A Complete Set of Satisfaction Rules for Property Detection in Distributed Computations

Michel Hurfin, Masaaki Mizuno

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Complete Set of Satisfaction Rules for Property Detection in Distributed Computations

Michel HURFIN, Mazaaki MIZUNO

N ° 2908

Juin 1996

THÈME 1

ISSN 0249-6399

INRIA
RENNES

A Complete Set of Satisfaction Rules for Property Detection in Distributed Computations

Michel HURFIN*, Mazaaki MIZUNO†

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n° 2908 — Juin 1996 — 21 pages

Abstract: This paper presents a general framework for specification and detection of properties in distributed computations. A property of states in distributed computation in progress is defined by predicates (called *behavioral patterns*) and satisfaction rules (called *modal operators*). A behavioral pattern is obtained by combining basic predicates that are defined over either local states or consistent global states of the computation. In both cases, we model a distributed computation by a directed acyclic graph in which vertices represent (local or global) states and edges represent causal relation over the states. Specification and verification of behavioral patterns are formulated as instances of the language recognition problem, and basic concepts used in formal language theory are applied. A basic predicate is assimilated to a symbol, and a behavioral pattern is specified as a language (*i.e.* a set of words) defined over an alphabet of symbols.

Based on this model and given a behavioral pattern, we define four modal operators (*i.e.* four different satisfaction rules). Three of them are equivalent to modal operators previously introduced in related works. Finally, we present algorithms to verify each of the four modal operators over a class of behavioral patterns, called regular patterns.

Key-words: distributed computation, distributed debugging, property detection

(Résumé : tsvp)

* hurfin@irisa.fr – This work was supported in part by a post doctoral grant from the INRIA Research Institute.

† Department of Computer and Information Science, Kansas State University, Manhattan KS 66506. masaaki@cis.ksu.edu – This work was supported in part by the National Science Foundation under Grant CCR-9201645 and Grant INT-9406785.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Un ensemble complet de règles de satisfactions adaptées à la détection de propriétés dans les calculs répartis

Résumé : Cet article présente un modèle général pour spécifier et détecter des propriétés dans les calculs distribués. Une propriété portant sur le déroulement d'un calcul distribué est définie par un prédicat (appelé "motif comportemental") et une règle de satisfaction (également appelée "opérateur modal"). Un motif comportemental est obtenu en combinant des prédicats élémentaires spécifiés sous forme d'une expression booléenne portant soit sur l'état local d'un processus soit sur un état global cohérent. Dans les deux cas, un calcul réparti est modélisé par un graphe orienté sans circuit dans lequel un sommet représente un état (local ou global) tandis que les arcs représentent la relation de dépendance causale entre états. La spécification et la vérification des motifs comportementaux se ramènent à un problème de reconnaissance d'un langage formel et de fait, les concepts de base de la théorie des langages peuvent être employés. Un prédicat élémentaire est assimilé à un symbole et un motif comportemental est spécifié par un langage (*i.e.* un ensemble de mots) défini sur un alphabet de symboles.

A partir de ce modèle et étant donné un motif comportemental, nous définissons quatre opérateurs modaux (*i.e.* quatre règles de satisfaction différentes). Pour trois d'entre eux, nous soulignons les correspondances avec des opérateurs définis dans divers autres travaux. Finalement, nous présentons des algorithmes permettant de vérifier une propriété obtenue en associant l'un des quatre opérateurs modaux à un prédicat appartenant à une classe de motifs comportementaux appelés motifs reguliers.

Mots-clé : calculs répartis, mise au point répartie, détection de propriétés

1 Introduction

In a distributed computation, observation of execution states is essential for analysis, debugging, and control of the behavior of application programs. This paper presents algorithms to detect whether a desired (or undesired) property of program's states is satisfied during computation. Predicates over local variables of component processes are used to describe the properties.

Several classes of predicates have been proposed in the last several years. One class of predicates is called *basic predicates*. A basic predicate is a general boolean expression defined over either (1) the local state of one process (*local predicate*) or (2) a global state that consists of local variables of multiple processes (*global predicate*). Basic predicates may be used as building blocks to form a more general class of predicates called *behavioral patterns*. These predicates are of interest when one tries to grasp the progressive relevant changes which are useful in characterizing a computation. Due to the asynchronous nature of a distributed computation, behavioral patterns refer to a classical causal precedence relation between states rather than to a non-available global clock. Behavioral patterns include, among others, regular patterns [7, 13] and atomic sequences [12, 10]. Regular patterns include linked predicates [15], simple sequences [2, 9, 10], and interval-constrained sequences [2].

Verification of behavioral patterns over local predicates (*local behavioral patterns*) can be done at run time by piggy backing some additional information in messages. On the other hand, verification of behavioral patterns over global predicates (*global behavioral patterns*) generally requires construction of lattice of consistent global states at a designated process [4, 5]. Many algorithms that verify local behavioral patterns are described as part of message passing protocols. Many algorithms for global behavioral patterns are described by two separate algorithms: one to traverse lattice and another to verify predicates.

As did in [3], we model a distributed computation by a directed acyclic graph (dag), in which vertices represent states and edges represent dependent relation over states. This model gives us a unified view of verification of both local and global behavioral patterns. As it was previously suggested in [6, 7, 8, 13], each basic predicate is associated with a label, and the set of all the labels is considered as an alphabet. A set of labels is associated with each vertex in the dag; a label is in a set, if the state (modeled by the vertex) associated with the set satisfies the basic predicate denoted by the label. Consequently, a set of words is associated with each path in the labeled dag.

Specification and verification of behavioral patterns are formulated as instances of the language recognition problem. In this approach, a predicate is specified by a set of words (*i.e.* a language). For example, regular patterns use regular expressions over basic predicates to describe behaviors of a given execution. Detecting a behavioral pattern requires to traverse the labeled directed acyclic graph that models the distributed computation. Vertices are visited based on a topological sort of the dag.

Given a computation and a behavioral pattern Φ, we introduce four satisfaction rules and formally define the following four properties:

1. whether there exists a path that has words which verify the behavioral pattern ;

2. whether all the paths have some words which verify the behavioral pattern;

3. whether all the words on all the paths verify the behavioral pattern; and

4. whether there exists a path on which all the words verify the behavioral pattern.

Satisfaction rules similar to these have been proposed in other works. The first two properties are equivalent to the properties *possibly* Φ and *definitely* Φ defined in [4]. In [9], Garg and Waldecker call those two properties *weak* and *strong* formulae, respectively. In [3], Babaoglu and *et al.* defined properties $SOME\ \Phi$ and $ALL\ \Phi$ which are equivalent to the first and third properties, respectively.

This paper presents an algorithm (and its variations) to verify all of the four properties. Note that no algorithms that verify properties 2 and 4 in the framework of our general model have been proposed before. Properties 1 and 3 and properties 2 and 4 form dual relations. We present an alternative approach, based on this dual relation, to verify properties 3 and 4 by using the algorithm to verify properties 1 and 2. We analyze the space complexities of the two approaches.

