A Distributed Abstraction Algorithm for Online Predicate Detection

Himanshu Chauhan  Vijay K. Garg
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
himanshu@utexas.edu  garg@ece.utexas.edu

Aravind Natarajan  Neeraj Mittal
Dept. of Computer Science
The University of Texas at Dallas
aravindn@utdallas.edu  neerajm@utdallas.edu

Abstract—Analyzing a distributed computation is a hard problem in general due to the combinatorial explosion of the state-space with the number of processes in the system. By abstracting the computation, unnecessary state explorations can be avoided. Computation slicing is an approach for abstracting distributed computations with respect to a given predicate. We focus on regular predicates, a family of predicates that covers many commonly used predicates for runtime verification. We present first distributed online algorithm for computing the slice of a distributed computation with respect to a regular predicate. Our algorithm distributes the work and storage requirements across the system, thus reducing the space and computation complexity per process.

I. INTRODUCTION

Global predicate detection [1] for runtime verification is an important technique for detecting violations of invariants for debugging and fault-tolerance in distributed systems. It is a challenging task on a large system with a large number of processes due to the combinatorial explosion of the state space. The predicate detection problem is not only applicable to conventional distributed systems, but also to multicore computing. With growing popularity of large number of CPU-cores on processing chips [2], some manufacturers are exploring the distributed computing model on chip with no shared memory between the cores, and information exchange between the cores only using message passing [3]. Recent research efforts [4] have shown that with sufficiently fast on-chip networking support, such a message passing based model can provide significantly fast performance for some specific computational tasks. With emergence of these trends, techniques in predicate detection for distributed systems can also be useful for systems with large number of cores.

Multiple algorithms have been proposed in literature for detection of global predicates in both offline and online manner (e.g. [1], [5], [6]). Online predicate detection is important for many system models: continuous services (such as web-servers), collection of continuous observations (such as sensor-networks), and parallel search operations on large clusters. However, performing predicate detection in a manner that is oblivious to the structure of the predicate can lead to large runtime, and high memory overhead. The approach of using mathematical abstractions for designing and analyzing computational tasks has proved to be significantly advantageous in modern computing. In the context of predicate detection, and runtime verification, the problem of abstraction can be viewed as the problem of taking a distributed computation as input and outputting a smaller distributed computation that abstracts out parts that are not relevant to the predicate under consideration. The abstract computation may be exponentially smaller than the original computation resulting in significant savings in predicate detection time.

Computation slicing is an abstraction technique for efficiently finding all global states, of a distributed computation, that satisfy a given global predicate, without explicitly enumerating all such global states [5]. The slice of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states. As an illustration, consider the computation shown in Fig. 1(a). The computation consists of three processes $P_1$, $P_2$, and $P_3$ hosting integer variables $x_1$, $x_2$, and $x_3$, respectively. An event, represented by a circle is labeled with the value of the variable immediately after the event is executed.

Suppose we want to determine whether the property (or the predicate) $(x_1 * x_2 + x_3 < 5) \land (x_1 \geq 1) \land (x_3 \leq 3)$ ever holds in the computation. In other words, does there exist a global state of the computation that satisfies the predicate? The predicate could represent the violation of an invariant. Without computation slicing, we are forced to examine all global states of the computation, twenty-eight in total, to ascertain whether some global state satisfies the predicate. Alternatively, we can compute a slice of the computation automatically with respect to the predicate $(x_1 \geq 1) \land (x_3 \leq 3)$ as shown in Fig. 1(b). We can now restrict our search to the global states of the slice, which are only six in number, namely:

- $\{a, e, f, u, v\}$
- $\{a, e, f, u, v, b\}$
- $\{a, e, f, u, v, w, g\}$

and $\{a, e, f, u, v, b, w\}$ and $\{a, e, f, u, v, w, g\}$.

The slice has much fewer global states than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

In this paper, we focus on abstracting distributed computations with respect to regular predicates (defined in Sec. II). The family of regular predicates contains many useful predicates that are often used for runtime verification in distributed systems. Some such predicates are:

Conjunctive Predicates: Global predicates which are conjunctions of local predicates. For example, predicates of the form,
Fig. 1: A Computation, and its slice with respect to predicate \( (x_1 \geq 1) \land (x_3 \leq 3) \)

\[ B = (l_1 \geq x_1 \geq u_1) \land (l_2 \geq x_2 \geq u_2) \land \ldots \land (l_n \geq x_n \geq u_n), \]

where \( x_i \) is the local variable on process \( P_i \), and \( l_i, u_i \) are constants, are conjunctive predicates. Some useful verification predicates that are in conjunctive form are: detecting mutual exclusion violation in pairwise manner, pairwise data-race detection, detecting if each process has executed some instruction, etc.

**Monotonic Channel Predicates** [7]: Some examples are: all messages have been delivered (or all channels are empty), at least \( k \) messages have been sent/received, there are at most \( k \) messages in transit between two processes, the leader has sent all “prepare to commit” messages, etc.

Centralized offline [8] and online [9] algorithms for slicing based predicate detection have been presented previously. In this paper, we present the first distributed online slicing algorithm for regular predicates in distributed systems.

### A. Challenges and Contributions

Computing the slice of a computation with respect to a general predicate is a NP-Hard problem in general [8]. Many classes of predicates have been identified for which the slice can be computed efficiently in polynomial time (e.g., regular predicates, co-regular predicates, linear predicates, relational predicates, stable predicates) [8], [5], [9], [10]. However, the existing slicing algorithms are centralized in nature. The slice is computed by a single slicer process that examines every relevant event in the computation. The centralized algorithms may be offline, where all events are known a priori, or online, where the slice is updated incrementally with the arrival of every new relevant event. For systems with large number of processes, such centralized algorithms require a simple process to perform high number of computations, and to store very large data. In comparison, a distributed online algorithm significantly reduces the per-process costs for both computation and storage. Additionally, for predicate detection, the centralized online algorithm requires at least one message to the slicer process for every relevant event in the computation, resulting in a bottleneck at the slicer process.

