# Modeling and Analyzing Periodic Distributed Computations

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

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

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

## 1 Introduction

Correctness properties of distributed programs can be classified either as safety properties or liveness properties. Informally, a safety property states that the program never enters a bad (or an unsafe) state, and a liveness property states that the program eventually enters into a good state. For example, in the classical dining philosopher problem a safety property is that "two neighboring philosophers never eat concurrently" and a liveness property is that "every hungry philosopher eventually eats." Assume that a programmer is interested in monitoring for violation of a correctness property in her distributed program. It is clear how a runtime monitoring system would check for violation of a safety property. If it detects that there exists a consistent global state[1] in which two neighboring philosophers are eating then the safety property is violated. The literature in the area of global predicate detection deals with the complexity and algorithms for such tasks [2, 3]. However, the problem of detecting violation of the liveness

* currently at Google

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

*** currently at Microsoft

property is harder. At first it appears that detecting violation of a liveness property may even be impossible. After all, a liveness property requires something to be true eventually and therefore no finite observation can detect the violation. We show in this paper a technique that can be used to infer violation of a liveness property in spite of finite observations. Such a technique would be a basic requirement for detecting a temporal logic formula[4] on a computation for runtime verification.

There are three important components in our technique. First, we use the notion of a *recurrent* global state. Informally, a global state is recurrent in a computation $\gamma$ if it occurs more than once in it. Existence of a recurrent global state implies that there exists an infinite computation $\delta$ in which the set of events between two occurrences of can be repeated ad infinitum. Note that $\gamma$ may not even be a prefix of $\delta$. The actual behavior of the program may not follow the execution of $\delta$ due to nondeterminism. However, we know that $\delta$ is a legal behavior of the program and therefore violation of the liveness property in $\delta$ shows a bug in the program.

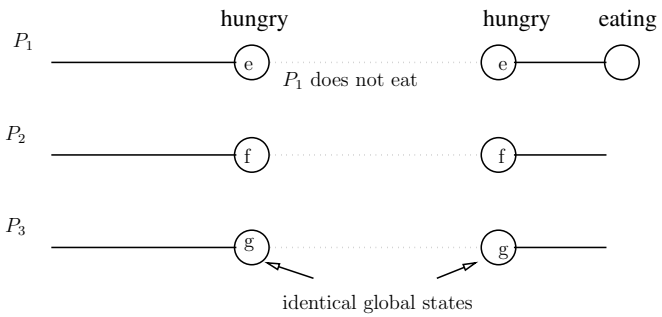


**Fig. 1.** A finite distributed computation $C$ of dining philosophers

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

The second component of our technique is to develop a finite representation of the infinite behavior $\gamma$. Mathematically, we need a finite representation of the infinite but periodic poset $\gamma$. In this paper, we propose the notion of d-diagram to capture infinite periodic posets. Just as finite directed acyclic graphs (dag's) have been used to represent and analyze finite computations, d-diagrams may be used for representing periodic infinite distributed computations for monitoring or logging purposes. The logging may be useful for replay or offline analysis of the computation. Figure 2 shows a d-diagram and

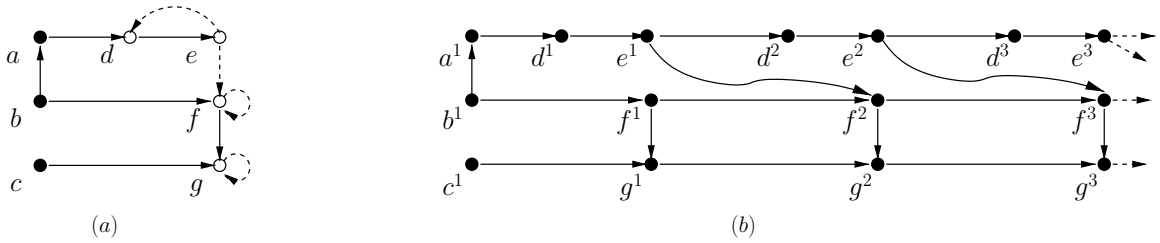


**Fig. 2.** (a) A d-diagram and (b) its corresponding infinite poset

the corresponding infinite computation. The formal semantics of a d-diagram is given in Section 3. Intuitively, the infinite poset corresponds to the infinite unrolling of the recurrent part of the d-diagram.

