Debugging Distributed Executions Using Language Recognition

Özalp Babaoğlu†, Eddy Fromentin‡, Michel Raynal†

To a large extent, the dependability of complex distributed programs relies on our ability to effectively test and debug their executions. Such an activity requires that we be able to specify dynamic properties that the distributed computation must (or must not) exhibit, and that we be able to construct algorithms to detect these properties at run time. In this paper we formulate dynamic property specification and detection as instances of the language recognition problem. Considering boolean predicates on states of the computation as an alphabet, dynamic property specification is akin to defining a language over this alphabet. Detecting a property, on the other hand, is akin to recognizing at run time if the sentence produced by a distributed execution belongs to the language. This formal language-oriented view not only unifies a large body of work on distributed debugging and property detection, it also leads to simple and efficient detection algorithms. We give examples for the case of properties that can be specified as regular grammars through finite automata.

Keywords: Program testing, distributed debugging, distributed executions, property detection, formal languages, finite-state automata.
1 Introduction

Our inability to formally prove the correctness of all but the most trivial distributed programs leaves testing and debugging as the only viable alternatives for arguing about their dependability. A fundamental step in program debugging is specifying which set of executions are considered desirable and which ones are considered erroneous. Informally, a desired (or undesired) temporal evolution of a distributed program’s states is called a dynamic property. As such, a dynamic property (property for short) defines a subset of executions among all those that are possible for the program. Once the properties of interest for a distributed program have been defined, the act of debugging consists of verifying if its executions satisfy these properties.

There are three possible times at which one can prove properties of a distributed program: prior to any execution, during an execution or after an execution. The ability to prove a program property prior to executing it requires reasoning about the program itself as well as the distributed system. In other words, we need to characterize all possible executions of the program given a formal description of its actions (the code) and the environment in which it is to be run. Model checking [5] is one such technique where the program, modeled as a finite-state transition system, is analyzed by traversing the trees representing all possible executions, checking to see if they satisfy properties expressed as temporal logic formulas. As an alternative to proving properties a priori, they can be checked concurrently with an actual execution of the program through run-time property detection. With this technique, conclusions that are drawn are not about all possible executions of the program, but about all possible observations [18, 1] of an actual execution. The third alternative — reasoning about program properties after an execution — is called post-mortem analysis and is similar to run-time property detection. The basic difference is that the analysis has to be based on data collected during the execution (traces) and is performed at program termination, thus making it unsuitable for reactive-architecture applications.

Ideally, one would like to prove as many properties as possible for a program prior to its execution. Unfortunately, techniques such as model checking may be infeasible or have prohibitive costs for complex programs. Furthermore, while these techniques are effective for proving program properties such as “is the program deadlock free?” they cannot address inherently run-time properties such as “has the program terminated?” Much of distributed debugging is an inherently run-time activity in that, short of having proven that the program is error free, we can only hope to detect desired or erroneous sequences of states that result during actual executions as soon as feasible. Applications that have a reactive architecture [12] (of which distributed debugging is an instance) are yet another example where the fundamental abstraction is run-time property detection. Clearly, run-time property detection and model checking can be seen as complementary techniques towards distributed debugging — the greater the number of properties that can be verified a priori, the fewer the number of properties that need to be detected at run time.

Early work in run-time property detection has concentrated on the detection of stable properties [4] including distributed termination [8] and deadlocks [3]. Informally, these properties are stable in the sense that once verified during an execution, they remain true thereafter. Efficient algorithms have been developed that can detect stable properties of distributed computations at run-time [4, 13]. Unfortunately, stable property detection has limited utility in the context of distributed debugging. Most of the properties that characterize desirable or erroneous executions for debugging purposes are transient whereby
they may be verified for some prefix of the execution but cease to be satisfied later on. As such, the appropriate formalism for distributed debugging can be seen as unstable property detection. If erroneous behaviors are specified as unstable properties, then their detection during an execution reveals a fault in the corresponding program. Existence of many executions during which the unstable properties are not detected increases our confidence in the correctness of the corresponding program.

