Observation of global properties in distributed systems

Vijay K. Garg[*]
http://maple.ece.utexas.edu/
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

Observation of global properties of a distributed program is required in many applications such as debugging of programs and fault-tolerance in distributed systems. I present a survey of algorithms for observing various classes of global properties. These properties include those possibly true in a computation, definitely true in a computation and those based on the control flow structure of the computation.

1 Introduction

One of the fundamental problems in development of distributed software is that no process has access to the global state. Consequently, computation of any global predicate or a function requires a non-trivial programming effort. For example, consider a distributed debugging system. The detection of global predicate arises in implementing the most basic command of a debugging system:"stop the program when the predicate q is true." To stop the program, it is necessary to detect the predicate q; a non-trivial task if q requires access to the global state.

There have been three approaches in solving the detection of global predicates. The first approach is based on the global snapshot algorithm by Chandy and Lamport [3, 2, 22]. Their approach requires repeated computation of consistent global snapshots of the computation till the desired predicate becomes true. This approach works only for stable predicates, that is, predicates which do not turn false once they become true. Some examples of stable predicates are deadlock and termination. Once a system has terminated it will stay terminated. The desired predicate q may not be stable and may turn true only between two successive snapshots.

The second approach to global predicate detection is based on the construction of the lattice of global states. This approach, first presented by Cooper and Marzullo [6], allows user to detect unstable predicates. However, given n processes each with k "relevant" local states, their approach requires exploring $O(k^n)$ possible global states in the worst case.

The third approach is based on exploiting the structure of the predicate q [15]. This approach, instead of building the lattice, directly uses the computation to deduce if q became true. The emphasis of this approach is to develop practical algorithms albeit for special classes of predicates. In this paper, we present a survey of algorithms that use this approach.

2 Our Model

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of N processes denoted by $\{P_1,P_2,...,P_N\}$ communicating via asynchronous messages. In this paper, we will be concerned with a single run of a distributed program. We assume that no messages are altered or spuriously introduced. We do not make any assumptions about FIFO nature of the channels.

A *local state* is the value of all program variables and processor registers (including the program counter) for a single process. The execution of a process can be viewed as a sequence of local states. We use a causally precedes relation, '$\rightarrow$,' between states similar to that of Lamport's causally precedes relation between events [18]. The causally precedes relation between two states s and t can be formally stated as: $s \rightarrow t$ iff s occurs before t in the same process, or the action following s is a send of a message and the action preceding t is a receive of that message, or there exists a state u such that s causally precedes u and u causally precedes t. Two states s and t are *concurrent* if s does not causally precede t and t does not causally precede s. A *cut* is a collection of local states such that exactly one state is

[*]supported in part by the NSF Grants ECS-9414780, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

included from each process. A cut is called a *consistent cut* or a *global state* if all states are pairwise concurrent. The set of all global states form a lattice [19].

A *local predicate* is defined as any boolean-valued formula on a local state. For any process, P_i, a local predicate is written as l_i. A process can obviously detect a local predicate on its own.

We categorize the properties that we survey in this paper into three classes. The first class of predicates are of the form *possibly*: q where q is any predicate defined on a single global state [6]. The predicate *possibly*: q is true if in the lattice of global states there is a path from the initial global state to the final global state in which q is true in some intermediate state. The second class of predicates are of the form *definitely*: q. The predicate *definitely*: q is true if q becomes true in all paths from the initial state to the final state in the lattice of global states. *Possibly*: q and *definitely*: q roughly correspond to weak and strong predicates in [15]. Possibly true predicates are useful for detecting bad conditions such as violation of mutual exclusion, whereas definitely true predicates are useful to verify occurrence of good predicates such as commit point on transaction systems. The third class includes poset based predicates that require more than one global state for their evaluation. An example is a sequence of local predicates $l_1 \rightarrow l_2 \rightarrow \ldots l_n$. This predicate is true in an execution if and only if there exists a sequence of local states $s_1, s_2, ..., s_n$ sequenced by Lamport's causally precedes relation such that l_i is true in the local state s_i for $1 \leq i \leq n$. This type of predicate cannot be evaluated on an individual global state. We discuss each of these classes next.