The third component of our technique is to develop efficient algorithms for analyzing the infinite poset given as a d-diagram. Two kinds of computation analysis have been used in past for finite computations. The first analysis is based on vector clocks which allows one to answer if two events are dependent or concurrent, for example, works by Fidge[5] and Mattern[6]. We extend the algorithm for timestamping events of a finite poset to that for the infinite poset. Of course, since the set of events is infinite, we do not give explicit timestamp for all events, but only an implicit method that allows efficient calculation of dependency information between any two events when desired. The second analysis we use is to detect a global predicate $B$ on the infinite poset given as a d-diagram. In other words, we are interested in determining if there exists a consistent global state which satisfies $B$. Since the computation is infinite, we cannot employ the traditional algorithms [2, 3] for predicate detection. Because the behavior is periodic it is natural that a finite prefix of the infinite poset may be sufficient to analyze. The crucial problem is to determine the number of times the recurrent part of the d-diagram must be unrolled so that we can guarantee that $B$ is true in the finite prefix *iff* it is true in the infinite computation. We show in this paper that it is sufficient to unroll the d-diagram $N$ times where $N$ is the number of processes in the system.

We note here that there has been earlier work in detection of temporal logic formulas on distributed computation, such as [7–10]. However, the earlier work was restricted to verifying the temporal logic formula on the finite computation with the interpretation of liveness predicates modified to work for finite posets. For example, the interpretation of "a hungry philosopher never gets to eat" was modified to "a hungry philosopher does not eat by the end of the computation." This interpretation, although useful in some cases, is not accurate and may give false positives when employed by the programmer to detect bugs. This paper is the first one to explicitly analyze the periodic behavior to ensure that the interpretation of formulas is on the infinite computation.

In summary, this paper makes the following contributions:

– We introduce the notion of recurrent global states in a distributed computation and propose a method to detect them.

– We introduce and study a finite representation of infinite directed computations called d-diagrams.
– We provide a method of timestamping nodes of a d-diagram so that the happened-before relation can be efficiently determined between any two events in the given infinite computation.
– We define the notion of core of a d-diagram that allows us to use any predicate detection algorithm for finite computations on infinite computations as well.

## 2 Model of Distributed Computation

We first describe our model of a distributed computation. We assume a message passing asynchronous system without any shared memory or a global clock. A distributed program consists of $N$ sequential processes denoted by $P = \{P_1, P_2, \ldots, P_N\}$ communicating via asynchronous messages. A *local computation* of a process is a sequence of events. An event is either an internal event, a send event or a receive event. The predecessor and successor events of $e$ on the process on which $e$ occurs are denoted by $pred(e)$ and $succ(e)$.

Generally a distributed computation is modeled as a partial order of a set of events, called the *happened-before relation* [11]. In this paper, we instead use directed graphs to model distributed computations as done in [9]. When the graph is acyclic, it represents a distributed computation. When the distributed computation is infinite, the directed graph that models the computation is also infinite. An infinite distributed computation is *periodic* if it consists of a subcomputation that is repeated forever.

Given a directed graph $G = \langle E, \rightarrow \rangle$, we define a *consistent cut* as a set of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. For example, the set $C = \{a^1, b^1, c^1, d^1, e^1, f^1, g^1\}$ is a consistent cut for the graph shown in figure 3(b). The set $\{a^1, b^1, c^1, d^1, e^1, g^1\}$ is not consistent because it includes $g^1$, but does not include its incoming neighbor $f^1$. The set of finite consistent cuts for graph $G$ is denoted by $\mathcal{C}(G)$.

In this work we focus only on finite consistent cuts (or *finite order ideals* [12]) as they are the ones of interest for distributed computing.

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

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

For the cut $C$ in figure 3(b), $frontier(C) = \{e^1, f^1, g^1\}$. A consistent cut is uniquely characterized by its frontier and in this paper we always identify a consistent cut by its frontier.

Two events are said to be *consistent* iff they are contained in the frontier of some consistent cut, otherwise they are inconsistent. It can be verified that events $e$ and $f$ are consistent iff there is no path in the computation from $succ(e)$ to $f$ and from $succ(f)$ to $e$.

## 3 Infinite Directed Graphs

From distributed computing perspective, our intention is to provide a model for an infinite computation of a distributed system which eventually becomes periodic. To this end, we introduce the notion of *d-diagram* (directed graph diagram).

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

Figure 2(a) is an example of a d-diagram. The recurrent vertices and non-recurrent vertices in the d-diagram are represented by hollow circles and filled circles respectively. The forward edges are represented by solid arrows and the shift-edges by dashed arrows. The recurrent vertices model the computation that is periodic.