The structure of the paper is as follows: the model of distributed computation is described in Section 2. Section 3 presents regular patterns and four properties based on the patterns. In Section 4, we present an algorithm and its variations for verifying the properties.

2 Computational Models

2.1 Distributed computation

A distributed computation consists of a collection of n sequential processes denoted $P_1, P_2, \ldots, P_n$ which communicate with each other only by means of message passing. The underlying system that supports the computation consists of processors connected by a network, where each process runs on a separate processor. Neither shared memory nor a global clock is available, and the system is asynchronous: there exists no bound on message delays, clock drift, or the time necessary to execute an action. Message passing between processes is carried out by a potentially unreliable protocol that transmits a single message at a time from an originating process to a destination process. Messages may be lost, duplicated or delivered out of order (*i.e.* channels are not necessarily FIFO). Yet all messages received are neither corrupted nor forged.

2.1.1 Events

During a computation, each process executes a sequence of actions asynchronously. Each occurrence of an action is called an event. Three kinds of events are considered: internal, send, and receive events. Let e_i^k denote the k^{th} event which occurs at P_i. Figure 1 shows an example of distributed computation performed by three processes. This graphical representation is called a space-time diagram. Horizontal lines represent execution of processes, with time progressing from left to right. An arrow from one process to another represents

a message being sent, with the send event at the base of the arrow and the corresponding receive event at the head of the arrow. Internal events (e_1^1 for example), have no arrows associated with them.

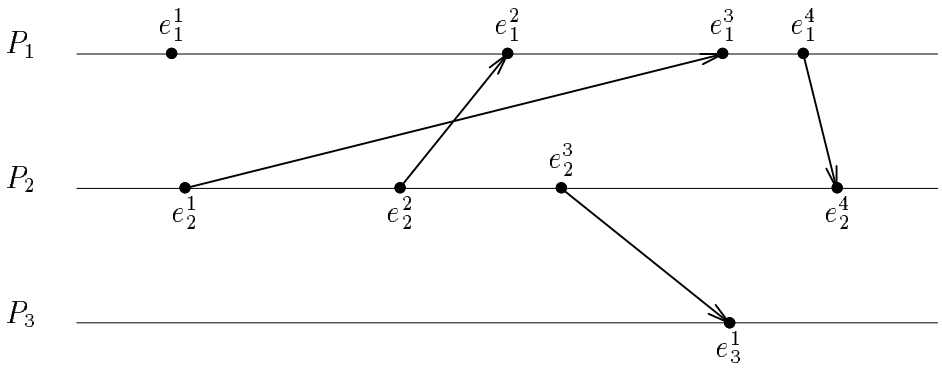

Figure 1: A Distributed Computation

A binary relation $\prec$ is defined over communication events as follows:

$$\forall\, e_i^x,\ \forall\, e_j^y,\quad e_i^x \prec e_j^y \iff \begin{cases} \text{There exists a message } m \text{ such that:} \\ \quad (1)\ m \text{ is sent during the execution of } e_i^x \\ \quad (2)\ m \text{ is consumed during the execution of } e_j^y \end{cases}$$

Relation $\prec$ expresses causal dependencies between the pairs of corresponding send and receive events. Based on this relation, we define a direct precedence relation among the events of a distributed computation, denoted $\xrightarrow{e}$.

$$\forall\, e_i^x,\ \forall\, e_j^y,\quad e_i^x \xrightarrow{e} e_j^y \iff \begin{cases} (i=j) \wedge (y = x+1) \\ \text{or} \\ e_i^x \prec e_j^y \end{cases}$$

The transitive closure of $\xrightarrow{e}$, denoted $\xrightarrow{e*}$, is a partial order relation and equivalent to the Lamport's "happen before" relation [14].

2.1.2 Local states

The local state of a process P_i is defined over the set of its local variables. The values of variables may change only when an event occurs. Although an occurrence of an event does not always cause a change in the local state, we identify the local state of a process at a given time with respect to the last occurrence of an event at the process. We use σ_i^x to denote the local state of P_i during the period of time between atomic executions of e_i^x and e_i^{x+1}. σ_i^0 is

called the initial local state of process P_i. Hence, each process, P_i, consists of a sequence of interleaved local states and events: $< \sigma_i^0, e_i^1, \sigma_i^1, e_i^2, \sigma_i^2, \ldots, \sigma_i^{x-1}, e_i^x, \sigma_i^x, e_i^{x+1}, \ldots >$.

Relation $\stackrel{\sigma}{\rightarrow}$ expresses direct precedence among the local states of the distributed computation. This relation is defined as follows.

$$\forall\ \sigma_i^x,\ \ \forall\ \sigma_j^y,\quad \sigma_i^x\ \stackrel{\sigma}{\rightarrow}\ \sigma_j^y\ \iff\ \left\{\begin{array}{ll} & (i=j)\wedge(y=x+1)\\ \text{or} & \\ & e_i^{x+1}\ \prec\ e_j^y \end{array}\right.$$

Figure 2 shows all the local states of the computation depicted in Figure 1 and their direct precedence relation.

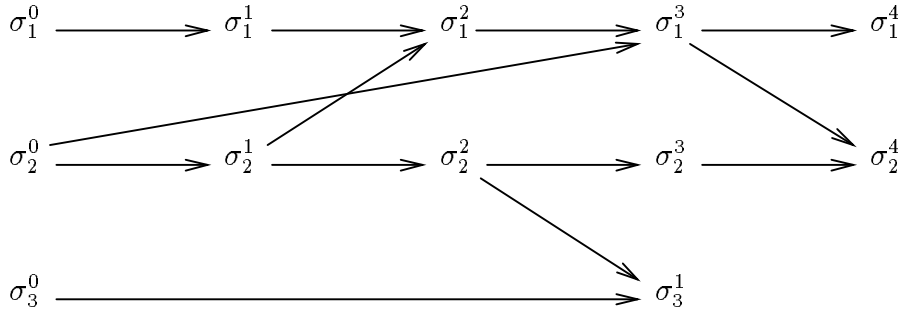

Figure 2: Direct Precedence Relation Between Local States

The transitive closure of $\stackrel{\sigma}{\rightarrow}$, denoted $\stackrel{\sigma*}{\rightarrow}$, is a partial order relation. Two local states σ_i^x and σ_j^y are said to be *concurrent* if there is no causal dependency between them (*i.e.* neither $\sigma_i^x \stackrel{\sigma*}{\rightarrow} \sigma_j^y$ nor $\sigma_j^y \stackrel{\sigma*}{\rightarrow} \sigma_i^x$).

2.1.3 Consistent global state

A set of local states is *consistent* if any pair of elements are concurrent. In the distributed computation shown in Figure 1, $\{\sigma_1^0\}$, $\{\sigma_1^1, \sigma_2^0\}$ and $\{\sigma_1^1, \sigma_2^2\}$ are three examples of consistent sets of local states.

A global state denoted by $\{\sigma_1^{x_1}, \sigma_2^{x_2}, \ldots, \sigma_n^{x_n}\}$ is a collection of n local states, containing exactly one local state for each process P_i. A consistent set of n local states $\{\sigma_1^{x_1}, \sigma_2^{x_2}, \ldots, \sigma_n^{x_n}\}$ is called *a consistent global state* and identified by $\Sigma^{x_1 x_2 \ldots x_n}$.