A method of devising a distributed algorithm from a centralized algorithm is to decompose the centralized execution steps into multiple steps to be executed by each process independently. However, for performing online abstraction using computation slicing, such an incremental modification is inefficient as direct decomposition of the steps of the centralized online algorithm requires that each process sends its local state information to all the other processes whenever the local state (or state interval) is updated. In addition, a simple decomposition leads to a distributed algorithm that wastes significant computational time as multiple processes may end up visiting (and enumerating) the same global state. Thus, the task of devising an efficient distributed algorithm for slicing is non-trivial. In this paper, we propose a distributed algorithm that exploits not only the nature of the predicates, but also the collective knowledge across processes. The optimized version of our algorithm reduces the required storage per slicing process, and computational workload per slicing process by \( O(n) \). An experimental evaluation that compares the distributed approach of this paper with the centralized approach can be found in the extended technical report [11].

### B. Applications

Our algorithm is useful for global predicate detection. Suppose the predicate \( B \) is of the form \( B_1 \land B_2 \), where \( B_1 \) is regular but \( B_2 \) is not. We can use our algorithm to slice with respect to \( B_1 \) to reduce the time and space complexity of the global predicate detection. Instead of searching for the global state that satisfies \( B \) in the original computation, with the distributed algorithm we can search the global states in the slice for \( B_1 \). For example, the Cooper-Marzullo algorithm traverses the lattice of global states in an online manner [1], which can be quite expensive. By running our algorithm together with Cooper-Marzullo algorithm, the space and time complexity of predicate detection is reduced significantly (possibly exponentially) for predicates in the above mentioned form.

Our algorithm is also useful for recovery of distributed programs based on checkpointing. For fault-tolerance, we may want to restore a distributed computation to a checkpoint which satisfies the required properties such as “all channels are empty”, and “all processes are in some states that have been saved on storage”. If we compute the slice of the computation in an online fashion, then on a fault, processes can restore the global state that corresponds to the maximum of the last vector of the slice at each surviving process. This global state is consistent as well as recoverable from the storage.

### II. BACKGROUND: REGULAR PREDICATES AND SLICING

#### A. Model

We assume a loosely coupled asynchronous message passing system, consisting of \( n \) reliable processes (that do not fail), denoted by \( \{P_1, P_2, \ldots, P_n\} \), without any shared memory or global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event; and every message activity (send or receive) is also represented by a new event. We assume that the computation being analyzed does not deadlock.

A distributed computation is modeled as a partial order on a set of events [12], given by Lamport’s happened-before (\( \rightarrow \)) relation [12]. We use \( (E, \rightarrow) \) to denote the distributed computation on a set of events \( E \). Mattern [13] and Fidge [14] proposed vector clocks, an approach for time-stamping events in a computation such that the happened-before relation can be tracked. If \( V \) denotes the vector clock for an event
e in a distributed computation, then for any event f in the computation: \( e \rightarrow f \leftrightarrow e.V < f.V \). For any pair of events \( e \) and \( f \) such that \( e \not\rightarrow f \land f \not\rightarrow e \), \( e \) and \( f \) are said to be concurrent, and this relation is denoted by \( e||f \). Fig. 2(a) shows a sample distributed computation, and its corresponding vector clock representation is presented in Fig. 2(b).

We now present some required concepts:

**Definition 1 (Consistent Cut).** Given a distributed computation \((E, \rightarrow)\), a subset of events \( C \subseteq E \) is said to form a consistent cut if \( C \) contains an event \( e \) only if it contains all events that happened-before \( e \). Formally, \( e \in C \land f \rightarrow e \implies f \in C \).

The concept of a consistent cut (or, a consistent global state) is identical to that of a down-set (or order-ideal) used in lattice theory [15]. Intuitively, a consistent cut captures the notion of a global state of the system at some point during its execution [16].

Consider the computation shown in Fig 2(a). The subset of events \( \{a, b, e\} \) forms a consistent cut, whereas the subset \( \{a, b, f\} \) does not; because \( b \rightarrow f \) (\( b \) happened-before \( f \)) but \( b \) is not included in the subset. A consistent cut can also be represented with a vector clock notation. For any consistent cut \( C \), its vector clock \( C.V \) can be computed as \( C.V = \text{component-wise-max\{e.V | \text{event } e \in C\} \} \), where \( e.V \) denotes the vector clock of event \( e \). For this paper, we use a shortened notation for a cut of the computation. A cut \( C \) is denoted by the latest events on each process. Thus, \( \{a, b, e\} \) is denoted by \( [b, e] \) and \( \{a, e, f\} \) is represented as \( [a, f] \).

Table I shows all the consistent global states/cuts and their corresponding vector clock values for the computation in Fig. 2. A visual representation of the lattice of consistent cuts for Fig. 2 can be found in Fig. 3. We now present additional notions from lattice theory that are key to our approach.

**Definition 2 (Join).** A join of a pair of global states is defined as the set-union of the set of events in the states.

**Definition 3 (Meet).** A meet of a pair of global states is defined as the set-intersection of the set of events in the states.

For two global states \( C_1 \) and \( C_2 \), their join is denoted with \( C_1 \sqcup C_2 \), whereas \( C_1 \sqcap C_2 \) denotes their meet.

**Theorem 1.** [15], [13] Let \( C(E) \) denote the set of all consistent cuts of a computation \((E, \rightarrow)\). \( C(E) \) forms a lattice under the relation \( \sqsubseteq \).

A global predicate (or simply a predicate) is a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated on the state resulting after executing all events in the cut. A global predicate is local if it depends only on variables of a single process. If a predicate \( B \) evaluates to true for a consistent cut \( C \), we say that “\( C \) satisfies \( B \)” and denote it by \( C_B \).

**Definition 4 (Linearity Property of Predicates).** A predicate \( B \) is said to have the linearity property, if for any consistent cut \( C \), which does not satisfy predicate \( B \), there exists a process \( P_i \) such that a cut that satisfies \( B \) can never be reached from \( C \) without advancing along \( P_i \).

Predicates that have the linearity property are called linear predicates.

For example, consider the cut \([b, e]\) of the computation shown in Fig. 2(a). The cut does not satisfy the predicate “all channels are empty”, and for the given cut, progress must be made on \( P_2 \) to reach the cut \([b, f]\) which satisfies the predicate.