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

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

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

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

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

Although acyclic d-diagrams cannot represent all infinite posets, they are sufficient for the purpose of modeling distributed computations. Let the *width* of a directed graph be defined as the maximum size of a set of pairwise concurrent vertices. A distributed computation generated by a finite number of processes has finite width and hence we are interested in only those d-diagrams which generate finite width directed graphs. The following property of the posets generated by acyclic d-diagrams is easy to show.

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

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

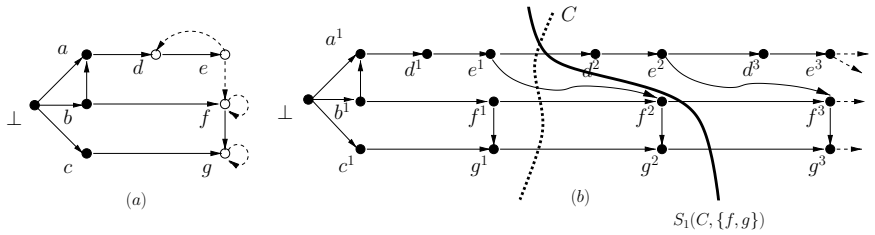


**Fig. 3.** (a) A d-diagram (b) The computation for the d-diagram showing a cut and the shift of a cut

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

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

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

Figure 2 shows a d-diagram and the corresponding computation. Note that for any two events $x, y$ on a process, either there is a path from $x$ to $y$ or from $y$ to $x$, i.e., two events on the same process are always ordered.

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

**Definition 3 (d-shift of a cut).** *Given a frontier $C$, a set of recurrent events $X \subseteq R$ and an integer $d$, a* d-shift cut *of $C$ with respect to $X$, is represented by the frontier $S_d(C, X)$*

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

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

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

As an example, consider the infinite directed graph for the d-diagram in figure 3. 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 3 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.

## 4 Vector Clock Timestamps

In this section, we present algorithms to assign vector timestamps to nodes in the infinite computation given as a d-diagram. The objective is to determine dependency between

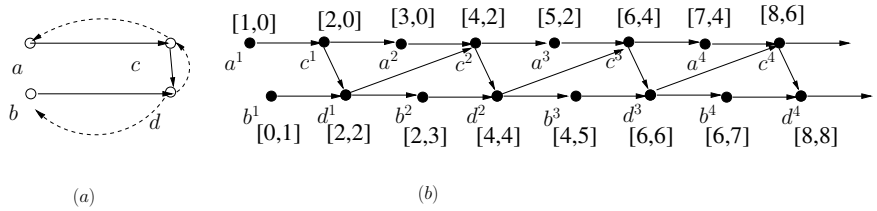


**Fig. 4.** (a) A d-diagram(b) The corresponding computation with vector timestamps

any two events, say $e^i$ and $f^j$, based on the vector clocks assigned to these events rather than exploring the d-diagram. Since there are infinite instances of recurrent events, it is clear that we can only provide an implicit vector timestamp for the events. The explicit vector clock can be computed for any specific instance of $i$ and $j$.

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

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

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

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

**Definition 4 (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 we mean the path with minimum number of shift-edges.*

When $Q$ is clear from the context, we simply use $\eta$ to denote $\eta(Q)$. For the d-diagram in Figure 3, $\eta = 1$. In figure 4, we can see that $\eta = 2$. We first give a bound on $\eta$.

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

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

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

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

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

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

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

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

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

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

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

Each recurrent event, $e$, has a special *p-timestamp* ($PV(e)$) associated with it, which lets us compute the time stamp for any arbitrary iteration of that event. Therefore, this

result gives us an algorithm for assigning p-timestamp to a recurrent event. The p-timestamp for a recurrent event $e$, $PV(e)$ would be a list of the form

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

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

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

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

The notion of vector clock also allows us to keep only the *relevant* events[13] of the d-diagram. Any dependency related question on the relevant events can be answered by simply examining the vector timestamps instead of the entire d-diagram.

## 5 Detecting Global Predicates

We now consider the problem of detecting predicates in d-diagrams. A predicate is a property defined on the states of the processes and possibly channels. An example of a predicate is "more than one philosopher is waiting."

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

It is easy to see that it does not suffice to detect the predicate on the d-diagram without unrolling it. As a simple example, consider figure 4, where though $\{a^1, d^1\}$ is not a consistent cut, but $\{a^2, d^1\}$ is consistent.