In this paper we consider the problem of specifying unstable properties and detecting them at run time. We formalize the problem as the design of a decision algorithm that, when superimposed on a distributed execution, will answer “yes” if and only if the distributed execution satisfies a property specified as a formula in some formal language. We develop a general framework for property specification and detection drawing on concepts from formal language theory. The framework, described in Section 3, is based on the labeling of a directed acyclic graph (DAG). Each node of the DAG is associated a set of labels from a given alphabet and each path of the DAG is associated a set of words constructed from the same alphabet. The set of all words corresponding to all paths terminating at a node defines a language. The language recognition problem is then defined in Section 3.3 as the decision procedure for determining if a given word belongs to this language. We instantiate this abstract problem for two possible models of distributed computations as a DAG — in Section 4 as the partially ordered set of local states and in Section 5 as the lattice of consistent global states. When the alphabet used for labeling these DAGs consists of boolean predicates, the formal language associated with the nodes of the graph effectively specifies dynamic properties of the corresponding distributed computation. In addition, run-time property detection reduces to the problem of language recognition. In Sections 4.3 and 5.3 we give examples of simple run-time detection algorithms that result for the case of properties that can be specified as regular grammars through finite automata. This framework unifies a large body of work on distributed debugging and property detection including: linked predicates [17], conjunction of local predicates [11], unstable predicates on global states [6], interval-constrained sequences [2], regular patterns [10] and regular properties [15].

2 Models for Distributed Computations

A distributed program is one that is executed by a collection of sequential processes, denoted \( P_1, \ldots, P_n \) for some \( n > 1 \), that can communicate by exchanging messages. Processes have access to neither shared memory nor a global clock. Communication incurs finite but arbitrary delays. Without loss of generality, we assume that each process can reliably communicate with every other process.

2.1 Partially-Ordered Sets

Execution of process \( P_k \) produces a sequence of events \( h_k \) called its local history. Each event of the local history may be either internal, causing only a local state change, or involve communication with another process through send or receive events. Let \( h_k = e_0^k e_1^k \ldots \) be the local history of process \( P_k \). Events of \( h_k \) are enumerated according to the total order in which they are executed by \( P_k \) and \( e_0^k \) is a fictitious event introduced for initializing the local state of \( P_k \).

Let \( H \) be the set of all events and let \( \rightarrow \) be the binary relation denoting causal precedence [16] between events defined as follows:
modeled as yet another DAG, denoted an ordered set (poset) of a precedence relation between local states. The computation enters that state. More formally, a global state is a sequence of local states, one for each process. Intuitively, a global state of a computation is a set of all local states. Analogous to the causal precedence relation between local states as follows:

\[
e_i \prec e_j \iff \begin{cases} (i = j) \land (b = a + 1) \\ (e_i \prec e_j) \land (e_i = \text{send}(m)) \land (e_j = \text{receive}(m)) \\ \exists e_k : (e_i \prec e_k) \land (e_k \prec e_j) \end{cases}
\]

Formally, a distributed computation can then be modeled as the partially ordered set (poset) \( \mathcal{H} = (\mathcal{P}, \preceq) \). Figure 1 illustrates a distributed computation consisting of three processes using a graphical representation of the partial order between events known as a space-time diagram.

### 2.2 Directed Acyclic Graphs of Local States

Let \( \sigma_i^a \) be the local state of process \( P_i \) immediately after having executed event \( e_i^a \), and let \( S \) be the set of all local states. Analogous to the causal precedence relation between pairs of events, we define the binary relation \( \prec \) to denote immediate causal precedence between local states as follows:

\[
\sigma_i^a \prec \sigma_j^b \iff \begin{cases} (i = j) \land (b = a + 1) \\ (e_i^a \prec e_j^b) \land (e_j^b = \text{receive}(m)) \end{cases}
\]

With respect to the set of local states, a distributed computation can be modeled as yet another DAG, denoted \( S = (S, \prec) \). Figure 2 depicts such a DAG of local states corresponding to the distributed computation of Figure 1.

Two local states of a distributed computation that are related through \( \prec \) are said to be adjacent in that computation. A control flow associated with some local state \( \sigma \) of a computation \( S \) is a sequence of local states that are pairwise adjacent in \( S \) and the sequence begins at some initial state and terminates at \( \sigma \).

### 2.3 Lattices of Global States

A global state \( \Sigma = (\sigma_1, \ldots, \sigma_n) \) of a distributed computation is an \( n \)-tuple of local states, one for each process. Intuitively, a global state of a computation is said to be consistent if an omniscient external observer could actually observe the computation enter that state. More formally, a global state \( \Sigma = (\sigma_1, \ldots, \sigma_n) \) is consistent if and only if for all pairs of its local states \( (\sigma_i, \sigma_j) \), neither \( \sigma_i \prec \sigma_j \) nor \( \sigma_j \prec \sigma_i \), where \( \prec \) denotes the transitive closure of the immediate causal precedence relation between local states.
The set of all consistent global states for a distributed computation has a lattice structure whose minimal element corresponds to the initial global state $\Sigma^0 = (\sigma^0_1, \ldots, \sigma^0_n)$. Let $\mathcal{L}$ be this lattice. An edge exists from node $\Sigma = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n)$ to node $\Sigma' = (\sigma_1, \ldots, \sigma_i^{e+1}, \ldots, \sigma_n)$ if and only if there exists an event $e$ that can be executed by $P_i$ in local state $\sigma_i^e$. Figure 3 depicts the lattice $\mathcal{L}$ of global states associated with the distributed computation of Figure 1. In the lattice, an $n$-tuple $(x_1, \ldots, x_n)$ of natural numbers is used as a shorthand to denote the global state $\Sigma = (\sigma_1^n, \ldots, \sigma_n^n)$.