The process \( P_1 \) in the above definition is called a forbidden process. For a computation involving \( n \) processes, given a consistent cut that does not satisfy the predicate, \( P_i \) can be found in \( O(n) \) time for most linear predicates used in practice.

To find a forbidden process given a consistent cut, a process first checks if the cut needs to be advanced on itself; if not it checks the states in the total order defined using process ids, and picks the first process whose state makes the predicate false on the cut. The set of linear predicates has a subset, the set of regular predicates, that exhibits a stronger property.

**Definition 5 (Regular Predicates).** A predicate is called regular if for any two consistent cuts \( C \) and \( D \) that satisfy the predicate, the consistent cuts given by \((C \sqcap D) \) (the meet of \( C \) and \( D \)) and \((C \sqcup D) \) (the join of \( C \) and \( D \)) also satisfy the predicate.

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Cut Vec. Clock</th>
<th>Pred. Eval.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[a]</td>
<td>[0, 0]</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>[b]</td>
<td>[1, 0]</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>[c]</td>
<td>[2, 0]</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>[d]</td>
<td>[3, 0]</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>[e]</td>
<td>[0, 1]</td>
<td>True</td>
</tr>
<tr>
<td>6</td>
<td>[a,e]</td>
<td>[1, 1]</td>
<td>True</td>
</tr>
<tr>
<td>7</td>
<td>[b,e]</td>
<td>[2, 1]</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>[c,e]</td>
<td>[3, 1]</td>
<td>False</td>
</tr>
<tr>
<td>9</td>
<td>[b,g]</td>
<td>[2, 2]</td>
<td>True</td>
</tr>
<tr>
<td>10</td>
<td>[c,g]</td>
<td>[3, 2]</td>
<td>False</td>
</tr>
<tr>
<td>11</td>
<td>[c,e]</td>
<td>[3, 3]</td>
<td>False</td>
</tr>
</tbody>
</table>

**TABLE I:** Consistent Global States of Fig. 2 and Predicate Evaluation for \( B = \text{"all channels empty"} \)
Examples of regular predicates include local predicates (e.g., \( x \leq 4 \)), conjunction of local predicates (e.g., \((x \leq 4) \land (y \geq 2)\)) where \(x\) and \(y\) are variables on different processes) and monotonic channel predicates (e.g., there are at most \(k\) messages in transit from \(P_i\) to \(P_j\)) [8]. Table I indicates whether or not the predicate “all channels empty” is satisfied by each of the consistent global cuts of the computation in Fig. 2. To use computation slicing for detecting regular predicates, we first need to capture the notion of join-irreducible elements for the lattice of consistent cuts.

**Definition 6** (Join-Irreducible Element). Let \(L\) be a lattice. An element \(x \in L\) is join-irreducible if

1) \(x\) is not the smallest element of \(L\)

2) \(\forall a, b \in L : (x = a \lor b) \implies (x = a) \lor (x = b)\).

Intuitively, a join-irreducible element of a lattice is one that cannot be represented as the join of two distinct elements of the lattice, both different from itself. For the lattice of consistent cuts of a distributed computation, the join-irreducible elements correspond to consistent cuts that cannot be reached by joins (set-union) of two or more consistent cuts. For the computation of Fig. 2, the join-irreducible consistent cuts are: \([a], [b], [c], [e], [b, f], [b, g]\). Fig. 3 shows the join-irreducible consistent cuts of the sub-lattice induced by predicate “all channels empty” for computation of Fig. 2.

![Fig. 3: Lattice of Consistent Cuts for Fig.2(a)](image)

**B. Computation Slice**

A computation slice of a computation with respect to a predicate \(B\) is a concise representation of all the consistent cuts of the computation that satisfy the predicate \(B\). When the predicate \(B\) is regular, the set of consistent cuts satisfying \(B\), \(L_B\), forms a sublattice of \(L\), that is the lattice of all consistent cuts of the computation \((E, \rightarrow)\). \(L_B\) can equivalently be represented using its join-irreducible elements [15]. Intuitively, join-irreducible elements form the basis of the lattice. The lattice can be generated by taking joins of its basis elements. Let \(L_B\) be the set of all join-irreducible elements of \(L_B\). Let \(J_B(e)\) denote the least consistent cut that includes an event \(e\) and satisfies predicate \(B\). Then, it can be shown [17] that

\[
J_B = \{ J_B(e) | e \in E \}
\]

The \(J_B(e)\) values, in vector clock notation, for each event of the computation in Fig. 2 are:

\[
J_B(a) = [1, 0], J_B(b) = [2, 2], J_B(c) = [3, 2], J_B(e) = [0, 1], J_B(f) = [2, 2], J_B(g) = [2, 3].
\]

We can now define a computation slice formally.

**Definition 7.** Let \((E, \rightarrow)\) be a computation and \(B\) be a regular predicate. The slice of the computation with respect to \(B\) is defined as \(\{J_B, \subseteq\}\).

**Note:** It is important to observe that \(J_B(e)\) does not necessarily exist because there may not be any consistent cut that includes \(e\) and satisfies \(B\). Also, multiple events may have the same \(J_B(e)\).

For the computation shown in Fig. 2(a), Fig. 2(c) presents a visual representation of the slice.

A centralized online algorithm to compute \(J_B\) was proposed in [5]. In the online version of this centralized algorithm, a pre-identified process, called slicer process, plays the role of the slice computing process. All the processes in the system send their event and local state values whenever their local states change. The slicer process maintains a queue of events for each process in the system, and on receiving the data from a process adds the event to the relevant queue. In addition, the slicer process also keeps a map of events and corresponding local states for each process in the system. For each received event, the slicer appends the event and local state mapping to the respective map. For every event \(e\) it receives, the slicer computes \(J_B(e)\) using the linearity property.

The centralized approach suffers from the drawback of causing a heavy load of messages as well as computation on just one process, namely the slicer process. Thus, for any large distributed computation, this approach would not scale well. To address this issue, we propose a distributed algorithm that significantly reduces the computational, as well as the message load on any process.