3 Possibly True Predicates

In a distributed computation, *possibly*: q is true if and only if there exists a consistent global state in which q is true. The method of constructing all global states is, however, prohibitively expensive for most applications. Even when q is a boolean expression, and processes do not communicate, the problem of detecting *possibly*: q is NP-complete. The proof of NP-completeness of this problem first presented in [4] is as follows. The problem is in NP, since a non-deterministic Turing machine can guess the global state and then verify that q is indeed true in that global state. To see NP-hardness, consider the satisfiability problem on boolean variables $x_1, x_2, ..., x_n$. By defining each process to host a single variable which takes value false and true in two states, it is easy to see that a given boolean expression q is true iff *possibly*: q is true in the computation.

Even though the problem is NP-complete for general boolean expressions, there exist efficient algorithms for several classes of q which occur in practice. In this section, we survey classes of q for which efficient algorithms are known.

3.1 Conjunctive Predicates

A weak conjunctive predicate (WCP) is of the form $possibly : \ (l_1 \wedge l_2 \wedge ... \wedge l_n)$ where each l_i is a predicate local to a single process. For example, suppose we are developing a mutual exclusion algorithm between two processes. Let CS_i represent the local predicate that the process P_i is in the critical section. Then, the WCP formula $possibly : (CS_1 \wedge CS_2)$ represents any possibility of violation of mutual exclusion for a particular run. As another example, let $Read_i$ ($Write_i$) denote that P_i has a read lock (write lock resp.) on a shared data item. Then, $possibly : Read_1 \wedge Write_2$ represents the condition that P_1 has a read lock and concurrently P_2 has a write lock. Note that the interpretation of conjunction is not with respect to time, but with respect to causality. That is, $possibly : Read_1 \wedge Write_2$ may be true even if there is no instant of time in which P_1 has the read lock and P_2 has the write lock. An advantage of this interpretation is that WCP algorithms detect even those conditions which may not manifest themselves in a particular execution, but which would show up with different processing speeds. For example, in a distributed mutual exclusion algorithm it may be possible that two processes do not access the critical region *simultaneously* even if they both had permission to enter the critical region *concurrently*. WCP algorithms will detect such a scenario.

From the definition, $possibly : (l_1 \wedge l_2 \wedge ... \wedge l_n)$ is true if there exists a consistent cut in which all local predicates are true. There can be two approaches to detect a WCP. We could form consistent cuts and check whether $(l_1 \wedge ... \wedge l_n)$ is true in that cut. Alternatively, we could piece together local states in which the local predicates are true and check whether the global state so formed is consistent. The second approach has the advantage that it needs to examine only those local states in which local predicates are true. Therefore, we follow the second approach.

In the second approach of predicate detection, there are two main difficulties for efficient detection. First, the total number of local states in which the local predicate is true may be quite large. Second, there is a combinatorial explosion if we construct all possible global states by piecing together all local states. For example, even if we have only two states from each of n processes, there are 2^n possible global states.

We solve the first problem by observing that if two local states say s and t on the same process are separated only by internal events, then they are indistinguishable to other processes so far as consistency is concerned. That is, if u is a local state on some other process, then $s||u$ if and only if $t||u$. Thus, it is sufficient to consider at most one local state between two external events. A slightly more detailed argument shows that it is sufficient to consider at most one local state between two send events [15]. This observation leads to a significant reduction in complexity since we expect the total number of external events to be much less than the total number of events in a process.

We now turn our attention to the problem of combinatorial explosion. Assume that we are considering a global state G formed by collecting one local state from each process. If G is consistent, we are done. Otherwise, there exist $s, t \in G$ such that $s \rightarrow t$. A crucial observation is that $s \rightarrow t$ implies that not only G is inconsistent but also all global states which include state s and any state u following t are inconsistent. In other words, we can eliminate state s from our consideration altogether. The only thing that remains is the ability to determine whether $s \rightarrow t$. But this can be easily accomplished using vector clocks [8, 19].