Informally, a sequential observation (observation for short) of a distributed computation is the sequence of its global states that could have been constructed by an omniscient external observer. Equivalently, an observation is a sequence of global states that would result if the distributed program were to be executed on a single sequential processor. More formally, a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \cdots \Sigma^{-1} \Sigma^i \cdots$ is an observation if there exists a sequence of events $e^1 e^2 \cdots$ that is a linear extension of the partial order $\mathcal{H}$ (i.e., all events appear in an order consistent with the relation $\rightarrow$) such that $\Sigma^i$ is the global state that results after executing event $e^i$ in $\Sigma^{i-1}$.

By construction, each path of the lattice starting at the minimal element and proceeding upwards corresponds to an observation of the computation and each observation corresponds to a path in the lattice [1, 18]. In other words, the lattice of consistent global states represents all possible observations for the computation. Note that, internal to the computation, the actual sequence of global states that is produced cannot be known and this lattice represents the best information that is available.

3 Properties as Languages over DAGs

In the previous section we have shown that a distributed computation $\mathcal{H}$ can be represented either as $\mathcal{S}$, the DAG of local states, or as $\mathcal{L}$, the DAG corresponding to the lattice of global states. In this section, we will develop a general frame-
work for specifying dynamic properties based on labeling a generic DAG. In the following sections, we will instantiate this framework with the two specific DAGs $\mathcal{S}$ and $\mathcal{L}$, depending on whether we are interested in sequences of local or global properties, respectively. Development of this framework helps us understand and unify a large number of proposals for detection of properties on local and global states as instances of a single problem.

3.1 Graph Labeling and Languages

Let $G = (V, E)$ be a DAG and let $A$ be a finite alphabet of symbols. We define a labeling function $\lambda$ that maps nodes of $G$ to non-empty sets of symbols drawn from $A$. For each node $v \in V$, the set $\lambda(v)$ is called the label of $v$ (we assume that the empty symbol $\epsilon$ is implicitly in every alphabet and constitutes the implicit label of a node if the labeling function defines no other symbols). Figure 4 illustrates a DAG of ten nodes labeled from the alphabet $A = \{a, b, c\}$.

For each node $v$, let $G_v$ be the subgraph obtained from $G$ by retaining only node $v$ and all of its predecessors in $G$. Clearly if $G$ is a DAG, then so is $G_v$. A directed path of $G_v$ is a sequence of nodes starting at a source node (i.e., one with no predecessors) and ending at node $v$. Let $\Pi_v$ be the set of all such paths in $G_v$. We extend the notion of labeling to paths by associating with them words constructed from the same alphabet used to label nodes. Let $A$ be a finite alphabet, $\lambda$ be a labeling function and $\pi_v = u_0u_1 \ldots u_k$ be a directed path of $G_v$. The label of path $\pi_v$ is the set $\lambda(\pi_v)$ of all words $\omega = \omega_0\omega_1 \ldots \omega_k$ such that $\omega_i \in \lambda(u_i)$ for each node $u_i$ in path $\pi_v$.

Since each path label is a set of words, sets of path labels can be seen as defining a language. The language associated with node $v$ of $G$ under the labeling function $\lambda$, denoted $L^\lambda(v)$, is defined as the set of words that are in the labels of all directed paths of $G_v$. In other words,

$$L^\lambda(v) = \bigcup_{\pi_v \in \Pi_v} \lambda(\pi_v).$$

As an example, consider node $v_{10}$ of the labeled DAG of Figure 4. The set of all directed paths for subgraph $G_{v_{10}}$ is $\Pi_{v_{10}} = \{v_1v_{10}, v_1v_2v_{10}, v_2v_5v_{10}, v_5v_6v_{10}, v_5v_7v_{10}\}$. Under the labeling function $\lambda$ that is illustrated, the language associated with node $v_{10}$ is $L^\lambda(v_{10}) = \{ac, abac, abac, acac, bcaac, acaca, bcaac, acac\}$.

3.2 Dynamic Properties

Informally, we would like dynamic properties to characterize the time evolution of states that occur during program execution. In our framework, a dynamic property (property for short) defines a set of words over some finite alphabet. Let
$L(\Phi)$ be the set of words associated with property $\Phi$. We find it convenient to distinguish the name of the property from the language that it defines.