**III. A DISTRIBUTED ONLINE ALGORITHM FOR SLICING**

In this section, we present the key ideas and routines for distributed online algorithm for computing the slice. The required optimizations that tackle the challenges listed in Section I-A are discussed later. In our algorithm, we have \(n\) slicer processes, \(S_1, S_2, \ldots, S_n\), one for every application process. All slicer processes cooperate to compute the task of slicing \((E, \rightarrow)\). Let \(E\) be partitioned into \(n\) sets \(E_i\) such that \(E_i\) is the set of events that occurred in \(P_i\). In our algorithm, \(S_i\) computes

\[
J_i(B) = \{ J_B(e) | e \in E_i \}
\]

Observe that by the definition of join-irreducible consistent cut, \(e \rightarrow f\) implies \(J_B(e) \subseteq J_B(f)\). Since all events in a process are totally ordered, the set of consistent cuts generated by any \(S_i\) are also totally ordered.

**Note:** In this paper, the symbol \(\rightarrow\) indicates a happened-before relation; whereas the symbol \(\leftarrow\) in the pseudo-code denotes assignment operation.

Algorithm 1 presents the distributed algorithm for online slicing with respect to a regular predicate \(B\). Each slicer process has a token assigned to it that goes around in the system. Other slicer processes cooperate in maintaining and processing the token. The goal of the token for the slicer process \(S_i\) is to compute \(J_B(e)\) for all events \(e \in E_i\). Whenever the token
has computed \( J_B(e) \) it returns to its original process, reports \( J_B(e) \) and starts computing \( J_B(succ(e)) \), \( succ(e) \) being the immediate successor of event \( e \). The token \( T_i \) carries with it the following data:

- **pid**: Process id of the slicer process to which it belongs.
- **event**: Details of event \( e \), specifically the event id and event’s vector clock, at \( P_i \), for which this token is computing \( J_B(e) \). The identifier for event \( e \) is the tuple \(<pid, eid>\) that identifies each event in the computation uniquely.
- **gcut**: The vector clock corresponding to the cut which is under consideration (a candidate for \( J_B(e) \)).
- **depend**: Dependency vector for events in \( gcut \). The dependency vector is updated each time the information of an event is added to the token (steps explained later), and is used to decide whether or not some cut being considered is consistent. On any token, its vector \( gcut \) is a consistent global state iff for all \( i \), \( depend[i] \leq gcut[i] \).
- **gstate**: Vector representation of global state corresponding to vector \( gcut \). It is sufficient to keep only the states relevant to the predicate \( B \).
- **eval**: Evaluation of \( B \) on \( gstate \). The evaluation is either true or false; in our notation we use the values: \{\text{predtrue}, \text{predfalse}\}.
- **target**: A pointer to the unique event in the computation for which a token has to wait. The event need not belong to the local process.

A token waits at a slicer process \( P_i \) under three specific conditions:

- (C1) The token is for process \( S_i \) and it has computed \( J_B(pred(e)) \), \( pred(e) \) being the immediate predecessor event of \( e \), and is waiting for the arrival of \( e \).
- (C2) The token is for process \( S_i \) and it is computing \( J_B(f) \), where \( f \) is an event on \( P_i \) prior to \( e \). The computation of \( J_B(f) \) requires the token to advance along process \( P_i \).
- (C3) The token is for process \( S_j \) such that \( j \neq i \), and it is computing \( J_B(f) \) which requires the token to advance along process \( P_j \).

On occurrence of each relevant event \( e \in E_i \), the computation process \( P_i \) performs a local enqueue to slicer \( S_i \), with the details of this event. Note that \( P_i \) and its slicer \( S_i \) are modeled as two threads on the same process, and therefore the local enqueque is simply an insertion into the queue (that is shared between the threads on the same process) of the slicer. The message contains the details of event \( e \), i.e. the event identifier \(<pid, eid>\), the corresponding vector clock \( e.V \), and \( P_i \)’s local state \( localstate_e \) corresponding to \( e \). The steps of the presented routines are explained below:

**ReceiveEvent** (Lines 1-8): On receiving the details of event \( e \) from \( P_i \), \( S_i \) adds them in the mapping of \( P_i \)’s local states \( procsstates \) (line 2). It then iterates over all the waiting tokens, and checks their target. For each token that has \( e \) as the target (required event to make progress), \( S_i \) updates the state of the token, and then processes it.

**AddEventToToken** (Lines 9-15): To update the state of some token \( t \) on \( S_i \), we advance the candidate cut to include the new event by setting \( t.gcut[i] \) to the id of event \( e \). If \( S_i \) is the parent process of the token \( (T_i) \), then the \( t.event \) pointer is updated to indicate the event id for which token is computing the join-irreducible cut that satisfies the predicate. The causal-dependency is updated at line 15, which is required for checking whether or not the cut is consistent.

**ProcessToken** (Lines 16-22): To process any token, \( S_i \) first checks that the global state in the token is consistent (line 17) and at least beyond the global states that were earlier evaluated to be false. For \( t \)’s evaluation of a global cut \( t.gcut \) to be consistent, \( t.gcut \) must be at least \( t.depend \). This is verified by checking the component-wise values in both these