We are now ready to present a centralized algorithm, a token-based decentralized algorithm and a completely distributed algorithm to detect WCP.

Centralized Algorithm

The centralized algorithm for WCP [15] requires every process to send its vector clock to a centralized checker process whenever its local predicate becomes true for the first time between two external events by that process. Assuming that a process sends or receives at most m messages, the checker process will receive at most mn such vectors. We will assume that the checker process receives vectors from any process in a FIFO order and stores them in a queue. The checker process can then simply check whether all vectors at the head of the queue are incomparable. If they are, the checker process has succeeded in finding a consistent cut. Otherwise, the checker process can discard any vector which is found to be smaller than some other vector. Since it takes at most n comparisons before a vector is discarded and there are at most mn vectors, this algorithm requires $O(n^2m)$ comparisons to determine if the WCP was true in that run. Further, [10] shows that any algorithm based on comparing vector clocks requires at least $\Omega(n^2m)$ comparisons.

Token-based decentralized Algorithm

The above algorithm requires the checker process to keep queues of vectors for all processes. This may impose unreasonable space and time requirements for some applications. We now discuss a decentralized algorithm first presented in [12]. With each application process, we use a monitor process M_i that maintains the queue of local snapshots of P_i. Further, instead of a checker process, this algorithm uses a token which carries in it the candidate global state and information sufficient to determine if the global state satisfies the WCP. That is, the token contains two vectors. The first vector is labeled *G*. This vector defines the current candidate cut. If $G[i]$ has the value k, then state k from process P_i is part of the current candidate cut. Note that all states on the candidate cut satisfy local predicates. However, the states may not be mutually concurrent (i.e. the candidate cut may not be a consistent cut). The token is initialized with $\forall i : G[i] = 0$.

The second vector is labeled $color$, where $color[i]$ indicates the color for the candidate state from application process P_i. The color of a state can be either red or green. If $color[i] = red$ then the state $G[i]$ and all its predecessors have been eliminated and can never satisfy the WCP. If $color[i] = green$, then there is no state in G such that $G[i]$ causally precedes that state. The token is initialized with $\forall i : color[i] = red$.

The token is sent to monitor process M_i only when $color[i] = red$. When it receives the token, M_i waits to receive a new candidate state from P_i and then checks for violations of consistency conditions with this new candidate. This activity is repeated until the candidate state did not causally precede any other state on the candidate cut (i.e. the candidate can be labeled green). Next, M_i examines the token to see if any other states violate concurrency. If it finds any j such that $G[j]$ causally precedes $G[i]$, then it makes $color[j]$ red. Finally, if all states in G are green, that is G is consistent, then M_i has detected the WCP. Otherwise, M_i sends the token to a process whose color is red. Note that the token can start on any process. Since the entire color vector is initialized to red, it must eventually visit every process at least once.

This algorithm requires $O(n^2m)$ total work and $O(nm)$ work per process where m is the number of messages sent or received by any process and n is the number of processes over which the predicate is defined.

One drawback of the single-token WCP detection algorithm is that it has no concurrency — a monitor process is active only if it has the token. We can increase the parallelism in the algorithm by using multiple tokens in the system. For this we partition the set of monitor processes into groups and use one token-algorithm for each group. Once there are no longer any red states from processes within the group, the token is returned to a pre-determined process (say P_0). When P_0 has received all the tokens, it merges the information in the tokens to identify a new global cut. Some processes may not satisfy the consistency condition for this new cut. If so, a token is sent into each group containing such a process.

