slice$(\langle E, \rightarrow \rangle, p)$ or slice$(\langle E, \rightarrow \rangle, q)$ and (2) $F_{min}(x,k)$ is used instead of $F_{p \wedge q}(x,k)$ for $x \in E'$. It was shown that the resulting directed graph is cut-equivalent to the skeletal representation [MG01] and hence is a valid representation of slice$(\langle E, \rightarrow \rangle, p \wedge q)$. Here again, we want to obtain the d-diagram representation of the graph $\langle E', \rightsquigarrow \rangle$ given the d-diagrams Q_p and Q_q for slice$(\langle E, \rightarrow \rangle, p)$ and slice$(\langle E, \rightarrow \rangle, q)$.

We briefly sketch the algorithm for obtaining the d-diagram Q corresponding to $\langle E', \rightsquigarrow \rangle$. Let i_1 be the iteration such that for any $j \geq i_1$, $J_p(e^j)$ becomes stable for any element e. Similarly, i_2 be the iteration such that for any $j \geq i_2$, $J_q(e^j)$ becomes stable for any element e. Let $k = \max\{i_1, i_2\}$. Unroll the d-diagrams Q_p and Q_q till they have events from iteration k. Since the cuts $J_p(e^j)$ and $J_q(e^j)$ are stable for $j \geq k$, $F_{min}(e^i, k)$ also stabilizes. As a result, the set of vertices to be deleted and the reset cut can be ascertained as in the case of non-temporal predicates and using a similar procedure, the d-diagram Q can be computed.

Algorithm A3

Input: A computation $\langle E, \rightarrow \rangle$ as d-diagram Q and slice$(\langle E, \rightarrow \rangle, p)$ as d-diagram Q'
Output: slice$(\langle E, \rightarrow \rangle, \mathbf{EG}(p))$ as d-diagram Q''
Input: A computation $\langle E, \rightarrow \rangle$ as d-diagram Q and slice$(\langle E, \rightarrow \rangle, p) = \langle E', \rightsquigarrow \rangle$ as d-diagram Q'
Output: slice$(\langle E, \rightarrow \rangle, \mathbf{AG}(p))$ as d-diagram Q''

1. Initialize Q'' to Q'
2. **If** there exists a vertex $e \in R$ such that for all $f \in R'$ and for all j, $M(f^1) \neq e^j$
   or there exists a pair of vertices $e, f \in R'$ and i such that,
   (i) $(M(e^1), M(f^i)) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$, and
   (ii) (e^1, f^i) is part of a SCC in $\langle E', \rightsquigarrow \rangle$
   then
3. **return** empty slice
4. **else**
5. For each pair of vertices $e, f \in V'$ such that,
   (i) $(M(e^1), M(f^1)) \notin \mathcal{P}(\langle E, \rightarrow \rangle)$, and
   (ii) (e^1, f^1) is part of a SCC in $\langle E', \rightsquigarrow \rangle$
   add an edge from vertex e to the vertex $\perp$ in Q''
6. **return** Q''

Figure 17: Algorithm for generating a slice with respect to $\mathbf{EG}(p)$.

11 Predicate Detection using slicing

In the previous sections we looked at the algorithms for slicing an infinite directed graph with respect to non-temporal predicates, temporal predicates and for grafting two slices. These algorithms take as input a d-diagram and return another d-diagram as an output. All the operators in our logic ($\wedge$, **EF**, **AG** and **EG**) preserve the regularity of the atomic propositions. To compute the slice of a d-diagram with respect to any arbitrary predicate p, we recursively process p from inside to outside while applying the boolean and temporal operators to compute the final slice.

Even if the predicate we are interested in detecting does not belong to RCTL, we can still make use of slicing in some cases. For example, to detect a predicate of the form $p \wedge q$ where p is a regular predicate and q is not a regular predicate, we can compute the slice of the d-diagram with respect to p and then detect q in this slice using other methods such as using SPIN [Hol97].

12 Conclusion and Future Work

In this paper, we presented a finite representation for a class of infinite directed graph. These infinite directed graphs can be used to model an infinite distributed computation (and not all execution behaviors of a distributed program). We showed that it is possible to *slice* these d-diagrams with respect to a predicate from the RCTL class of predicates. The RCTL class of predicates includes temporal operators and conjunction operators, allowing us to check for violation of properties such as liveness. Computing the slice gives us the additional benefit of using other techniques to detect predicates outside the RCTL class of predicates on smaller computations.

As future work, we aim to extend the techniques developed here to detect a broader class of predicates than RCTL. This includes relaxing the assumption of index-independence to include some classes of channel predicates and extending RCTL to include other operators such as disjunction and negation. Another direction of future work would be develop techniques for representing a distributed program as a collection of d-diagrams and model check the program for properties by checking for properties on individual d-diagrams.

References

[AY99] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. of the 10th Intlernational Conference on Concurrency Theory*, pages 114–129. Springer Verlag, 1999.

[BAL97] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Third International Workshop, TACAS 97*, number 1217 in Lecture Notes in Computer Science, 1997.

[CBDGF95] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan 1995.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. of the Workshop on Logics of Programs*, volume 131 of *LNCS*, Yorktown Heights, New York, May 1981.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[EC99] D. Peled E.M. Clarke, O. Grumberg. *Model Checking*. The MIT Press, 1999.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994.

[Gar02] V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

[GM01] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proc. of the* 15^{th} *Int'l. Conference on Distributed Computing Systems (ICDCS)*, 2001.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proc. of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[HMRS98] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, 1998.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[ITS96] *Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC)*, 1996.

[KG05] S. Kashyap and V. K. Garg. Exploiting predicate structure for efficient reachability detection. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

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

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MG01] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *In Proc. of the* 15^{th} *Int'l. Symposium on Distributed Computing (DISC)*, 2001.

[MNW80] G. Plotkin M. Nielsen and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1980.

[N. 01] N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proc. of the* 15^{th} *Int'l. Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2001.

[Pel93] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *5th Int'l. Conference on Computer-Aided Verification (CAV)*, pages 409–423. Springer, Berlin, Heidelberg, 1993.

[Pet62] Carl Adam Petri. *Kommunikation mit Auto-maten*. PhD thesis, Bonn: Institut fuer Instru- mentelle Mathematik, 1962.

[SG02a] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[SG02b] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in the happened before model. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Florida, April 2002.

[SG03] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *7th International Conference on Principles of Distributed Systems*, La Martinique, France, December 2003.

[Sta86] R. Stanley. *Enumerative Combinatorics*. Wadsworth and Brookes/Cole, 1986.

[Val90] A. Valmari. A Stubborn Attack On State Explosion. In *2nd Int'l. Conference on Computer-Aided Verification (CAV)*, volume 531 of *LNCS*, pages 156–165, 1990.