A direct precedence relation between consistent global states is denoted $\stackrel{\Sigma}{\rightarrow}$ and defined as follows:

$$\forall\ \Sigma^{x_1 x_2 \ldots x_n},\ \ \forall\ \Sigma^{y_1 y_2 \ldots y_n},\quad \Sigma^{x_1 x_2 \ldots x_n}\ \stackrel{\Sigma}{\rightarrow}\ \Sigma^{y_1 y_2 \ldots y_n}\ \iff$$
$$\exists\ i,\ 1 \le i \le n,\ \text{such that}\ \ \forall\ j,\ 1 \le j \le n, \left\{\begin{array}{ll} & (j=i)\wedge(y_j = x_j + 1)\\ \text{or} & \\ & (j \neq i)\wedge(y_j = x_j) \end{array}\right.$$

Intuitively, a relationship exists from consistent global state $\Sigma^{x_1 \ldots x_i \ldots x_n}$ to consistent global state $\Sigma^{x_1 \ldots x_i+1 \ldots x_n}$ if, in the distributed computation, the latter state can be reached from the former when process P_i executes its next event e_i^{x+1}.

The set of all consistent global states of a distributed computation along with the relation $\stackrel{\Sigma}{\rightarrow}$ defines a lattice whose minimal element is the initial global state $\Sigma^{00\ldots0}$. Figure 3 illustrates the global state lattice corresponding to the distributed computation depicted in Figure 1.

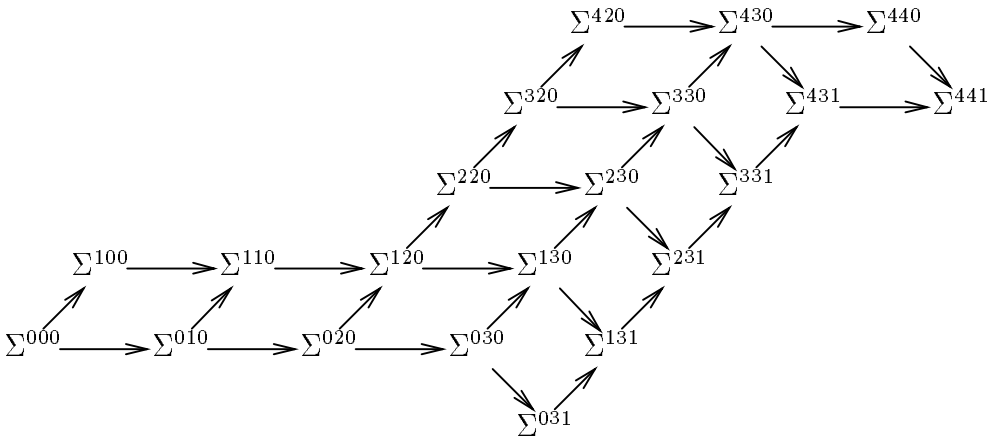

Figure 3: Direct Precedence Relation Between Consistent Global States

Each path of the lattice starting at the minimal element corresponds to a possible observation of the distributed computation. Each observation is identified by a sequence of events where all events of the computation appear in an order consistent with the relation $\stackrel{e*}{\rightarrow}$ (*i.e.* the "happen before" relation by Lamport).

2.2 General model

We model a distributed computation by an irreflexive directed acyclic graph (dag for short) $G = (V, E)$. G is a finite, nonempty set of *vertices* called *states* and E is a set of *edges* (*i.e.* ordered pairs of vertices). In this paper, we consider two different ways to define this graph.

1. A model based on local states

   In that model, $V = V_l$ is the set consisting of all the local states of all processes, and R is defined by:

$$\forall\ \sigma_i^x \in V,\quad \forall\ \sigma_j^y \in V,\quad (\sigma_i^x, \sigma_j^y) \in E \iff \sigma_i^x \stackrel{\sigma}{\rightarrow} \sigma_j^y$$

Each path in the graph is part of a control flow (*i.e.* a sequence of local states).

2. A model based on consistent global states

   In this model, $V = V_g$ represents the set of all consistent global states in the computation and R is defined by:

$$\forall\ \Sigma^{x_1 x_2 \ldots x_n} \in V,\ \ \forall\ \Sigma^{y_1 y_2 \ldots y_n} \in V,\ \ (\Sigma^{x_1 x_2 \ldots x_n}, \Sigma^{y_1 y_2 \ldots y_n}) \in E \iff \Sigma^{x_1 x_2 \ldots x_n} \stackrel{\Sigma}{\Rightarrow} \Sigma^{y_1 y_2 \ldots y_n}$$

   Each path in the graph is part of an observation (*i.e.* a sequence of consistent global states).

2.3 A suitable model for a property detection problem

2.3.1 Relevant States

To avoid being overwhelmed by numerous irrelevant low-level information, the programmer adapts the complexity of his model (*i.e.* the graph G) to the requirements of his analysis by identifying a subset of relevant states.

Let $S_l \subseteq V_l$ and $S_g \subseteq V_g$ be the set of relevant local states and the set of relevant consistent global state, respectively. A dag $M = (S, R)$ is used to represent the distributed computation where S is equal to either $S_l \cup \{s^\perp, s^\top\}$ or $S_g \cup \{s^\perp, s^\top\}$. Note that we introduce two special states, $s^\perp$ and $s^\top$, called the source vertex and the sink vertex, respectively.

The set of edges R is defined as follows:

1. Let s^x and s^y be two relevant states. (s^x, s^y) belongs to R if, in graph G, there exists at least one path from s^x to s^y such that all the states on this path belong to the set $((V - S) \cup \{s^x, s^y\})$.

2. Let s be a relevant state. $(s^\perp, s)$ belongs to R if, in graph G, there exists no path from a relevant state to s.

3. Let s be a relevant state. $(s, s^\top)$ belongs to R if, in graph G, there exists no path from s to a relevant state.

In [6, 7], a method to reduce the cardinality of the set of edges R is also proposed. We will not discuss their reduction method in this paper.

2.3.2 Basic predicates defined over local/global relevant states

The simplest class of predicates is called *basic predicates*. A basic predicate refers to the program's execution state at a given time. These predicates are divided into two classes: *local predicates* and *global predicates*. A local predicate is a general boolean expression defined over the local state of a single process, whereas a global predicate is a boolean expression

dealing with the local states of at least two distinct processes (*i.e.* a global predicate is defined over a consistent set of local states).

Detecting whether a global predicate is satisfied during a computation usually requires construction and traversal of a lattice of consistent global states that represents all observations of the computation. Therefore, if at least one global predicate is used as a building block to state a condition on the program's execution states (*global behavioral pattern*), the model based on relevant global states must be used.

On the contrary, evaluation of a local predicate can be done locally without introducing any delays, without defining a coordinating process, or without exchanging any control messages. Conditions on the program's execution state that combine only local predicates (*local behavioral patterns*) can be detected by just piggybacking control information in messages exchanged by the application. In this case, the model based on relevant local states can be used.