Distributed Algorithm

An even more parallel algorithm can be used as follows. As in the token based algorithm, each process maintains its queue of local snapshots. Each process P_i attempts to color itself green as follows. Let the local snapshot at the head of its queue be s. The predecessor of this state on any process P_j can be determined from the vector clock at the state $s.v$. To color itself green, the process sends out messages *red* messages containing $s.v[j]$ to all processes P_j. On receiving a red message with k, P_j needs to discard all local snapshots in which its local state is less than or equal to

k. The algorithm implicitly terminates when all processes are green and there are no red messages in transit. The termination can be detected by any termination detection algorithm such as [7].

If the global predicate involves local predicates from all processes, then direct dependences instead of vector clock can be used. This is because a cut which consists of all processes in the system is consistent if and only if there is no message which is received in the cut but not sent. This is not true when the cut does not contain all processes - there may be indirect causal precedence between two states in the cut which goes through a process outside the cut. It is easy to adapt the centralized algorithm, the token based algorithm, or distributed algorithm to use direct dependency instead of vector clocks [12].

Distributed algorithms for off-line evaluation of global predicates are also discussed in [28].

3.2 Channel Predicates

Many properties in distributed systems such as termination detection and bounding of global virtual time [24] are based on the state of message channels. Therefore, they are not suitable for specification via weak conjunctive predicates which are based on states of processes only. Conjunctive channel predicates are an extension of weak conjunctive predicates to include states of message channels.

A *channel predicate* is any boolean function of the state of a channel. We define a channel to be a uni-directional connection between two processes — one process performs all send events and the other all receive events. Let s and t be states at P_i and P_j. Let $s.Sent[j]$ denote the sequence of all messages sent at or before state s from i to j, and $t.Rcvd[i]$ denote the sequence of all messages received at or before state t from i to j. Channels have no memory. Hence, the state of a channel is the difference between the set of messages sent and the set of messages received. A channel predicate can then be written as: $chanp(s.Sent[j] - t.Rcvd[i])$. A global predicate formed by the conjunction of local predicates and channel predicates is called a Generalized Conjunctive Predicate (GCP). An example of a GCP is: “all processes are passive and all channels are empty,”.

We now discuss a centralized algorithm for channel predicates [13]. The key to making our algorithm efficient is to restrict the channel predicates to a class which we call *linear*. A channel predicate is linear if given any channel state, in which the predicate is false, then either sending more messages without receiving any message is guaranteed to leave the predicate false, or receiving more messages without sending any messages is guaranteed to leave the predicate false. For example, consider the condition “the channel is empty.” If this condition is false, that is, more messages have been sent than received, it cannot be made true by sending more messages. As another example, consider the predicate, “The channel contains exactly 5 messages”. When the channel contains less than 5 messages, receiving more messages will not make the predicate true. If there are more than 5 messages in the channel, then sending more messages cannot make the predicate true. The channel predicate, “the channel has an even number of messages” is not linear. Most channel predicates used in practice are linear.

Linearity is an important key to efficient detection of channel predicates. In any global state in which the predicate is false, we can be certain of at least one process which must make further progress before the channel predicate can become true. Furthermore, it can be shown that the first global state satisfying a GCP can be well defined only when channel predicates are linear [13].

We first discuss a centralized algorithm to detect a GCP. The work of detection of the GCP is divided among checker and non-checker processes. The non-checker processes are used in the computation and have local predicates and channels with predicates. The checker process is the process that determines if these predicates are true in the same global state.

The non-checker processes monitor local predicates. These processes also maintain information about the send and receive channel history for all channels incident to them, that is, connections to all processes for which they can send or receive messages. The non-checker processes send a message to the checker process whenever the local predicate becomes true for the first time since the last program message was sent or received. This message is called a local snapshot and is of the form: *(vector, incsend, increcv)* where $vector$ is the current vector timestamp while $incsend$ and $increcv$ are the list of messages sent to and received from other non-checker processes since the last message for predicate detection was sent.

The checker process is responsible for searching for a consistent cut that satisfies the GCP. Its pursuit of this cut can be most easily described as considering a sequence of candidate cuts. If the candidate cut either is not a consistent cut, or does not satisfy some term of the GCP (local predicate or a channel predicate), the checker can efficiently eliminate one of the states along the cut. The eliminated state can never be part of a consistent cut that satisfies the GCP. The checker can then advance the cut by considering the successor to one of the eliminated states on the cut. If the checker finds a cut for which no state can be eliminated, then that cut satisfies the GCP and the detection algorithm halts.