Informally, we think of each word in language $L(\Phi)$ as satisfying property $\Phi$. In defining properties for DAGs, we can extend this notion of satisfaction to the entire graph in two possible ways. Given an alphabet $A$, a directed acyclic graph $G$, a labeling function $\lambda$, a node of $G$ and a property $\Phi$ over $A$, we define the following two satisfaction rules\(^1\):

**Definition 1** Given an alphabet $A$, a directed acyclic graph $G$, a labeling function $\lambda$, a node $v$ of $G$ and a property $\Phi$ over $A$, we say that node $v$ satisfies \textbf{SOME} $\Phi$, denoted $v \models \text{SOME} \Phi$, if and only if there exists some labeling of some directed path in $G_v$ that defines a word in the language of $\Phi$. In other words, $v \models \text{SOME} \Phi \equiv L^\lambda(v) \cap L(\Phi) \neq \emptyset$.

**Definition 2** Given an alphabet $A$, a directed acyclic graph $G$, a labeling function $\lambda$, a node $v$ of $G$ and a property $\Phi$ over $A$, we say that node $v$ satisfies \textbf{ALL} $\Phi$, denoted $v \models \text{ALL} \Phi$, if and only if all labelings of all directed paths in $G_v$ define words in the language of $\Phi$. In other words, $v \models \text{ALL} \Phi \equiv L^\lambda(v) \subseteq L(\Phi)$.

As an example, consider the two properties $\Phi_1$ and $\Phi_2$ defined over the alphabet $A = \{a, b, c\}$ that define languages (without loss of generality, they are specified using regular expressions with the usual syntax rules) $L(\Phi_1) = a(ba)^*(acac)^*$ and $L(\Phi_2) = (b + c)^*(a + b)^*(c(a + b + c)^*)$, respectively. With respect to the DAG of Figure 4, the following assertions hold (words $abacac$ and $abicac$ of $L^\lambda(v_{10})$ are in $L(\Phi_1)$):

\begin{align*}
v_{10} \models \text{SOME} \Phi_1 \\
v_{10} \models \text{ALL} \Phi_2.
\end{align*}

### 3.3 Detection of Dynamic Properties

For the sake of simplicity, in what follows we consider only properties that correspond to regular languages. As we shall see, many proposals including behavior patterns on local states [17, 10] and global states [6, 2] happen to be special cases of such properties. Moreover, these properties admit simple and efficient detection algorithms.

It is well known that regular grammars that specify regular languages are equivalent to deterministic finite-state automata. Formally, an automaton is a 5-tuple $(Q, A, q_0, Q_F, \delta)$ where $Q$ is a finite set of states, $q_0$ an initial state, $A$ a finite alphabet, $Q_F$ a set of accepting states and $\delta$ a deterministic transition function.

Let $\Phi$ be a property such that $L(\Phi)$ is a regular language. A finite state automaton that recognizes $L(\Phi)$ is given the name $\Phi$ just as the property itself. Figure 5 illustrates the two automata recognizing the languages associated with the properties $\Phi_1$ and $\Phi_2$ of the above example. In the Figure, accepting states are shown as triangles.

Given a property $\Phi$ and DAG $G$, let $R^\Phi(v)$ denote the set of states of the automaton recognizing $L(\Phi)$ that are reached after analyzing all of the words in $L^\lambda(v)$. Recall that $L^\lambda(v)$ is the language associated with node $v$ of $G$ defined as the set of all words that are in the labels of all directed paths of $G_v$. For the automata of Figure 5 recognizing $L(\Phi_1)$ and $L(\Phi_2)$, and the DAG of Figure 4, we have $R^\Phi(v_{10}) = \{q_5, q_7\}$ and $R^\Phi(v_{10}) = \{q_2\}$. Note that $R^\Phi(v_{10}) \cap Q_F \neq \emptyset$.

\(^1\) Satisfaction rules similar to these have been proposed in other contexts [6, 11, 15, 10, 9]. In particular, our definitions are in the same spirit as those of modal operators $\text{POS}$ and $\text{DEF}$ of [6], $\text{strong}$ and $\text{weak}$ of [11] and $\text{POT}$, $\text{INEV}$, $\text{SOME}$ and $\text{ALL}$ of more general transition systems [5].
while $R^F(v_0) \subseteq Q_F$. From the previous section, we know that for this example $v_0 \models \text{SOME } \Phi_1$ and $v_0 \models \text{ALL } \Phi_2$. In general, we can rewrite the satisfaction rules of Section 3.2 as follows as a consequence of the definitions:

\[
\begin{align*}
\models \text{SOME } \Phi & \equiv R^F(v) \cap Q_F \neq \emptyset \\
\models \text{ALL } \Phi & \equiv R^F(v) \subseteq Q_F.
\end{align*}
\]

Expressing the satisfaction rules in terms of relationships between the accepting states of the automaton and the set of reachable states gives us effective decision procedures for computing them.