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

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

Predicate detection algorithms explore the lattice of global states in BFS order as in Cooper-Marzullo [2] algorithm, or a particular order of events as in Garg-Waldecker [14] algorithm. For finite directed graphs, once the exploration reaches the final global state it signals that the predicate could never become true. In the case of infinite directed

graphs, there is no final global state. So, the key problem is to determine the stopping rule that guarantees that if the predicate ever becomes true then it would be discovered before the stopping point. For this purpose, we show that for every cut in the computation, a subgraph of the computation called the *core* contains a cut with the same label. The main result of this section is that the core of the periodic infinite computation is simply the set of events in the computation with iteration less than or equal to $N$, the number of processes.

**Definition 5 (core of a computation).** *For a d-diagram $Q$ corresponding to a computation with $N$ processes, we define $U(Q)$, the* core *of $Q$, as the directed graph given by the set of events $E' = \{e^j | e \in R \wedge 2 \leq j \leq N\} \cup \{e^1 | e \in V\}$ and the edges are the restriction of $\rightarrow$ to set $E'$.*

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

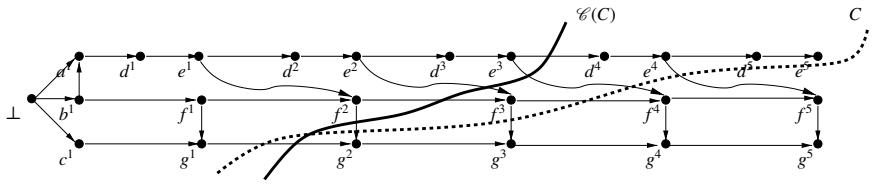


**Fig. 5.** Compression operation being applied on a cut

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

**Definition 6 (Compression).** *Given a frontier $C$ and index $i$ , define $\mathscr{C}(C, i)$ as shifting of all events with index greater than $i$ by sufficient iterations such that in the shifted frontier the event with next higher index than $i$ is $i + 1$. The cut obtained after all possible compressions is denoted as $\mathscr{C}(C)$.*

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

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

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

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

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

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

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

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

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

## 6 Recurrent Global State Detection Algorithm

We now briefly discuss a method to obtain a d-diagram from a finite distributed computation. The *local state* of a process is the value of all the variables of the process including the program counter. The *channel* state between two processes is the sequence of messages that have been sent on the channel but not received. A *global state* of a computation is defined to be the cross product of local states of all processes and all the

channel states at any cut. Any consistent cut of the computation determines a unique consistent global state. A global state is *recurrent* in a computation, if there exist consistent cuts $Y$ and $Z$ such that the global states for $Y$ and $Z$ are identical and $Y$ is a proper subset of $Z$. Informally, a global state is recurrent if there are at least two distinct instances of that global state in the computation.

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

The first step of our recurrent global state detection (RGSD) algorithm consists of computing the global state of a distributed system. Assuming FIFO, we could use the classical Chandy and Lamport's algorithm[1] for this purpose. Otherwise, we can use any of the algorithms, such as [15–17]. Let the computed global snapshot be $G$. Let $Z$ be the vector clock for the global state $G$.

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

Note that replaying a distributed computation requires that all nondeterministic events (including the message order) be recorded during the initial execution [18]. Monitoring the computation to determine the least consistent cut that matches $G$ can be done using algorithms for conjunctive predicate detection [3, 19].

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

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

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

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

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

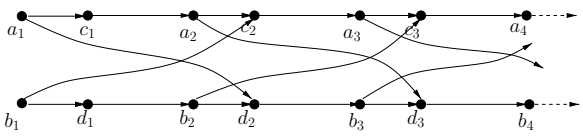


**Fig. 6.** A poset which cannot be captured using MSC graphs or HMSC

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

## 7 Related Work

A lot of work has been done in identifying the classes of predicates which can be efficiently detected [7, 9]. However, most of the previous work in this area is mainly restricted to finite traces.

Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [7, 20], *disjunctive* [7], *observer-independent* [21, 7], *linear* [7], *non-temporal regular* [22, 9] predicates and *temporal* [8, 23, 24].

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) [25] and petri nets [26]. MSCs and related formalisms such as time sequence diagrams, message flow diagrams, and object interaction diagrams are often used to specify design requirements for concurrent systems. An MSC represents one (finite) execution scenario of a protocol; multiple MSCs can be composed to depict more complex scenarios in representations such as MSC graphs and high-level MSCs (HMSC). These representations capture multiple posets but they cannot be used to model all the posets (and directed graphs) that can be represented by d-diagrams. In particular, a message sent in a MSC node must be received in the same node in MSC graph or HMSC. Therefore, some infinite posets which can be represented through d-diagrams cannot be represented through MSCs. Therefore, an infinite poset such as the one shown in figure 6 is not possible to represent through MSCs.