---

**Algorithm 1: Algorithm at \( S_i \)**

1. ReceiveEvent (Event \( e \), State \( \text{localstate}_e \))
2. save \(<e.eid, \text{localstate}_e>\) in local state map \(<\text{procsstates}>\)
3. foreach waiting token \( t \) at \( S_i \) do
4. if \((t.target = e)\) then // waiting for event \( e \)
5. AddEventToToken \((e)\)
6. ProcessToken \((t)\)
7. end
8. AddEventToToken (Token \( t \), Event \( e \))
9. \( t.gstate[e.pid] \leftarrow \text{procsstate}[e.eid] \)
10. \( t.gcut[e.pid] \leftarrow e.eid \)
11. if \((t.pid = i)\) then //my token: update token’s \( \text{event} \)
12. \( t.event \leftarrow e \)
13. t.depend \leftarrow \max(t.depend, e.V) // set causal dependency
14. ProcessToken (Token \( t \))
15. if \((t.gcut \text{ is inconsistent})\) then
16. // find \( k \) : \( gcut[k] < t.depend[k] \) \*/
17. \( t.target \leftarrow t.gcut[k] + 1 \) // set desired event
18. send \( t \) to \( S_k \)
19. else // \( t.gcut \) is consistent
20. EvaluateToken \((t)\)
21. end
22. EvaluateToken (Token \( t \))
23. if \((B(t.gstate) \text{ then } // B \text{ is true on cut given by } t.gcut \)
24. \( t.eval \leftarrow \text{predtrue} \)
25. send \( t \) to \( S_{i.pid} \)
26. else // \( B \) is false on \( t.gstate \)
27. \( t.eval \leftarrow \text{predfalse} \)
28. \( //\text{ PK } \text{ forbidden process in } t.gstate \text{ for } B \) \*/
29. \( t.target \leftarrow t.gcut[k] + 1 \)
30. send \( t \) to \( S_k \)
31. end
32. ReceiveToken (Token \( t \))
33. if \((t.eval = \text{predtrue}) \land (t.pid = i)\) then //my token, \( B \) true
34. output\((t.pid, t.eid, t.gcut)\)
35. // token waits for the next event \*/
36. \( t.target \leftarrow t.gcut[k] + 1 \)
37. t.waiting \leftarrow \text{true} \)
38. else //either inconsistent cut, or predicate false
39. \( newid \leftarrow t.target \) // id of next event \( t \) requires
40. \( //\text{ required event has happened} \)
41. AddEventToToken \((t,f)\)
42. EvaluateToken \((t)\)
43. end
44. ReceiveStopSignal
45. foreach token \( t \) : \( t.pid \neq i \) do
46. //not my token, send back to parent
47. send \( t \) to \( S_{i.pid} \)
48. end
49. end
50. end
vectors. If some index $k$ is found where $t.\text{depend} > t.\text{gcut}$, the token’s cut is inconsistent, and $t.\text{gcut}$ must be advanced by at least one event on $P_k$, by sending the token to slicer of $P_k$. If the cut is consistent, the predicate is evaluated on the variables stored as part of $t.\text{gstate}$ by calling the EvaluateToken routine.

**EvaluateToken (Lines 22-31):** The cut represented by $t.\text{gstate}$ is evaluated; if the predicate is true, then the token has computed $J_B(e)$ for the event $e = <t.\text{pid}, t.\text{eid}>$. The token is then sent to its parent slicer. If the evaluation of the predicate on the cut is false, the target pointer is updated, at line 29, and the token is sent to the forbidden process on which the token must make progress.

**ReceiveToken (Lines 32-45):** On receiving a token, the slicer checks if the predicate evaluation on the token is true, and the token is owned by the slicer. In such a case, the slicer outputs the cut information, and now uses the token to find $J_B(\text{suc}(e))$, where $\text{suc}(e)$ denotes the event that locally succeeds $e$. This is done by setting the new event id in $t.\text{target}$ at line 35, and then setting the waiting flag (line 36). If the predicate evaluation on the token is false, then the target pointer of the token points to the event required by the token to make progress. $S_i$ looks for such an event (line 39), and if it has been reported to $S_i$ by $P_i$, then adds that event (line 41) to the token and processes it (line 42). In case the desired event has not been reported yet to the slicer process, the token is retained at the process $S_i$, and is kept in the waiting state until the required event arrives. Upon arrival of the required event, its details are added to the token and the token is processed.

**Note:** The notation of $\text{target} \leftarrow t.\text{gcut}[i] + 1$ means that if the $t.\text{gcut}[i]$ holds the event id $<\text{pid}, \text{eid}>$, then the target pointer is set to $<\text{pid}, \text{eid} + 1>$. 

**ReceiveStopSignal (Lines 46-50):** For finite computations, a single token based termination detection algorithm is used in tandem. When termination is detected, a pre-determined slicer sends the ‘stop’ signal to all the slicer processes, including itself. On receiving the ‘stop’ signal, $S_i$ sends all the slicing tokens that do not belong to it back to their parent processes.

Note that the routines of Algorithm 1 require atomic updates and reads on the local queues, as well as on tokens present at $S_i$. In the interest of space we skip presenting the lower level implementation details, that involve common local synchronization techniques.

### A. Example of Algorithm Execution

This example illustrates the algorithm execution steps for one possible run (real time observations) of the computation shown in Fig. 2, with respect to the predicate $B = \text{“all channels empty”}$. The algorithm starts with two slicing processes $S_1$ and $S_2$, each having a token $- T_1$ and $T_2$ respectively. The target pointer for each token $T_i$ is initialized to the event $<i, 1>$. When event $a$ is reported, $S_1$ adds its details to $T_1$, and on its evaluation finds the predicate “all channels empty” to be true, and outputs this information. It then updates $T_1.\text{target}$ pointer and waits for the next event to arrive. Similar steps are performed by $S_2$ on $T_2$ when $e$ is reported.

When $b$ is reported to $S_1$, and $T_1$ is evaluated with the updated information, the predicate is false on the state $[b]$. Given that $b$ is a message send event, it is obvious that for the channel to be empty, the message receive event should also be incorporated. Thus, $S_1$ sends $T_1$ to $S_2$ after setting the target pointer to the first event on $S_2$. On receiving $T_1$, $S_2$ fetches the information of its first event ($e$) and updates $T_1$. The subsequent evaluation still leads to the predicate being false. Thus $S_2$ retains $T_1$ and waits for the next event.

When $f$ is reported, $S_2$ updates both $T_1$ and $T_2$ with $f$’s details. $S_2$’s evaluation on $T_1.\text{gstate}$, represented by $[b, f]$ is true, and as per line 25, $T_1$ is sent back to $S_1$ where the consistent cut $[b, f]$ is output. $T_1$ now waits for the next event. However, after being updated with the details of event $f$, the resulting cut on $T_2$ is inconsistent, as the message-receive information is present but the information regarding the corresponding send event is missing. By using the vector clock values, $T_2$’s target would be set to the id of message-send event $b$. $S_2$ would then send $T_2$ to $S_1$. On receiving $T_2$, $S_1$ finds the required event (looking at $T_2.\text{target}$) and after updating $T_2$ with its details, evaluates the token. The predicate is true on $T_2.\text{gstate}$ now, and $T_2$ is sent back to $S_2$. On receiving $T_2$, $S_2$ sends the consistent cut $[b, f]$, and waits for the next event. On receiving details of event $e$, and adding them to the waiting token $T_1$, the predicate is found to be true again on $T_1$, and $S_1$ outputs $[c, f]$. Similarly on receiving $g$, $S_2$ performs similar steps and outputs $[b, g]$. Note that the consistent cuts $[a, b]$ and $[c, g]$, both of which satisfy the predicate are not enumerated as they are not join-irreducible, and can be constructed by the unions of $[a, b]$ and $[c, f]$, $[b, g]$ respectively.