Given a property $\Phi$, an automaton accepting $L(\Phi)$ and a labeled DAG $G$, the problem of detecting the property can be reduced to computing the sets $R^F(v)$ for each node $v$ of $G$. We proceed inductively in defining $R^F(v)$.

**Base case:** Let $G'$ be a DAG obtained from $G$ by adding a fictitious node $v_0$ that is an immediate predecessor of all source nodes of $G$. The node labeling function is extended such that $\lambda(v_0) = \{c\}$. Then, by definition

\[R^F(v_0) = \{q_0\}.\]

**Inductive step:** Let $\text{pred}(v)$ be the set of nodes that are immediate predecessors of node $v$ in $G$. Let $R^F_{\text{pred}(v)}$ be the set of reachable states of the automaton recognizing $L(\Phi)$ after analyzing all words associated with nodes in $\text{pred}(v)$. In other words,

\[R^F_{\text{pred}(v)} = \bigcup_{u \in \text{pred}(v)} R^F(u)\]

and by induction we have

\[R^F(v) = \bigcup_{\alpha \in R^F_{\text{pred}(v)}, q \in \lambda(v)} \delta(q, \alpha).\]

The above inductive definition can be easily transformed into a computation by doing a breadth-first traversal of $G$ starting at node $v_0$ as shown in Figure 6.

### 4 Properties on Control Flows

In this section, we will instantiate the generic dynamic property detection algorithm of the previous section in order to detect properties on control flows. In other words, the directed acyclic graph of interest is the DAG of local states $\mathcal{S} = (S, \prec)$ defined in Section 2.2 and the alphabet of interest is a set of local predicates.
\[ R^\Phi(v_0) := \{ \phi_0 \}; \text{done} := \{ v_0 \} \]

while (\exists v \in V : (v \notin \text{done}) \land (\exists u \in \text{done} : (u, v) \in E))
do

foreach \( v \notin \text{done} \) such that \( \text{pred}(v) \subseteq \text{done} \)
do

\( R^\Phi(v) := \{ \} \)

foreach \( \alpha \in \lambda(v), u \in \text{pred}(v), q \in R^\Phi(u) \)
do

\( R^\Phi(v) := R^\Phi(v) \cup \{ \delta(q, \alpha) \} \)
od

done := done \cup \{ v \}
od

Figure 6. A generic algorithm for detecting property \( \Phi \) in directed acyclic graph \( G = (V, E) \) with labeling function \( \lambda() \).

\[ P_1 \quad \begin{array}{c} \{ \phi_1, \phi_2 \} \\
\sigma_1(x = 2) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_1(x = 4) \end{array} \quad \begin{array}{c} \{ \phi_1, \phi_2 \} \\
\sigma_1(x = 2) \end{array} \]

\[ P_2 \quad \begin{array}{c} \{ \phi_1 \} \\
\sigma_2(y = 5) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_2(y = 4) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_2(y = 0) \end{array} \quad \begin{array}{c} \{ \phi_1 \} \\
\sigma_2(y = 4) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_2(y = 2) \end{array} \]

\[ P_3 \quad \begin{array}{c} \{ \phi_1 \} \\
\sigma_3(z = 1) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_3(z = 0) \end{array} \quad \begin{array}{c} \{ \phi_1 \} \\
\sigma_3(z = 4) \end{array} \quad \begin{array}{c} \{ \phi_1 \} \\
\sigma_3(z = 4) \end{array} \quad \begin{array}{c} \{ \} \\
\sigma_3(z = 2) \end{array} \]

Figure 7. DAG of local states annotated with values of three variables and labels.

4.1 Local Predicates

A local predicate is a formula in propositional logic (boolean expression) naming only variables that are local to a single process. Let \( \phi \) be such a local predicate. If the predicate holds in some local state \( \sigma \) of \( S \), we say that \( \sigma \) satisfies \( \phi \) and write \( \sigma \models \phi \). Let \( A \) be an alphabet consisting of a finite set of local predicates. We define a labeling function for the distributed computation \( S = (S, \prec) \) such that the labels of a local state are the local predicates it satisfies:

\[ \forall \sigma \in S : \lambda(\sigma) = \{ \phi \in A : \sigma \models \phi \} . \]