2.3.3 Labels

Basic predicates are specified by the programmer. Let Π denote the set of all the labels.

Given a model $M = (S, R)$, let λ be the function from S to 2^{Π} (the power set of Π) such that $\lambda(s)$ is the set of labels representing basic predicates evaluated to true in state s. That is, a label ξ is associated with a state s (*i.e.* $\xi \in \lambda(s)$) if and only if basic predicate denoted by ξ is verified when computation is in state s. We assume that the "empty" label ϵ is implicitly associated with every relevant states for which the labeling function defines no label. Furthermore: $\lambda(s^{\perp}) = \lambda(s^{\top}) = \{\epsilon\}$. In the rest of the paper, we use "label" and "basic predicate" interchangeably. A computation is now modeled by its labeled dag M.

Figure 4 shows a labeled dag $M = (S, R)$ obtained by selecting a set of relevant local states in the dag of Figure 3. Labels ξ_1, ξ_2, ξ_3 and ξ_4 are associated with four local predicates. In this Figure, the value of the set $\lambda(s)$ is indicated near the relevant state s: $\lambda(\sigma_0^2) = \{\xi_2\}$, $\lambda(\sigma_1^2) = \{\xi_1, \xi_2\}$, $\lambda(\sigma_2^2) = \{\xi_3\}$ and $\lambda(\sigma_2^4) = \{\xi_1, \xi_4\}$.

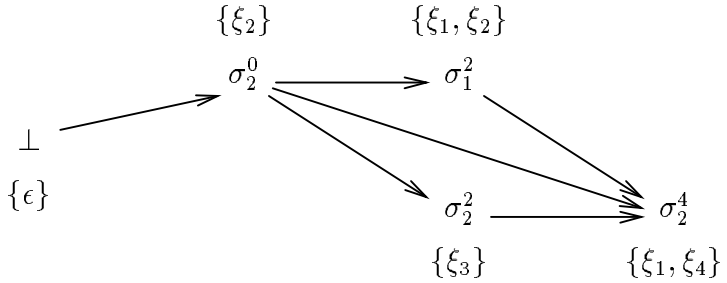

Figure 4: The Labelled Dag of Relevant Local States

2.3.4 Predecessors, successors, and paths

For each state $s \in S$, we define two sets of states denoted, Pred(s) and Succ(s), as follows:

- Pred(s) $=$ $\{ s^p \in S \mid (s^p, s) \in R \}$: the set of immediate predecessor states of s.
- Succ(s) $=$ $\{ s^s \in S \mid (s, s^s) \in R \}$: the set of immediate successor states of s.

We define a set of sequences of states for a pair of states $(s^x, s^y) \in S \times S$, denoted Path($s^x, s^y$), as follows: A sequence of states $\Omega = s^{h_1} \cdot s^{h_2} \cdot \cdots s^{h_m}$ belongs to Path(s^x, s^y) iff:

$$\begin{cases} & s^{h_1} = s^x \\ \wedge & \\ & s^{h_m} = s^y \\ \wedge & \\ & (s^{h_k}, s^{h_{k+1}}) \in R \ , \ \text{for all } k,\ 1 \leq k \leq m-1 \end{cases}$$

A sequence $\Omega \in$ Path(s^x, s^y) is called a *path* (from s^x to s^y). A single vertex of the graph, s, constitutes a sequence in M whose length is equal to 0.

For example, in Figure 4, Pred(σ_1^2) = $\{\sigma_2^0\}$ and Pred(σ_2^4) = $\{\sigma_1^2\ ,\ \sigma_2^0\ ,\ \sigma_2^2\}$. Succ($\sigma_1^2$) = $\{\sigma_2^4\}$ and Succ(σ_2^0) = $\{\sigma_1^2\ ,\ \sigma_2^2\ ,\ \sigma_2^4\}$. Path($\perp, \sigma_2^4$) = $\{\perp \cdot \sigma_2^0 \cdot \sigma_1^2 \cdot \sigma_2^4\ ,\ \perp \cdot \sigma_2^0 \cdot \sigma_2^4\ ,\ \perp \cdot \sigma_2^0 \cdot \sigma_2^2 \cdot \sigma_2^4\}$.

2.3.5 Alphabet, words, and languages

The set of label Π can be interpreted as an alphabet of symbols. The set of all words over this alphabet is denoted by Π^*.

Each path $\Omega = s^1 \cdot s^2 \cdot \cdots s^m$ is associated with a set of sequences of labels, denoted Word(Ω), that is defined by:

$$\text{Word}(\Omega) = \{\omega \mid \omega = \xi_1\ \xi_2\ \cdots \xi_m \ \text{ where } \ \xi_k \in \lambda(s^k) \ \text{ for all } k,\ 1 \leq k \leq m \}$$

A sequence of labels $\omega \in$ Word(Ω) is called a *word* (on path Ω).

By definition: $\forall s \in S, \text{Word}(s) = \lambda(s)$.

In the graph in Figure 4, Word($\sigma_2^0 \cdot \sigma_1^2 \cdot \sigma_2^4$) = $\{\xi_2\xi_1\xi_1, \xi_2\xi_1\xi_4, \xi_2\xi_2\xi_1, \xi_2\xi_2\xi_4\}$, Word($\sigma_2^0 \cdot \sigma_2^4$) = $\{\xi_2\xi_1, \xi_2\xi_4\}$ and Word($\sigma_2^0 \cdot \sigma_2^2 \cdot \sigma_2^4$) = $\{\xi_2\xi_3\xi_1, \xi_2\xi_3\xi_4\}$

3 Properties as Languages over Labeled Dags

3.1 Regular patterns

A *regular pattern* describes a property of distributed computation M by a regular expression $\Re$ over basic predicates in Π [13, 7, 6]. Let $L(\Re)$ denote the language described by $\Re$. For $\Re$, there exists a nondeterministic finite automata (NFA) such that word ω is in $L(\Re)$ if ω is accepted by the NFA. An NFA is described by a 5-tuple (Q, Π, δ, q_0, F) where

1. Q is a finite set of NFA states;[1]
2. Π is a set of basic predicates associated with states in a computation;
3. $q_0 \in Q$ is the initial NFA state;
4. $F \subseteq Q$ is the set of final NFA states; and
5. δ is the transition function mapping $Q \times \Pi$ to 2^Q.

We extend δ to map $Q \times \Pi^*$ to 2^Q by reflecting sequences of inputs as follows: Let ω be a sequence of inputs, ξ be a basic predicate, and q be an NFA state. Then,

1. $\delta(q, \epsilon) = \{q\}$, and
2. $\delta(q, \omega \cdot \xi) = \{q_i \mid q_i \in \delta(q_j, \xi) \text{ where } q_j \in \delta(q, \omega)\}$

$\delta(q_0\ ,\ \omega)$ represents a set of NFA states in which the NFA could be in after reading ω. Furthermore, we extend δ to a function $\widehat{\delta}$ that maps $2^Q \times \Pi^*$ to 2^Q by:

3. $\widehat{\delta}(P, \omega) = \bigcup_{q \in P} \delta(q, \omega)$, where P is a set of NFA states.