### B. Proof of Correctness

We now prove the correctness and termination of the distributed algorithm of Algorithm 1 for finite computations. The correctness argument can be easily extended to infinite computations.

**Lemma 1.** The algorithm presented in Algorithm 1 does not deadlock.

**Proof:** The algorithm involves $n$ tokens, and none of the tokens wait for any other token to complete any task. With non-lossy channels, and no failing processes, the tokens are never lost. The progress of any token depends on the target event, and as per lines 4-7, whenever an event is reported to a slicer, it always updates the tokens with their target being this event. Thus, the algorithm can not lead to deadlocks.

**Lemma 2.** If a token $T_i$ is evaluating $J_B(e)$ for $e \in E_i$, assuming $J_B(e)$ exists, and if $T_i.\text{gcut} < J_B(e)$, then $T_i.\text{gcut}$ would be advanced in finite time.

**Proof:** If during the computation of $J_B(e)$, at any instance $T_i.\text{gcut} < J_B(e)$, then there are two possibilities for $\text{gcut}$:

(a) $\text{gcut}$ is consistent: This means that the evaluation of predicate $B$ on $\text{gcut}$ must be false, as by definition $J_B(e)$ is the least consistent cut that satisfies $B$ and includes $e$. In this case, by line 29 and subsequent steps, the token would be
forced to advance on some process.
(b) gcrit is inconsistent: The token is advanced on some process by execution of lines 17-18. ■

Lemma 3. While evaluating $J_B(e)$ for event $e \in E_i$ on token $T_i$, if $T_i, gcrit < J_B(e)$ currently and $J_B(e)$ exists then the algorithm eventually outputs $J_B(e)$.  

Proof: By Lemma 2, the global cut of $T_i$ would be advanced in finite time. Given that $J_B(e)$ exists, we know that by the linearity property, there must exist a process on which $T_i$ should progress its gcrit and gstate vectors in order to reach the $J_B(e)$; lines 29-31 ensure that this forbidden process is found and $T_i$ sent to this process. By the previous Lemma, the cut on the $T_i$ would be advanced until it matches $J_B(e)$. By line 33 of the algorithm, whenever $J_B(e)$ is reached, it would be output. ■

Lemma 4. For any token $T_i$, the algorithm never advances $T_i, gcrit$ vector beyond $J_B(e)$ on any process, when searching $J_B(e)$ for $e \in E_i$.

Proof: The search for $J_B(e)$ starts with either an empty global state vector, or from the global state that is at least $J_B(pred(e))$, where pred(e) is the immediate predecessor event of e on $S_i$. Thus, till $J_B(e)$ is reached, the global cut under consideration is always less than $J_B(e)$. From the linearity property of advancing on the forbidden process, and Lemma 2, the cut would be advanced in finite time. Whenever the cut reaches $J_B(e)$, it would be output as per Lemma 3 and the token would be sent back to its parent slicer, to either begin the search for succ(e) or to wait for succ(e) to arrive (succ(e) being the immediate successor of e). Thus, $T_i, gcrit$ would never advance beyond $J_B(e)$ on any process when searching for $J_B(e)$ for any event e. ■

Lemma 5. If token $T_i$ is currently not at $S_i$, then $T_i$ would return to $S_i$ in finite time.

Proof: Assume $T_i$ is currently at $S_j$ ($j \neq i$). $S_j$ would advance $T_i, gcrit$ in finite time as per Lemma 2. With no deadlocks (Lemma 1), and by the results of Lemma 3 and Theorem 4, we are guaranteed that if $J_B(T_i, event)$ exists then within a finite time, $T_i, gcrit$ vector would be advanced to $J_B(T_i, event)$ and $T_i$ would be sent back to $S_i$. If $J_B(T_i, event)$ does not exist then at least one slicer process $S_k$ would run out of all its events while attempting to advance on $T_i, gcrit$. In such a case, knowing that there are no more events to process, $S_k$ would send $T_i$ back to $S_i$ (lines 46-50). ■

Theorem 2. (Termination): For a finite computation, the algorithm terminates in finite time.

Proof: We first prove that for any event $e \in E_i$, computation of finding $J_B(e)$ with token $T_i$ takes finite time. By Lemma 2, $T_i$ always advances in finite time while computing $J_B(e)$. If $J_B(e)$ exists, then based on this observation within a finite time the token $T_i$ would advance its gcrit to $J_B(e)$, if it exists. By Lemma 3, the algorithm would output this cut, thus finishing the $J_B(e)$ search and as per Theorem 4 would not advance any further for $J_B(e)$ computation. Thus, if $J_B(e)$ exists then it would be output in finite time. By Lemma 5 the token would be returned to its parent process and the $J_B(e)$ computation for $e \in E_i$ would finish in finite time.

If $J_B(e)$ does not exist, then as we argued in Lemma 5 some slicer would run out of events to process in the finite computation, and thus return the token to $S_i$, which would result in search for $J_B(e)$ computation to terminate. As each of these steps is also guaranteed to finish in finite time as per above Lemmas, we conclude that $J_B(e)$ computation for $e \in E_i$ finishes in finite time.

Applying this result to all the events in $E$ leads to the desired result of termination in finite time. ■

Theorem 3. The algorithm outputs all the elements of $J_B$. 