As an example, consider the distributed computation of Figure 7 where \( x \), \( y \) and \( z \) are three variables local to processes \( P_1 \), \( P_2 \) and \( P_3 \), respectively. Each local state \( \sigma_i^j \) is characterized by the value of the corresponding local variable. In Figure 7, values of the variables are shown next to each of the local states (e.g., in local state \( \sigma_1^2 \) the local variable \( y \) has value 2). Let us consider the following four local predicates \( \phi_1 \equiv (x < 3) \), \( \phi_2 \equiv (x \text{ is prime}) \), \( \phi_3 \equiv (y \neq 0) \) and \( \phi_4 \equiv (z < 4) \). For the computation of Figure 7, we have \( \sigma_1^0 \models \phi_1 \), \( \sigma_1^1 \models \phi_2 \), \( \sigma_1^2 \models \phi_2 \), \( \sigma_2^0 \models \phi_3 \), \( \sigma_2^1 \models \phi_3 \), \( \sigma_2^2 \models \phi_3 \), \( \sigma_3^0 \models \phi_4 \), \( \sigma_3^1 \models \phi_4 \), \( \sigma_3^2 \models \phi_4 \) and \( \sigma_3^3 \models \phi_4 \).

Thus \( S \) labeled with the alphabet \( A = \{ \phi_1, \phi_2, \phi_3, \phi_4 \} \) is as shown in Figure 7.

4.2 Behavior Patterns on Local States

Consider the class of properties that can be specified as a set of local predicates that need to be satisfied in a particular order during a computation. Each such sequence is called a behavior pattern. Recall that in Section 2.2 we defined a control flow associated with some local state \( \sigma \) as a sequence of local states that are pairwise adjacent in \( S \) and the sequence begins at some initial state and terminates at \( \sigma \). In other words, control flows of a computation correspond to
directed paths of $\mathcal{S}$. A property $\Phi$ on control flows expresses behavior patterns that these control flows must exhibit as words of the language $L(\Phi)$ constructed out of local predicates.

The satisfaction rules introduced in Section 3.2 have the following meaning when interpreted in the context of distributed computations as a DAG of local states:

**Definition 3** Given a computation $\mathcal{S}$ and property $\Phi$ on control flows, a local state $\sigma$ satisfies **SOME** $\Phi$ if and only if there exists a control flow $\pi_\sigma$ terminating at local state $\sigma$ such that $\hat{X}(\pi_\sigma)$ includes at least one word of $L(\Phi)$. In other words,

$$\sigma \models \textbf{SOME} \Phi \equiv L^\lambda(\sigma) \cap L(\Phi) \neq \emptyset.$$ 

**Definition 4** Given a computation $\mathcal{S}$ and property $\Phi$ on control flows, a local state $\sigma$ satisfies **ALL** $\Phi$ if and only if for each control flow $\pi_\sigma$ terminating at local state $\sigma$, every word of $\hat{X}(\pi_\sigma)$ is included in $L(\Phi)$. In other words,

$$\sigma \models \textbf{ALL} \Phi \equiv L^\lambda(\sigma) \subseteq L(\Phi).$$

For example, consider the properties $\Phi_1 = \phi_1 \phi_3^+, \Phi_2 = \phi_1 \phi_3^+ \phi_4, \Phi_3 = \phi_1 \phi_4$ and $\Phi_4 = (\phi_1 + \phi_2 + \phi_4)^* \phi_3^+$ on control flows specified as regular expressions from the alphabet $A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$ as defined before. For the computation of Figure 7, the following results hold: $\sigma_2^2 \models \textbf{ALL} \Phi_1, \sigma_2^3 \models \textbf{SOME} \Phi_2, \neg(\sigma_3^2 \models \textbf{SOME} \Phi_3), \text{ and } \neg(\sigma_3^3 \models \textbf{ALL} \Phi_4)$.

### 4.3 Run-Time Detection

As discussed in Section 3.3, regular languages can be recognized using deterministic finite state automata. In the case of properties on control flows, we can instantiate the generic algorithm of Figure 6 such that detection can be done at run time without introducing any delays and without adding any control messages to the distributed computation. All that is necessary is to piggyback control information on the existing messages of the computation.

Let $\Phi$ be a property such that $L(\Phi)$ can be recognized by the finite state automaton $\Phi = (Q, A, q_0, Q_f, \delta)$. A controller is superimposed on each process $P_i$ of the computation that maintains an array $A_i[Q]$ of boolean values with the following semantics: For each $q \in Q$, element $A_i[q] = \text{true}$ if and only if there exists a control flow $\pi_\sigma$ terminating at the current local state $\sigma$ of $P_i$ such that at least one word in $\hat{X}(\pi_\sigma)$ places automaton $\Phi$ in state $q$. Initially, only $A_i[q_0]$ is defined to be true. The algorithm executed by each controller can be easily derived from the generic algorithm and shown in Figure 8.