Petri nets [26] are also used to model concurrent systems. Partial order semantics in petri nets are captured through net unfoldings [27]. Unfortunately, unfoldings are usually infinite sets and cannot be stored directly. Instead, a finite initial part of the unfolding, called the finite complete prefix [28] is generally used to represent the unfolding. McMillan showed that reachability can be checked using the finite prefix itself. Later Esparza [29] extended this work to use unfoldings to efficiently detect predicates from a logic involving the $\mathbf{EF}$ and $\mathbf{AG}$ operators. Petri nets are more suitable to model the behavior of a complete system whereas d-diagrams are more suitable for modeling distributing computations in which the set of events executed by a process forms a total order. They are a simple extension of process-time diagrams[11] which have been used extensively in distributed computing literature.

## 8 Conclusion

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

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

In this paper, we have restricted ourselves to very simple unnested temporal logic formulas. Detecting a general temporal logic formula efficiently in the model of d-diagram is a future work.

## References


1. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems **3**(1) (1985) 63–75
2. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, ACM/ONR (1991) 163–173
3. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. on Parallel and Distributed Systems **5**(3) (1994) 299–307
4. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual IEEE-ACM Symposium on Foundations of Computer Science. (1977) 46–57
5. Fidge, C.J.: Partial orders for parallel debugging. Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices **24**(1) (1989) 183–194
6. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms. (1989)
7. Garg, V.K.: Elements of Distributed Computing. John Wiley & Sons (2002)
8. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: 7th International Conference on Principles of Distributed Systems, La Martinique, France (2003)
9. Mittal, N., Garg, V.K.: Computation Slicing: Techniques and Theory. In: In Proc. of the $15^{th}$ Int'l. Symposium on Distributed Computing (DISC). (2001)
10. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In Pelc, A., ed.: Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings. Volume 4731 of Lecture Notes in Computer Science., Springer (2007) 420–434
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7) (1978) 558–565
12. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK (1990)

13. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in shared-memory systems. In Aguilera, M.K., Aspnes, J., eds.: PODC, ACM (2005) 19–28
14. Garg, V.K., Waldecker, B.: Detection of unstable predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, ACM/ONR (1991)
15. Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel and Distributed Computing (1993) 423–434
16. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: Proceedings of the ACM Conference on Supercomputing, 2006, ACM (2006)
17. Kshemkalyani, A.D.: A symmetric o(n log n) message distributed snapshot algorithm for large-scale systems. In: CLUSTER, IEEE (2009) 1–4
18. LeBlanc, Mellor-Crummey: Debugging parallel programs with instant replay. IEEETC: IEEE Transactions on Computers **36** (1987)
19. Garg, V.K., Chase, C.M., Kilgore, R.B., Mitchell, J.R.: Efficient detection of channel predicates in distributed systems. J. Parallel Distrib. Comput. **45**(2) (1997) 134–147
20. Hurfin, M., Mizuno, M., Raynal, M., Singhal, M.: Efficient detection of conjunctions of local predicates. IEEE Transactions on Software Engineering **24**(8) (1998) 664–677
21. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and temporal predicates in distributed systems. ACM Transactions on Programming Languages and Systems **17**(1) (1995) 157–179
22. Garg, V.K., Mittal, N.: On Slicing a Distributed Computation. In: Proc. of the $15^{th}$ Int'l. Conference on Distributed Computing Systems (ICDCS). (2001)
23. Sen, A., Garg, V.K.: Detecting temporal logic predicates in the happened before model. In: International Parallel and Distributed Processing Symposium (IPDPS), Florida (2002)
24. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In Pelc, A., ed.: DISC. Volume 4731 of Lecture Notes in Computer Science., Springer (2007) 420–434
25. : Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC). (1996)
26. Petri, C.A.: Kommunikation mit Auto-maten. PhD thesis, Bonn: Institut fuer Instru- mentelle Mathematik (1962)
27. M. Nielsen, G.P., Winskel, G.: Petri nets, event structures and domains. Theoretical Computer Science **13**(1) (1980) 85–108
28. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
29. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming **23**(2) (1994) 151–195