3.2 Examples of regular patterns

Several classes of regular patterns have been proposed. Let $\star$ denote any symbol in Π and $\star - \{\xi\}$ denote any symbol in $(\Pi - \{\xi\})$. Let φ_i and θ_i be basic predicates in Π.

Miller and Choi introduced *Linked predicates* [15]. They described linked predicates by giving the corresponding regular expressions. A linked predicate $\varphi_1 \rightarrow \varphi_2 \rightarrow \cdots \rightarrow \varphi_m$ is described by regular expression $\varphi_1\ (\star - \{\varphi_2\})^*\ \varphi_2\ \cdots\ (\star - \{\varphi_m\})^*\ \varphi_m$.

Simple sequences are defined in [2, 9, 10]. A simple sequence of length m is denoted $\varphi_1; \varphi_2; \cdots; \varphi_m$. The corresponding regular expression is $(\star)^*\ \varphi_1\ (\star)^*\ \varphi_2\ \cdots\ (\star)^*\ \varphi_m\ (\star)^*$.

In [2], Babaoglu and Raynal defined *Interval-constraint sequences*. An interval-constraint sequence of length m, denoted $\overline{\theta_1}\ \varphi_1;\ \overline{\theta_2}\ \varphi_2;\ \cdots;\overline{\theta_m}\ \varphi_m;\ \overline{\theta_{m+1}}$, is described by regular expression $(\star - \{\theta_1\})^*\ \varphi_1\ (\star - \{\theta_2\})^*\ \varphi_2\ \cdots\ (\star - \{\theta_m\})^*\ \varphi_m\ (\star - \{\theta_{m+1}\})^*$.

As example, the NFA associate with interval-constraint sequence $\overline{\xi_1}\ \xi_2; \overline{\xi_3}\ \xi_4; \overline{\xi_1}$ is given in Figure 5.

3.3 Properties based on regular expressions

We describe behavioral patterns of a computation M by a regular expression $\Re$ over basic predicates in Π. To analyze computation M from state $s^\perp$ to state s, the following properties are of interest:

[1] We use term "state" to denote a state in computation $M = (S, R)$ and term "NFA state" to denote a state of an NFA.

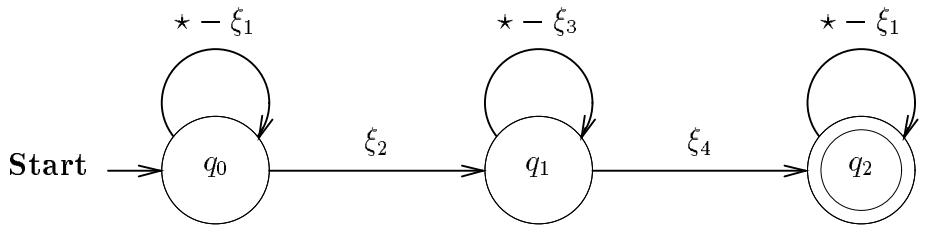

Figure 5: A nondeterministic finite automata

1. For each path Ω from $s^{\perp}$ to s in M,
   (a) whether there exists a word on Ω that is in $L(\Re)$.
   (b) whether all the words on Ω are in $L(\Re)$.
2. For computation M from $s^{\perp}$ to s,
   (a) whether there exists a path that satisfies either (a) or (b) in 1.
   (b) whether all the paths in the computation satisfy either (a) or (b) in 1.

Combining the above 1 and 2, we define the following four properties:

Definition Φ: For a regular expression $\Re$, a computation $M = (S, R)$, and a state $s \in S$:

1. $\Phi^{\exists\exists}(\Re, s) \equiv \exists\, \Omega \in \text{Path}(s^{\perp}, s),\ \exists\, \omega \in \text{Word}(\Omega),\ \omega \in L(\Re)$
2. $\Phi^{\forall\exists}(\Re, s) \equiv \forall\, \Omega \in \text{Path}(s^{\perp}, s),\ \exists\, \omega \in \text{Word}(\Omega),\ \omega \in L(\Re)$
3. $\Phi^{\exists\forall}(\Re, s) \equiv \exists\, \Omega \in \text{Path}(s^{\perp}, s),\ \forall\, \omega \in \text{Word}(\Omega),\ \omega \in L(\Re)$
4. $\Phi^{\forall\forall}(\Re, s) \equiv \forall\, \Omega \in \text{Path}(s^{\perp}, s),\ \forall\, \omega \in \text{Word}(\Omega),\ \omega \in L(\Re)$

Note that:

$$\omega \in L(\Re) \iff \exists\, q \in \delta(q_0, \omega) \text{ such that } q \in F$$

We may omit parameters $\Re$ and s, when they are obvious or not important. Sometimes, we write Φ^{**}, where $*$ represents $\exists$ or $\forall$.

Pairs of modal operators, **possibly** and **definitely**, are defined in [4]. They are essentially the same as **weak** and **strong** defined in [9] and similar to **SOME** and **ALL** defined in [3]. These modal operators can be described by Φ as follows: For $\Re$, a computation M satisfies

1. **possibly** $\Re$ **/ weak** $\Re$ **/ SOME** $\Re$ if $\Phi^{\exists\exists}(\Re, \top)$ holds; and
2. **definitely** $\Re$ **/ strong** $\Re$ if $\Phi^{\forall\exists}(\Re, \top)$ holds; and

3. **ALL** $\Re$ if $\Phi^{\forall\forall}(\Re, \top)$ holds.

Let $\overline{\Re}$ be the regular expression associated with the language $\Pi^* - L(\Re)$ (*i.e.* the complement of language $L(\Re)$). Then, it is easy to see that the following *dual property* holds.
Dual Property :

- **[A]** $\Phi^{\exists\forall}(\Re, s) \equiv \neg\ \Phi^{\forall\exists}(\overline{\Re}, s)$
- **[B]** $\Phi^{\forall\forall}(\Re, s) \equiv \neg\ \Phi^{\exists\exists}(\overline{\Re}, s)$.

In the next section, we will present an algorithm (and its variations) to verify each of Φ^{**}. We will also show a method to verify, based on the dual property, $\Phi^{\exists\forall}$ and $\Phi^{\forall\forall}$ by using algorithms for $\Phi^{\forall\exists}$ and $\Phi^{\exists\exists}$, respectively. We will compare the space complexities of these two approaches.

4 Verification Algorithms

This section presents an algorithm that verifies whether a given computation M satisfies various properties described by Φ^{**}. We assume the following:

1. Graph $M = (S, R)$ represents the computation, and λ denotes the labeling function on S.
2. Properties to be verified are $\Phi^{\exists\exists}(\Re, s)$, $\Phi^{\forall\exists}(\Re, s)$, $\Phi^{\exists\forall}(\Re, s)$, and $\Phi^{\forall\forall}(\Re, s)$.
3. NFA denotes an nondeterministic finite automaton (Q, Π, δ, q_0, F) that accepts $L(\Re)$.