Proof: Whenever any event $e \in E$ occurs, it is reported by some process $P_i$ on which it occurs, to the corresponding slicer process $S_i$. Thus $e$ can be represented as $e \in E_i$. If at the time $e$ is reported to $S_i$, $T_i$ is held by $S_i$ then by Lemmas 2 and 3, it is guaranteed that the algorithm would output $J_B(e)$. If $S_i$ does not hold the token $T_i$ when $e$ is reported to it, then by Lemma 5, $T_i$ would arrive on $S_i$ within finite time. If $S_i$ has any other events in its processing queue before $e$, then as per Theorem 2, $S_i$ would finish those computations in finite time too. Thus, within a finite time, the computation for finding $J_B(e)$ with $T_i$ would eventually be started by $S_i$. Once this computation is started, the results of Lemmas 2 and 3 can be applied again to guarantee that the algorithm would output $J_B(e)$, if it exists.

Repeatedly applying this result to all the events in $E$, we are guaranteed that the algorithm would output $J_B(e)$ for every event $e \in E$. Thus the algorithm outputs all the join-irreducible global states that satisfy predicate $B$. ■

Theorem 4. The algorithm only outputs join-irreducible global states that satisfy predicate $B$. 

Proof: By Lemma 4, while performing computations for $e \in E_i$ on token $T_i$, the algorithm would not advance on token $T_i$ beyond $J_B(e)$. Since only token $T_i$ is responsible for computing $J_B(e)$ for all the events $e \in E_i$, the algorithm would not advance beyond $J_B(e)$ on any token. In order to output a global state that is not join-irreducible we must advance the cut of at least one token beyond a least global state that satisfies $B$. The result follows from the above assertions. ■

Theorem 2 guarantees termination, and correctness follows from Theorems 3, and 4.

IV. OPTIMIZATIONS

The distributed algorithm presented in the previous section is not optimized to avoid redundant token messages, as well as duplicate computations. Whenever a slicer process $S_i$ needs to send any token to another process $S_k$, it should first check if it currently holds the token $T_k$, and if the desired information is present in $T_k$. If the information is available, the token $T_i$ can be updated with the information without being sent to $S_k$; and only if the details of required event are not available locally, the token is sent to $S_k$. These steps are captured in the procedure SendIfNeeded shown in Algorithm 2.

There are additional optimizations that significantly reduce the number of token messages. It is easy to observe
Algorithm 2: SendIfNeeded at Si

```plaintext
1 SendIfNeeded (Token t, int k) 
2     /* k: id of the slicer process to which t should be sent */
3     while (k ≠ i) ∧ (have_tokenk) do
4         if (t.target = tokenk.event) then /tokenk has info of t’s 
5             /target event
6                 t.gcut[k] ← tokenk.gcut[k]
7                 t.depend[k] ← tokenk.depend[k]
8                 t.state[k] ← tokenk.state[k]
9                 if (t.gcut is inconsistent) then /still inconsistent
10                    /* find j : t.gcut[j] < t.depend[j] */
11                        t.target ← t.gcut[j] + 1
12                        k ← j /set k for while condition
13                    else if t.gcut is consistent now, evaluate
14                        EvaluateToken (t)
15                        break
16     end
17     else desired token or event info not present */
18     if (t.target.pid ≠ i) then
19         /* target event on some other process */
20             send t to Sj
21     end
```

In the optimized algorithm an additional variable, currentE - at each slicer process Si, is used as a local pointer to keep track of the event e for which Si is currently computing J_B(e). Every token T_i also stores this information (using pointer token.event), however the token T_i is not always present on Si, and thus currentE information is needed. By keeping currentE updated, even in absence of token T_i, the slicer process Si can delay the progress of other tokens whenever it suspects that these tokens may undergo the same J_B(e) computation that is being considered by T_i. For stopping possibly duplicate computations, a flag, called stalled, is maintained in each token. By setting the stalled flag on any token, a slicer removes the token from the set of waiting tokens; and no updates are performed on tokens that are in the stalled state. The optimized algorithm also makes use of the type information of events, for identifying if an event is a send (or a receive) of a message. The modifications injected in the algorithm routines (we only discuss routines that change) are briefly explained below:

AddEventToToken: While adding an event e ∈ E_i to token T_i the slicer Si also updates the currentE pointer to store the details of e. In addition, Si checks if event e is a message receipt event. If yes, Si stalls its token T_i, unless it receives the required information (described shortly ahead) from the J_B(e) computation of the corresponding sender event f. For ensuring global process symmetry breaking (based on process-id i) is used before stalling any token. This step is to incorporate Lemma 8 in speculative manner. Whenever the corresponding slicer process of message send event f finishes computing the J_B(f), it informs Si about the computed cut. Si on receiving this cut, checks if e belongs to J_B(f) and thus J_B(e) computation is not needed; otherwise Si restarts the computation for J_B(e).

ReceiveToken: On receiving a token T_k, where k ≠ i, additional checks are performed to incorporate Lemma 7 in speculative manner. If the currently ongoing computation on token T_k is causally dependent on the computation on token T_i, then T_k is processed. However, if T_k.event and currentE are not causally related, i.e. they are concurrent, then T_k is stalled if the computation of J_B(T_k.event) wants to progress beyond the current ongoing computation on T_i. To guarantee deadlock freedom (because of stalling) symmetry breaking is performed using process-ids in favor of the token/process with larger process-id. This guarantees that whenever the cuts of two concurrent events are same, only one of the tokens (with the smaller process id) finishes computing the cut, and thus duplicate computations are not performed.

Output: Whenever Si finishes computing the J_B(e) for the event e ∈ E_i, it tries to update each stalled token, either present locally or at some other slicer process, that was speculatively stalled to avoid computing the same cut. The notification to other slicer processes is performed by sending the details of the cut to them. If the stalled tokens infer, that their cuts (if computed) and T_i.gcut would be same by the application of Lemmas 7 and 8, then they copy the received cut details and move on to the next events on their respective processes. The forwarding of cuts is performed in a cascading manner. Thus a slicer forwards a received cut, if and only if there has been a message activity that will cause other slicer
The details of these steps, including the pseudo-code, as well as the proof of correctness of the optimized algorithm can be found in the extended technical report [11].