The algorithm can also be decentralized based on ideas discussed for WCP algorithm. For example, we briefly discuss a decentralized algorithm based on the idea of a token. Each process is responsible for keeping its queue of local snapshots. As in the WCP algorithm, the token moves from one process to another till a consistent cut is found. Each process is also responsible for checking channel predicates for all channels for which it is a sender. To enable the sender to do so, the receiver for any channel sends the list of messages (or the list of message sequence numbers) received along the channel upto the state which is indicated in the token. The sender evaluates the channel predicate only when it has received this list from the receiver. If any channel predicate is found false, then either the sender or the receiver can be colored red. The GCP is detected by the token if all states in its cut are green. A more detailed description of this algorithm and its proof of correctness can be found in [13].

We note here that if a predicate is stable, then either the approach outlined above, or Chandy and Lamport's algorithm can be used for predicate detection. We now argue that even for stable predicates it is advantageous to use the general algorithm shown here. First, in many applications (such as debugging), it is desirable to compute the least global state which satisfies some given predicate. The snapshot algorithms cannot be used for this purpose. Second, the snapshot algorithm may result in excessive overhead depending upon the frequency of snapshots. A process in Chandy and Lamport's algorithm is forced to take a local snapshot upon receiving a marker even if it knows that the global snapshot that includes its local snapshot cannot satisfy the predicate being detected. For example, suppose that the property being detected is termination. Clearly, if a process is not terminated then the entire system could not have terminated. In this case, computation of the global snapshot is a wasted effort.

We also note here that the algorithm for GCP can be optimized by exploiting specific properties of the channel predicate. For example, to check whether a channel is empty it may suffice to deal with the number of messages rather than message themselves. Such optimizations are discussed in detail in [21].

3.3 Relational Predicates

So far, we have discussed only those predicates which can be written as boolean expression of local predicates. Now consider the predicate $(x_1 + x_2 + \ldots + x_n < k)$ where x_i's are variables on different processes and k is a constant. This predicate called *relational predicate* cannot be written as a *concise* boolean expression of local predicates.

Relational predicates are useful for detecting global conditions such as loss of tokens and violations of a limited resource. For example, consider a system in which there are k tokens indicating availability of k resources. If x_i denotes the number of tokens at process P_i, then $\sum_i x_i < k$ indicates loss of one or more tokens. As another example, consider a server which can handle at most k connections at a time. Client processes P_i have variables x_i which indicates the number of connections it has with the server. The predicate $(\sum_i x_i > k)$ indicates a potential error.

The predicate to be detected, previously expressed as $(x_1 + x_2 + \ldots + x_n < k)$, can be stated formally as:

$$\exists G : \text{consistent}(G) : \sum_{s_i \in G} s_i.x_i < k$$

We now discuss an algorithm first presented in [4]. We detect this predicate by computing

$$\min G : \text{consistent}(G) : \sum_{s_i \in G} s_i.x_i$$

and then comparing this value to the constant k.

We transform the poset into a flow graph such that the max-flow in the graph is equal to the min-value of the poset. The resulting flow graph is obtained as follows. The vertex set of the graph includes all local states and two additional nodes called source and sink. The edge set is given below.

- First, we add edges from the *source* to all initial states s with the capacity ∞.
- For any two states s and t such that s immediately precedes t, we add an edge between them with capacity $s.x$.
- We add edges from all final states s to the *sink* with the capacity $s.x$.
- For any two states s and t such that a message is sent immediately after s which is received before t, we first identify the successor to s, say s'. We then add an edge from t to s' with capacity ∞.

Note that the cut of G has finite value if and only if the cut is a consistent cut of S. We relate a cut in the flow graph to a cut in the poset as follows: If edge e connects vertices s and t in G, and if e is part of the cut of flow graph G, then the state corresponding to s is part of the cut in poset S. The min-value of a poset S is equal to the min cut of its flow graph G.