First, we present the strategy used to traverse graph M. The graph can be constructed off-line after the execution has terminated or at run time as the computation progresses. If the graph is constructed at run time, it can be used to verify local regular patterns [2] or to verify global regular patterns (Algorithms for constructing the lattice of global states at run time are discussed in [4, 5]).

Then, we show an algorithm and its variations to verify various Φ^{**}. We first show a general algorithm that verifies all of the four properties: $\Phi^{\exists\exists}, \Phi^{\forall\exists}, \Phi^{\forall\forall}$, and $\Phi^{\exists\forall}$. We identify necessary information to verify all the properties. Then, we explain the structure of the general algorithm, that is the basis of all the variations. If only some of (not all) the properties need to be verified, the amount of information can be reduced. We present three variations that reduce the amount of information. The space complexity of each variation is analyzed.

Finally, we briefly discuss alternative methods to compute $\Phi^{\forall\forall}$ and $\Phi^{\exists\forall}$ based on the dual property presented in Section 3.3.

4.1 Graph traverse

The verification algorithm traverses M, starting at $s^{\perp}$. Let $\Psi_{State}(s)$ denotes information necessary to verify $\Phi^{**}(\Re, s)$. This information is computed when the state s is visited. The traversal is done in the following manner: a vertex s is visited after all its predecessors have been visited; that is, the order of traversal follows a topological sort of the graph.

First, we initialize $\Psi_{State}(s^{\perp})$. The initial values differ in the variations of the verification algorithm.

If the verification is performed at run time, traversing the graph is automatic. When an event causes a change in states, from s_p to s (in this case, we say "s is activated"), then the run time verification algorithm computes $\Psi_{State}(s)$. Starting from $s^{\perp}$, each state s passes its $\Psi_{State}(s)$ value to its successor states. Obviously when s is activated, all the predecessor states have already been activated and completed. Thus, by inductive arguments, when the algorithm is computing $\Psi_{State}(s)$ in s, it has computed $\Psi_{State}(s^p)$ for the immediate predecessor states s^p. Since the algorithm that computes $\Psi_{State}(s)$ receives $\Psi_{State}(s^p)$ values only from their predecessor nodes, the states in Pred(s) are easily identified.

In the algorithm that verifies local regular patterns (refer to the model given in Section 2.3 and Figures 1 and 4), if s is activated by a receive event, then s^p may be in a different process. In such a case, the message carries $\Psi_{State}(s^p)$.

4.2 A general algorithm

This algorithm computes $\Psi_{State}(s)$ when it visits state s. First, we discuss what information the algorithm must maintain in order to compute all four Φ^{**}.

4.2.1 Information about each word ω

Let $\Psi_{Word}(\omega)$ denote the information that the algorithm maintains for each word ω. Recall that when the NFA reads word ω, it will be in a set of states defined by $\delta(q_0, \omega)$. The algorithm determines whether ω is accepted by the NFA by checking $\delta(q_0, \omega) \cap F \neq \emptyset$. Thus, $\Psi_{Word}(\omega) = \delta(q_0, \omega)$.

4.2.2 Information about each path Ω

Recall that a set of words, Word(Ω), is associated with each path Ω. Let $\Psi_{Path}(\Omega)$ denote the information that the algorithm maintains for each path Ω. Since the general algorithm must be able to verify whether every word in Word(Ω) is in $L(\Re)$, it needs to maintain $\Psi_{Word}(\omega)$ for each word ω in Word(Ω) separately. Thus, $\Psi_{Path}(\Omega) = \{\Psi_{Word}(\omega) \mid \omega \in \text{Word}(\Omega)\}$.

4.2.3 Information about each state s

Recall that a set of paths, Path($s^{\perp}, s$), is associated with computation M from state $s^{\perp}$ to state s. Since the general algorithm must be able to verify whether every word of every path

in Path$(s^{\perp}, s)$ is in $L(\Re)$, it needs to maintain the information about each path separately. Thus, $\Psi_{State}(s) = \{\Psi_{Path}(\Omega) \mid \Omega \in \text{Path}(s^{\perp}, s)\}$.

4.2.4 Description of the algorithm

Let s^p be a state in Pred(s). For each path $\Omega_{sp} \in$ Path$(s^{\perp}, s^p)$, the visit to state s extends Ω_{sp} to $\Omega_{sp} \cdot s$, Thus,

[**P1**] $\Psi_{State}(s) = \{\Psi_{Path}(\Omega_{sp} \cdot s) \mid \Omega_{sp} \in \text{Path}(s^{\perp}, s^p),$ where $s^p \in \text{Pred}(s)\}$.

In the process of the extension, each label in $\lambda(s)$ is concatenated with every word in Word(Ω_{sp}). For example, suppose that Word$(\Omega_{sp}) = \{\omega_1, \omega_2\}$, and that $\lambda(s) = \{\xi_1, \xi_2\}$. Then, Word$(\Omega_s)$ is $\{\omega_1 \cdot \xi_1,\ \omega_1 \cdot \xi_2,\ \omega_2 \cdot \xi_1,\ \omega_2 \cdot \xi_2\}$. Thus,

[**P2**] $\Psi_{Path}(\Omega_{sp} \cdot s) = \{\Psi_{Word}(\omega_i \cdot \xi_j) \mid \omega_i \in \text{Word}(\Omega_{sp}), \xi_j \in \lambda(s)\}$.

From the definition of δ,

[**P3**] $\Psi_{Word}(\omega_i \cdot \xi_j) = \bigcup_{q \in \Psi_{Word}(\omega_i)} \delta(q, \xi_j)$.

A detailed description of the general algorithm is given in Figure 6.

In summary, the algorithm performs the following operations, when visiting new state s:

1. $\Psi_{State}(s)$ is a set of $\Psi_{Path}(\Omega_{sp} \cdot s)$ computed as follows (refer to [P1] and to Lines from 1 to 15 in Figure 6):
2. For each immediate predecessor state s^p in Pred(s) (refer to Line 2):
3. For each path Ω_{sp} in Path$(s^{\perp}, s^p)$ (refer to Line 3), let $\Psi_{Path}(\Omega_{sp} \cdot s)$ be a set of $\Psi_{Word}(\omega_i \cdot \xi_j)$ computed as follows (refer to [P2] and to Lines from 4 to 13):
4. For each word ω_i in Word(Ω_{sp}) (refer to Line 5):
5. For each label ξ_j in $\lambda(s)$ (refer to Line 6):
6. $\Psi_{Word}(\omega_i \cdot \xi_j) = \bigcup_{q \in \Psi_{Word}(\omega_i)} \delta(q, \xi_j)$ (refer to [P3] and Lines from 7 to 10)

Procedure "Initialization" that is invoked in the algorithm to traverse M sets $\Psi_{State}(s^{\perp})$ to $\{\{\{q_0\}\}\}$.