Let $A_\sigma$ denote the value of $A$ at local state $\sigma$ as maintained by the algorithm of Figure 8. Then, the satisfaction rules for **SOME** and **ALL** can be computed through the following simple definitions:

$$\sigma \models \textbf{SOME} \Phi \equiv \exists q \in Q : ((A_\sigma[q] = \text{true}) \land q \in Q_f)$$

$$\sigma \models \textbf{ALL} \Phi \equiv \forall q \in Q : ((A_\sigma[q] = \text{true}) \Rightarrow q \in Q_f)$$

### 5 Properties on Observations

Here properties are on sequences of global states and consequently the lattice $L$ of global states introduced in Section 2.3 instantiates the DAG $G$ of Section 3 and a behavior pattern is a word on an alphabet of global predicates.
when $P_i$ enters a new local state $\sigma_i$:

\[
\text{foreach } \alpha \in \lambda(\sigma_i) \text{ do } \\
A^\alpha[Q] := (\text{false}, \ldots, \text{false}) \\
\text{foreach } q \in Q \text{ such that } A_i[q] \text{ do } \\
\text{foreach } r \in \delta(q, \alpha) \text{ do } A^\alpha[r] := \text{true} \text{ od } \\
\text{od } \\
\text{foreach } q \in Q \text{ do } A_i[q] := \bigvee_{\alpha \in \lambda(\sigma_i)} A^\alpha[q] \text{ od }
\]

when $P_i$ sends a message $m$:

piggyback $A_i[Q]$ on $m$

when $P_i$ receives $m$ containing $A_{mp}$:

\[
\text{foreach } q \in Q \text{ do } A_i[q] := A_i[q] \lor A_{mp}[q] \text{ od }
\]

Figure 8. Algorithm executed by the controller of process $P_i$ in order to detect properties on control flows.

5.1 Global Predicates

A global predicate is a general boolean expression defined over a consistent global state; such an expression may reference any variable of any process. Satisfaction of global predicate $\varphi$ by a global state $\Sigma$ is denoted $\Sigma \models \varphi$.

The lattice introduced in Section 2.3 constitutes the basic model where properties related to global states are interpreted. Given a finite alphabet $A$ of global predicates, a labeling function $\lambda$ is defined on global states in a way analogous to that for local states:

$$\forall \Sigma : \lambda(\Sigma) = \{ \varphi \in A : \Sigma \models \varphi \}.$$
Consider for example the lattice of Figure 9 corresponding to the distributed computation of Figure 7 and the two global predicates:

\[ \varphi_1 \equiv ((x > y) \land (y > z)) \]
\[ \varphi_2 \equiv ((x \neq y) \Rightarrow (x > z)). \]

We can see that global state \( \Sigma_1 = (\sigma^1_1, \sigma^1_2, \sigma^1_3) \) (indicated as 111 in Figure 9) satisfies both \( \varphi_1 \) and \( \varphi_2 \), obtaining the label \( \{ \varphi_1, \varphi_2 \} \). The labeling of the lattice with the alphabet \( A = \{ \varphi_1, \varphi_2 \} \) is shown in Figure 9.

### 5.2 Behavior Patterns on Observations

A property \( \Phi \) on global states is expressed as a set \( I(\Phi) \) of sequences of global predicates: each sequence of global predicates defines a particular pattern. A sequence of global predicates defining a pattern must be satisfied by a sequence of global states in order for the corresponding pattern be recognized. In other words, such a property \( \Phi \) describes behavior patterns on observations. By definition, a distributed computation is said to satisfy a property \( \Phi \) if its final global state \( \Sigma^{last} \) satisfies it. Informally, the satisfaction rules introduced in Section 3.2 express the following properties:

**Definition 5** \( \Sigma^{last} \models \text{SOME } \Phi \) is true if and only if there exists a labeled observation of the computation that belongs to \( I(\Phi) \).

**Definition 6** \( \Sigma^{last} \models \text{ALL } \Phi \) is true if and only if all possible labelings of all observations of the computation belong to \( I(\Phi) \).

For the example of Figure 9, we have \( \Sigma^{last} \models \text{SOME } (\varphi^+_1, \varphi^+_2) \) with alphabet \( A = \{ \varphi_1, \varphi_2 \} \) while \( \Sigma^{last} \models \text{ALL } \varphi^+_2 \) with alphabet \( A = \{ \varphi_2 \} \).