Based on the above result, a checker based algorithm can be devised as follows. First, the sequence of states from each process is reduced by replacing the subsequence of states between any two message events with a single state. The value of x_i for this new state is defined as the minimum of x_i over the original states. Second, each process locally maintains the direct dependence relation for each state. Each process creates a local snapshot for every state, consisting of the value of x_i and the direct dependence information. The local snapshots are sent to a checker process which forms the flow-graph. The checker then runs a max-flow algorithm to find the min cut. If this value is less than k, then the bounded sum predicate is detected.

There are other approaches possible for relational predicates. In [25], we discuss relational global predicates which have the form $(x_1+x_2 \geq k)$, where x_1 and x_2 are integer values at processes P_1 and P_2 in a system of N processes. The algorithm is fully decentralized, runs concurrently with the target program, uses constant size message tags (four integers), and generates one debug message for each message received by P_1 and P_2. The results have been generalized to an algebra $(D, \%, *)$ where % and $*$ are binary operators in domain D, % is commutative, associative and idempotent, and $*$ distributes over %. In this algebra we can calculate value of the expression $(v_1 \% v_2 \% ... \% v_n)$ where $\{v_1, v_2, ...v_n\}$ is the set which contains the value of $x_1 * x_2$ in each consistent cut. For example if $(D, \%, *) = (\text{Integers}, \min, +)$ then we could calculate the minimum value of $x_1 + x_2$ over all global states.

In [26], we present another special case of relational predicates. Here we assume that x_i are boolean variables. Such predicates are useful, for example, in detecting violation of k-mutual exclusion. In this case, even though the predicate $x_1 + x_2 .. + x_n \geq k$ can be written as a disjunction of conjunctive predicates, it is not efficient to do so since there are $\binom{n}{k}$ conjuncts in the boolean expression. Our algorithm is based on finding an anti-chain of size k in the poset of states in which the boolean predicate is true.

3.4 General Possibly true Predicates

The concept of *linearity* of channel predicates has been generalized to apply for any general predicate in [4]. A predicate is defined to be linear if its falsehood on any global state G implies that there exists at least one state s in G such that the predicate is also false for any global state $H \geq G$ containing s. We call s a forbidden state in G. It is easy to see that weak conjunctive predicates are linear. Linearity of a predicate can be exploited for a simple detection algorithm. If the predicate is false along a cut, then at least one state in that cut is forbidden and can be discarded.

In another approach, Stoller and Schneider [23] combine Garg and Waldecker's algorithm with that of Cooper and Marzullo to detect predicates of the form

$$\wedge_j \ \Phi_j(x_1, ..x_k), \tag{1}$$

where $\Phi()$ is a predicate with variables, x_i, from different processes. That is, $\Phi()$ is a predicate made up of conditions spread across multiple processes. An example of a predicate in this form is $(x_1 = x_2) \wedge (x_3 > x_4)$, where $x_1, .., x_4$ are variables on different processes. For any predicate defined using equation 1, they define a *fixed set* as the set of variables such that on fixing these variables, the predicate reduces to a WCP. In our example, if we fix $x_1 = 4$ and $x_4 = 6$, we get $(4 = x_2) \wedge (x_3 > 6)$ which a WCP. By evaluating all WCP predicates obtained by using all possible values of the variables in fixed set, the original predicate can be detected.