After the algorithm has computed $\Psi_{State}(s)$, it applies Definition Φ given in Section 3.3 to $\Psi_{State}(s)$ to verify Φ^{**}.

```
      Type
            Word_Description  :  set of NFA's States;
            Path_Description  :  set of Word_Description;
            State_Description  :  set of Path_Description;

      procedure Visit(s : State);
      Variable:
           q  :  NFA's State;
           Ψ_Word , New_Ψ_Word  :  Word_Description;
           Ψ_Path , New_Ψ_Path  :  Path_Description;
           Ψ_State(s)  :  State_Description;
      begin
1     Ψ_State(s)  :=  ∅;
2     foreach  s^p ∈ Pred(s)  do
3          foreach  Ψ_Path ∈ Ψ_State(s^p)  do
4               New_Ψ_Path  :=  ∅;
5               foreach  Ψ_Word ∈ Ψ_Path  do
6                    foreach  ξ ∈ λ(s)  do
7                         New_Ψ_Word  :=  ∅;
8                         foreach  q ∈ Ψ_Word  do
9                              New_Ψ_Word  := New_Ψ_Word  ∪ δ(q,ξ);
10                        od
11                        New_Ψ_Path  :=  New_Ψ_Path ∪ { New_Ψ_Word };
12                   od
13              od
14              Ψ_State(s)  :=  Ψ_State(s) ∪ { New_Ψ_Path };
15         od
      end
```

Figure 6: General algorithm

4.3 Verification of $\Phi^{\forall\exists}(\Re, s)$

$\Phi^{\forall\exists}(\Re, s)$ is true iff there exists at least one word ω such that $\omega \in L(\Re)$ in every path in Path$(s^{\perp}, s)$. Thus, $\Psi_{State}(s)$ needs to maintain information on $\Psi_{Path}(\Omega_s)$ separately, for each $\Omega_s \in$ Path$(s^{\perp}, s)$.

However, given a path Ω_s, the algorithm does not need to identify which words in Word(Ω_s) are in $L(\Re)$ and which words are not. Thus, for each path Ω_s, $\Psi_{Path}(\Omega_s)$ may maintain the combined information (union) of $\Psi_{Word}(\omega)$ (which is equal to $\delta(q_0, \omega)$) for all $\omega \in$ Word(Ω_s), rather than keeping $\Psi_{Word}(\omega)$ separately. By doing so, we can eliminate data structure Ψ_{Word}, and $\Psi_{Path}(\Omega_{s^p} \cdot s)$ becomes a set of NFA states, computed by

[P4] $\Psi_{Path}(\Omega_{s^p} \cdot s) = \bigcup_{q \in \Psi_{Path}(\Omega_{s^p})} (\bigcup_{\xi_j \in \lambda(s)} \delta(q, \xi_j))$.

The algorithm to compute $\Psi_{State}(s)$ for $\Phi^{\forall\exists}$ is described as follows:

1. $\Psi_{State}(s)$ is a set of $\Psi_{Path}(\Omega_{s^p} \cdot s)$ computed as follows (refer to [P1] in the general algorithm):
2. For each immediate predecessor state s^p in Pred(s):
3. Set $\Psi_{Path}(\Omega_{s^p} \cdot s) = \emptyset$.
4. For each path Ω_{S_p} in Path$(s^\perp, s^p)$:
5. For each label ξ_j in $\lambda(s)$:
6. $\Psi_{Path}(\Omega_{s^p} \cdot s) = \Psi_{Path}(\Omega_{s^p} \cdot s) \cup (\bigcup_{q \in \Psi_{Path}(\Omega_{s^p})} \delta(q, \xi_j))$ (refer to [P4]).

Procedure "Initialization" of the algorithm sets $\Psi_{State}(s^\perp)$ to $\{\{q_0\}\}$.

Note that this algorithm can still verify $\Phi^{\exists\exists}$ but can no longer verify $\Phi^{\exists\forall}$ or $\Phi^{\forall\forall}$.

4.4 Verification of $\Phi^{\forall\forall}(\Re, s)$

$\Phi^{\forall\forall}(\Re, s)$ is true iff all the words of all the paths are in $L(\Re)$. It is not necessary to identify which path each word belongs to. Thus, $\Psi_{State}(s)$ does not need to maintain information on $\Psi_{Path}(\Omega_s)$ for each $\Omega_s \in$ Path$(s^\perp, s)$ separately. Instead, $\Psi_{State}(s)$ may maintain combined information (union) of $\Psi_{Path}(\Omega_s)$. On the other hand, it must verify whether every word (no matter which path it belongs to) is in $L(\Re)$. Thus, it must maintain $\Psi_{Word}(\omega)$ separately, for each ω in Word(Ω_s). By doing so, data structure Ψ_{Path} is eliminated, and $\Psi_{State}(s)$ becomes a set of Ψ_{Word}'s.

Let s^p be a node in Pred(s). The set of words associated with the computation M from $s^\perp$ to s^p is $\bigcup_{\Omega_{s^p} \in \text{ Path}(s^\perp, s^p)}$ Word(Ω_{s^p}). Therefore, $\Psi_{State}(s)$ is computed as follows:

[P5] $\Psi_{State}(s) = \{\Psi_{Word}(\omega_i \cdot \xi_j) \mid \omega_i \in \bigcup_{\Omega_{s^p} \in \text{ Path}(s^\perp, s^p)}$ Word(Ω_{s^p}) where $s^p \in$ Pred$(s), \xi_j \in \lambda(s)\}$

The algorithm to compute $\Psi_{State}(s)$ for $\Phi^{\forall\forall}$ is described as follows:

1. $\Psi_{State}(s)$ is a set of $\Psi_{Word}(\omega_i \cdot \xi_j)$, computed as follows (refer to [P5]):
2. For each immediate predecessor state s^p in Pred(s):
3. For each word ω_i in $\bigcup_{\Omega_{s^p} \in \text{ Path}(s^\perp, s^p)}$ Word(Ω_i):
4. For each label ξ_j in $\lambda(s)$
5. $\Psi_{Word}(\omega_i \cdot \xi_j) = \bigcup_{q \in \Psi_{Word}(\omega_i)} \delta(q, \xi_j)$ (refer to [P3] in the general algorithm).

Procedure "Initialization" of the algorithm sets $\Psi_{State}(s^\perp)$ to $\{\{q_0\}\}$.

Note that this algorithm can still verify $\Phi^{\exists\exists}$ but can no longer verify $\Phi^{\exists\forall}$ or $\Phi^{\forall\exists}$.

4.5 Verification of $\Phi^{\exists\exists}(\Re, s)$

$\Phi^{\exists\exists}(\Re, s)$ is true iff there exists a path that has at least one word that is in $L(\Re)$. Thus, $\Psi_{State}(s)$ may maintain only the combined information (union) of all the words on all the paths in the computation M from $s^{\perp}$ to s. Therefore, data structures Ψ_{Path} and Ψ_{Word} are eliminated and $\Psi_{State}(s)$ becomes a "set of NFA states."

[**P6**] $\Psi_{State}(s) = \bigcup_{s^p \in \text{ Pred}(s)} (\bigcup_{\xi \in \lambda(s)} (\bigcup_{q \in \Psi_{State}(s^p)} \delta(q, \xi))))$

The algorithm to compute $\Psi_{State}(s)$ for $\Phi_{\exists\exists}$ is described as follows:

1. Set $\Psi_{State}(s) = \emptyset$
2. For each immediate predecessor state s^p in Pred(s):
3. For each label ξ_j in $\lambda(s)$
4. $\Psi_{State}(s) = \Psi_{State}(s) \cup (\bigcup_{q \in \Psi_{State}(s^p)} \delta(q, \xi_j))$

Procedure "Initialization" of the algorithm sets $\Psi_{State}(s^{\perp})$ to $\{q_0\}$.