### 5.3 Run-Time Detection

A basic prerequisite for detecting properties on global states is to build these global states in one way or another in order to check if they satisfy some global predicates. In the most general case, the entire lattice of global states, representing all observation has to be traversed. Each process \( P_i \) of the computation is augmented with a controller that sends local states produced by \( P_i \) to a monitor. The monitor pieces together local states received from the processes in order to construct consistent global states and to incrementally build the lattice. Several algorithm are known of doing such constructions [6] that use vector clocks to ensure consistency of the global states they compute.

Construction of the lattice and checking of properties is done at run time (i.e., concurrently with the distributed computation) by the monitor through the algorithm shown in Figure 10. As indicated in Section 3.3 we consider only the detection of properties corresponding to regular languages (and so each property \( \Phi \) can be recognized by a finite state automaton \( \Phi = (Q, A, q_0, Q_F, \delta) \)). With each global state \( \Sigma \) of the lattice, we associate a boolean array \( A_{\Sigma} \). For each \( q \in Q \), element \( A_{\Sigma}[q] \) is true if and only if there exists an observation terminating at \( \Sigma \) whose labeling puts the automaton in state \( q \). A fictitious global state \( \Sigma^{-1} \) is added to the lattice as the unique predecessor of \( \Sigma^0 \) and \( A_{\Sigma^{-1}}[q_0] \) is the only element of \( A_{\Sigma^{-1}} \) that is initially true. Function \( \text{pred}(\Sigma) \) returns the set of global states that immediately precede \( \Sigma \) in the lattice. The algorithm needs to consider only the global states of two adjacents levels of the lattice.
previous := \{ \Sigma^{-1} \}
current := \{ \Sigma^{0} \}
while (current \neq \emptyset) do
  foreach \Sigma \in current do
    foreach q \in Q do
      do \text{APred}[q] := \bigvee_{\Sigma' \in \text{Pred}(\Sigma)} A_{\Sigma'}[q] \od
    endforeach
    foreach \alpha \in \lambda(\Sigma) do
      \text{A}^\alpha[Q] := (false, \cdots, false)
    endforeach
    foreach q \in Q such that \text{APred}[q] = true do
      foreach r \in \delta(q, \alpha) do \text{A}^\alpha[r] := true \od
    endforeach
  endforeach
  foreach q \in Q do \text{A}_{\Sigma}[q] := \bigvee_{\alpha \in \lambda(\Sigma)} \text{A}^\alpha[q] \od
  endforeach
  previous := current;
current := \{ \text{global states directly reachable from those in previous} \}
od

Figure 10. Algorithm executed by the monitor in order to detect properties on observations.

maintained in sets previous and current (the immediate predecessors of a global state in current belongs to the set previous).

The algorithm computes values of \text{A}_{\Sigma}[Q] for each state \Sigma of current from the values associated with its predecessors and the labeling of \Sigma. Then, the satisfaction rules for \text{SOME} and \text{ALL} can be computed through the following simple definitions:

\[
\Sigma \models \text{SOME } \Phi \equiv \exists q \in Q : ((\text{A}_{\Sigma}[q] = \text{true}) \land q \in Q_f)
\]

\[
\Sigma \models \text{ALL } \Phi \equiv \forall q \in Q : ((\text{A}_{\Sigma}[q] = \text{true}) \Rightarrow q \in Q_f)
\]

This general detection algorithm for behavior patterns on observations can be simplified according to specifications of a particular behavior pattern. As indicated earlier, it includes as special cases proposals described in [6, 2].

6 Conclusion

This paper has first presented a general framework which allows the expression of properties (behavior patterns described as words on an alphabet of predicates). It has been shown that if the predicates are on local states the patterns are on the control flows of the execution; if the predicates are on global states, the patterns are on observations of the distributed executions (remember an observation is a sequence of global states that could be produced by executing the program on a monoprocessor). A control flow satisfies a pattern if the sequence of local states defining this control flow satisfies the sequence of local predicates defining the pattern. Similarly, an observation satisfies a pattern if
the sequence of global states defining this observation satisfies the sequence of global predicates defining the pattern.

It has been shown that the detection of regular patterns on control flows can be done at run time and without requiring additional messages; only piggybacking of an array of bits (one for each state of the finite state automaton) is necessary. So the detection of these properties is not expensive. Detection of regular patterns on observations, on the other hand, in general requires an additional monitor process whose aim is to build all possible observations of the computation. These detection algorithms can be seen as on-the-fly model checkers that work on only a small part of a directed acyclic graph of local or of global states.

From a practical point of view, run-time detection of dynamic properties (properties expressed here as regular patterns) is a fundamental point when one is interested in analyzing or debugging distributed executions. The work presented in this paper originated from a project whose aim is to design and implement a debugging facility for distributed programs [14]. The algorithms which have been described are currently implemented in this debugging facility.

References