4 Definitely True Predicates

We now discuss detection of predicates of the form *definitely*: q. Intuitively, *definitely*: q is true when q is true for all possible observations of that execution. We will restrict our attention to strong conjunctive predicates (SCP) in which q of the form $l_1 \wedge l_2 \wedge \ldots l_n$. For example, suppose we were testing a commit protocol. Let $Ready_i$ denote the local predicate that the process P_i is ready to commit. If the transaction was committed, then for all possible observations, there was a certain point in the execution when all processes were ready to commit. By detecting the SCP formula *definitely*: $(Ready_1 \wedge Ready_2 \ldots \wedge Ready_n)$ existence of such a point can be verified. The key concept in detecting SCP's is that of overlapping intervals. Let I_1 and I_2 be two sequences of contiguous states such that local predicates l_1 and l_2 are true in I_1 and I_2 respectively. We say that I_1 and I_2 overlap if the lower end point of I_1 causally precedes the higher endpoint of I_2 and vice-versa. An important result is that the SCP is true iff there exist intervals in which local predicates are true such that any pair of these intervals overlap. The proof of this result can be found in [16]. Based on this condition, algorithms to detect SCP can be developed in a manner similar to detection of WCP.

5 Poset based Predicates

The predicates we have discussed so far are based on formulas defined on a single cut. Informally, these predicates capture violation of safety properties. Many useful properties requires evaluation of formulas on a sequence of cuts.

5.1 Sequences of Local Predicates

An early work in this area is by Miller and Choi who discuss detection of a sequence of local predicates [20]. An example is a predicate $l_1 \rightarrow l_2$ that becomes true when there are two states s_1 and s_2 such that l_1 is true in state s_1, l_2 is true in state s_2 and $s_1 \rightarrow s_2$. Hurfin, Plouzeau and Raynal [17] extended the sequence of local predicates to the *atomic sequence of local predicates*. In this class, occurrences of local predicates can be forbidden between adjacent predicates in a sequence of local predicates. The example given above for linked predicates could be expanded to include: "local predicate r_i never occurs in between local predicates l_i and l_{i+1}". Each local predicate can belong to a different process in the computation. This can be further generalized to detect interval-constrained sequences of *global* predicates as shown in [1]. This approach, however, requires traversal of the lattice of the global states.

The work on sequence of predicates has also been generalized to detect any regular pattern of local predicates [9]. A regular pattern is defined as a regular expression of local predicates. For example, pq^*r is true in a computation if there exists a sequence of consecutive local states $(s_1, s_2, \ldots, s_n)$ such that p is true in s_1, q is true in $s_2, \ldots, s_{n-1}$, and r is true in s_n. Note that the states in the sequence need not belong to the same process – two states are consecutive if they are adjacent in the same process or one sends a message and the other receives it.

The algorithm for detecting regular patterns is very efficient. First the regular expression is converted into a deterministic finite state machine. Assume that there are m states in the state machine. To avoid any confusion we refer to the states of the distributed computation as local states in this section. Now with each local state s we keep a boolean bit string $X[1..m]$ such that $X[i]$ is 1 iff the state $X[i]$ can be reached by traversing a sequence of local states that ends in s. Observe that it is sufficient to give the update rules for X because if any of the final state $X[i]$ becomes 1, then the regular pattern has been detected. The boolean string $X[i]$ is easy to obtain given the boolean strings for its predecessor states. For example, if $X[i]$ is true in the predecessor local state, local predicate p is true in the current local state and the finite state machine moves from state $X[i]$ to $X[j]$ on label p, then $X[j]$ is set to 1 in the current local state. To ensure that any local state has access to X of predecessor local states, the bit strings are piggybacked with messages. Thus, this algorithm detects a regular pattern with no additional messages. Existing messages are tagged with a fixed number of bits independent of the number of processes in the system.

Regular patterns can be extended to include patterns on rooted directed acyclic graphs (dag) which are subposets of the original poset. A linear sequence of states is a special case of a rooted dag, hence regular patterns are a special case of regular dag patterns. Many program behaviors which could be easily described by regular dag patterns cannot be described with existing mechanisms. This is true even for fundamental behaviors such as data scattering, data collection, and barrier synchronization. Moreover, detection of rooted dag patterns is inherently efficient due to the structural similarities between the specified dag pattern and the program under test. An algorithm for detecting dag patterns can be found in [14].