A. Example of Optimized Algorithm Execution

We revisit the example presented in section III-A for the distributed algorithm run, in order to show the difference in execution for the optimized algorithm. When \( f \) is reported to \( S_2 \), the earlier version of the algorithm has to update the token \( T_2 \), and send it to \( S_1 \) in order to make the cut on \( T_2 \) consistent. The optimized algorithm determines that \( f \) being a message-receive event, the \( J_B(f) \) computation should not be started until the corresponding message-send event’s computation is reported to \( S_2 \). Thus \( T_2 \) would be kept in stalled state, until \( T_1 \) finishes the \( J_B(b) \) computation. When \( J_B(b) \) computation on \( T_1 \) is finished, with \( J_B(b) = [b, f] \), then \( S_1 \) would send the information of \( b \)'s join-irreducible cut to \( S_2 \). On receiving the cut details, \( S_2 \) would try to update its stalled token \( T_2 \), and it would infer (using Lemmas 7 and 8) that \( J_B(f) = J_B(b) \). Thus, it would just copy the details of the cut as the result for \( J_B(f) \), and move on to computing \( J_B(g) \).

B. Analysis

Each token \( T_i \) processes every event \( e \in E_i \) once for computing its \( J_B(e) \). If there are \(|E| \) events in the system, then in the worst case \( T_i \) does \( O(|n|E|) \) work, because it takes \( O(n) \) to process one event. We are assuming here that evaluation of \( B \) takes \( O(n) \) time given a global state. There are \( n \) tokens in the system, hence the total work performed is \( O(n^2|E|) \). Since there are \( n \) slicing processes and \( n \) tokens, the average work performed is \( O(n|E|) \) per process. In comparison, the centralized algorithm (either online or offline) requires the slicer process to perform \( O(n^2|E|) \) work.

Let \(|S|\) be the maximum number of bits required to represent a local state of a process. The actual value of \(|S|\) is subject to the predicate under consideration, as the resulting number/type of the variables to capture the necessary information for predicate detection depends on the predicate. The centralized online algorithm requires \( O(|E||S|) \) space in the worst case; however it is important to notice that all of this space is required on a single (central slicer) process. For a large computation, this space requirement can be limiting. The distributed algorithm proposed above only consumes \( O(|E_i||S|) \) space per slicer. Thus, we have a reduction of \( O(n) \) in per slicer space consumption.

The token can move at most once per event. Hence, in the worst case the message complexity is \( O(|E|) \) per token. Therefore, the message complexity of the distributed algorithm presented here is \( O(n|E|) \) total for all tokens. The message complexity of the centralized online slicing algorithm is \( O(|E|) \) because all the event details are sent to one (central) slicing process. However, for conjunctive predicates, it can be observed that the message complexity of the optimized version of the distributed algorithm is also \( O(|E|) \). With speculative stalling of tokens, only unique join-irreducible cuts are computed. This means that for conjunctive predicates, a token only leaves (and returns to) \( S_i \), \( O(|E_i|) \) times. As there are \( n \) tokens, the overall message complexity of the optimized version for conjunctive predicates is \( O(|E|) \).

V. RELATED WORK

The distributed algorithm presented in this paper constructs the slice of a distributed computation with respect to a regular state based predicate. The constructed slice can then be used to determine if some consistent cut of the computation satisfies the predicate. This is referred to as the problem of detecting a predicate under possibly modality [1]. In [1], a predicate is detected by exploring the complete lattice of consistent cuts in a breadth first manner. Alagar et al. [6] use a depth first traversal of the computation lattice to reduce space complexity. The algorithms in [1] and [6] can handle arbitrary predicates, but in general have exponential time complexity. In contrast, the slicing algorithm presented in this paper for a regular predicate has polynomial time complexity.

In this paper we assume a static distributed system. Predicate detection algorithms have been proposed for dynamic systems (e.g. [18], [19], [20], [21], [22]), where processes may leave or join. However, these algorithms detect restricted classes of predicates like stable predicates and conjunctive local predicates, which are less general than regular predicates. In computation slicing, we analyze a single trace (or execution) of a distributed program for any violation of the program’s specification. Model checking (cf. [23]) is a formal verification technique that involves determining if all traces of a program meets its specification. Model checking algorithms conduct reachability analysis on the state space graph, and have a time complexity that is exponential in number of processes.

Partial order methods (cf. [24]) aim to alleviate the state-explosion problem by minimizing the state space for predicate detection. This is done by exploring only a subset of the interleavings of concurrent events in a computation, instead of all possible interleavings. However, predicate detection algorithms based on partial order methods still have exponential time complexity, in the worst case. In this paper, the focus is on generating the slice with respect to a predicate. Partial order methods such as [25] can be used in conjunction with slicing to explore the state space of a slice in a more efficient manner [26].

The work presented in this paper is related to runtime verification (cf. [27]), which involves analyzing a run of a program to detect violations of a given correctness property. The input program is instrumented and the trace resulting from its execution is examined by a monitor that verifies its correctness. Some examples of runtime verification tools are Temporal Rover [28], Java-MaC [29], JPaX [30], JMPaX [31], and jPredictor [32]. The Temporal Rover, Java-Mac and JPaX tools model the execution trace as a total order of of events, which is then examined for violations. In the JMPaX and jPredictor tools the trace is also modeled as a partial order of events. However, these tools generate states not observed in the current trace, to predict errors that may occur in other runs, thereby increasing the size of the computation lattice. Chen et al. [33] note that computation slicing can be used to make tools like jPredictor more efficient by removing redundant states from the lattice. All of these tools are centralized in nature, where the events are collected at a central monitoring process. Sen et al. [34] present a decentralized algorithm that monitors a program’s execution, but can only detect a subset of safety properties. The distributed algorithm presented by Bauer et
VI. CONCLUSION

In this paper, we presented a distributed online algorithm for performing computation slicing, a technique to abstract the computation with respect to a regular predicate. The resulting abstraction (slice) is usually much smaller, sometimes exponentially, in size. For regular predicates, by detecting the predicate only on the abstracted computation, one is guaranteed to detect the predicate in the full computation, which leads to an efficient detection mechanism. By distributing the task of abstraction among all the processes, our distributed algorithm reduces the space required, as well as computational load on a single process by a factor of $O(n)$. We discuss optimizations that prevent redundant computations, and result in reduced number of messages.

REFERENCES