5.2 Poset Logic

So far we have discussed predicates which are either based on a single global state or a sequence of local states. Chiou and Korfhage [5] discuss a method of combining concurrency with sequencing. This has been generalized to a recursive logic called RCL in [27]. A formula in RCL is evaluated on a poset. One can think of a formula as a boolean function whose argument is a poset. The rules for constructing well formed formulas are given by the syntactic definitions shown below:

$$\begin{aligned} f &= S \mid f \wedge f \\ S &= g \mid g\langle f\rangle S \mid g\langle\langle f\rangle\rangle S \end{aligned}$$

The basic component of a formula is a weak conjunctive predicate which is represented by the terminal symbol g. The symbol S is a sequence of WCP formulas. The symbol f is a conjunction of these sequences. The symbols $\langle\rangle$ and $\langle\langle\rangle\rangle$ are used for weak and strong sequencing respectively. A cut g weakly precedes h if it lies in the causal past of h. A cut g strongly precedes h iff every state in g causally precedes every state in h.

When S is fully expanded, it has the form $g\langle f\rangle g\langle f\rangle g \ldots g\langle f\rangle g$ (or a sequence with $\langle\langle\rangle\rangle$). When such a sequence is true on a poset, then each g corresponds to an antichain. The regions in between these antichains are subposets upon which the f's in the sequence are evaluated. An efficient algorithm to detect any formula in RCL is given in [27].

6 Conclusions

Observation of a distributed computation is an useful abstraction for many fundamental problems in distributed software. In this paper, we have presented a survey of efficient algorithms for observation. Many of these algorithms are discussed in greater detail in [11].

Acknowledgements

I would like to thank C. M. Chase, E. Fromentin, R. Kilgore, J. R. Mitchell, M. Raynal, A. I. Tomlinson, and B. Waldecker for collaborating with me on the work reported in this paper.

References

[1] O. Babaoğlu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distributed Computing*, 28:173–185, 1995.

[2] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

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

[4] C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem. In *Proc. of the Workshop on Distributed Algorithms*, pages 303 – 317, France, Sept. 1995.

[5] H. Chiou and W. Korfhage. Efficient global event predicate detection. In *14th Intl. Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[6] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

[7] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, Aug. 1980.

[8] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, Jan. 1989.

[9] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc. of the 23rd Intl. Conf. on Parallel Processing*, pages 2:73 – 76, St. Charles, IL, Aug. 1994.

[10] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, Nov. 1992.

[11] V. K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publishers, Boston, MA, 1996.

[12] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.

[13] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting conjunctive channel predicates in a distributed programming environment. In *Proc. of the International Conference on System Sciences*, volume 2, pages 232–241, Maui, Hawaii, Jan. 1995.

[14] V. K. Garg, A. I. Tomlinson, E. Fromentin, and M. Raynal. Expressing and detecting general control flow properties of distributed computations. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 432 – 438, San Antonio, TX, Oct. 1995.

[15] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, Mar. 1994.

[16] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 1996.

[17] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[20] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the* 8^{th} *International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, July 1988. IEEE.

[21] J. R. Mitchell and V. K. Garg. Deriving distributed algorithms from a general predicate detector. In *The Nineteenth Intl. Computer Software and Applications Conference*, pages 268 – 273, Dallas, TX, 1995. IEEE Computer Society , Washington, DC.

[22] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[23] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the Workshop on Distributed Algorithms*, France, Sept. 1995.

[24] A. I. Tomlinson and V. K. Garg. An algorithm for minimally latent global virtual time. In *7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, May 1993.

[25] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. ACM/ONR.

[26] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal for Parallel and Distributed Computing*, 1994. Submitted.

[27] A. I. Tomlinson and V. K. Garg. Observation of software for distributed systems with rcl. In *Proc. of 15th Conference on the Foundations of Software Technology & Theoretical Computer Science*. Springer Verlag, Dec. 1995. Lecture Notes in Computer Science.

[28] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *Thirtieth Annual Allerton Conference on Communication, Control and Computing*, pages 137–146, Allerton, Illinois, Oct. 1992.