4.6 Optimization

In verification of $\Phi^{\forall\exists}$ and $\Phi^{\forall\forall}$, $\Psi_{State}(s)$ maintains a set of sets of NFA states. We use Θ to denote a set of NFA states (*i.e*, $\Theta \in 2^Q$). In both $\Phi^{\forall\exists}$ and $\Phi^{\forall\forall}$, the property is true if $\Theta \cap F \neq \emptyset$ for every $\Theta \in \Psi_{State}(s)$.

Let Θ_i and Θ_j be two sets of NFA states such that $\Theta_i \subseteq \Theta_j$. Let ω be a word. Then, from the definition of $\widehat{\delta}$, $\widehat{\delta}(\Theta_i, \omega) \subseteq \widehat{\delta}(\Theta_j, \omega)$ holds. Thus, if $\widehat{\delta}(\Theta_i, \omega) \cap F \neq \emptyset$, then $\widehat{\delta}(\Theta_j, \omega) \cap F \neq \emptyset$. This implies that if both Θ_i and Θ_j are in $\Psi_{State}(s)$ and $\Theta_i \subseteq \Theta_j$ holds, it is not necessary for $\Psi_{State}(s)$ to keep Θ_j, in order to verify $\Phi^{\forall\exists}$ or $\Phi^{\forall\forall}$ in future.

This leads to the following optimization rule:

Optimization Rule : In computation of $\Psi_{State}(s)$ for $\Phi^{\forall\exists}$ or $\Phi^{\forall\forall}$, after computing $\Psi_{State}(s)$, the following set may replace $\Psi_{State}(s)$: $\{\Theta \in \Psi_{State}(s) \mid \nexists \Theta' \in \Psi_{State}(s), \Theta' \subset \Theta\}$.

Note that the new algorithms described above can no longer be used to verify $\Phi^{\exists\exists}$.

4.7 Space complexity

The basic data structure of all the algorithms is a set of NFA states Θ. One way to represent a set of NFA states is to use a $|Q|$ bit word and define $q_i \in \Theta$ iff the i^{th} bit of the word is 1.

4.7.1 The general algorithm

$\Psi_{State}(s)$ maintains information on every word on every path separately. Thus, it needs to maintain a set of sets of sets of NFA states. Since each word requires $|Q|$ bits, the number of bits required to represent $\Psi_{State}(s)$ is bounded by $\min(|Q| * (\sum_{\Omega \in \text{ Path}(s^{\perp}, s)} |\text{Word}(\Omega)|) \ , \ 2^{2^{|Q|}})$.

4.7.2 The algorithms for $\Phi^{\forall\exists}(\Re, s)$ and $\Phi^{\forall\forall}(\Re, s)$

Since the optimization rule allows $\Psi_{State}(s)$ to keep only Θs that are not included by any other Θ. The upper bound of $|\Psi_{State}(s)|$ is equal to the maximal number of antichain in the partially ordered structure $(2^Q, \subset)$. In 1928, E. Sperner proved that this number is equal to the number of combinations of $\lceil |Q|/2 \rceil$ distinct objects chosen from Q, denoted $\mathrm{Comb}(|Q|, \lceil |Q|/2 \rceil)$ (See [1] for a complete explanation of this result). Thus, the upper bound of the number of bits required to represent $\Psi_{State}(s)$ is

1. $|Q| * \min(|\mathrm{Path}(s^\perp, s)|,\ \mathrm{Comb}(|Q|, \lceil |Q|/2 \rceil))$ for $\Phi^{\forall\exists}$; and
2. $|Q| * \min(\sum_{\Omega \in \mathrm{Path}(s^\perp, s)} |\mathrm{Word}(\Omega)|,\ \mathrm{Comb}(|Q|, \lceil |Q|/2 \rceil))$ for $\Phi^{\forall\forall}$.

The lower bound is obviously $|Q|$.

4.7.3 The algorithm for $\Phi^{\exists\exists}(\Re, s)$

$\Psi_{State}(s)$ is of type "set of NFA states." Thus, the number of bits required to represent $\Psi_{State}(s)$ is $|Q|$.

4.8 Alternative approach to verify $\Phi^{\forall\forall}$ and $\Phi^{\exists\forall}$

The dual property presented in Section 3.3 indicates that $\Phi^{\forall\forall}$ and $\Phi^{\exists\forall}$ can be verified by using the algorithms to verify $\Phi^{\exists\exists}$ and $\Phi^{\forall\exists}$, respectively.

We briefly sketch one approach to verify $\Phi^{\forall\forall}$ (or $\Phi^{\exists\forall}$):

1. If the finite automaton that accepts $L(\Re)$ is nondeterministic, convert the NFA to an equivalent deterministic finite automaton (DFA) [11].
2. Convert the DFA to accept $L(\overline{\Re})$ [11].
3. Apply the algorithm to verify $\Phi^{\exists\exists}$ (or $\Phi^{\forall\exists}$) and negate the result.

The number of the DFA states is between $|Q|$ and $2^{|Q|}$. A simple method to represent this information is to assign 1 bit for each DFA state, as we do in the NFA. Thus, the number of bits required to verify $\Phi^{\forall\forall}$ is between $|Q|$ and $2^{|Q|}$. The number of bits required to verify $\Phi^{\exists\forall}$ is bounded by $\mathrm{Comb}(2^{|Q|}, 2^{|Q|-1})$.

For example, recall the NFA given in Figure 5. If we apply the algorithm for $\Phi^{\forall\forall}$ described in Section 4.4, 3 bits are required to represent a set of NFA states. The maximal number of distinct sets of sets of NFA states is $\mathrm{Comb}(3, 2) = 3$. Thus, in the worst case, the number of bits required to verify $\Phi^{\forall\forall}$ is $9 = (3 * 3)$. In other cases, the algorithm maintains one set or two sets of NFA states. Thus, the number of bits can be 3 or 6.

Now consider that $\Phi^{\exists\exists}$ is used to verify $\Phi^{\forall\forall}$. An equivalent DFA has 8 distinct states. Thus, it always requires 8 bits to verify $\Phi^{\forall\forall}$.

5 Conclusion

In a distributed computation, observation of execution states is essential for analysis, debugging, and control of the behavior of application programs. Observation of unstable properties of execution states is done by verifying predicates that are defined over local or consistent global states of the computation.

We modeled a distributed computation by a directed acyclic graph in which vertices are local or global states and edges represent causal relation on the states. This model gives us a unified view of verification of both local and global predicates.

In the model, we defined four properties based on predicates and presented an algorithm to verify all four properties based on a class of predicates, called regular patterns.

References

[1] M. Aigner: "Combinatorial Theory", Springer Verlag, 483 pages, 1979.

[2] Ö. Babaoğlu and M. Raynal: "Specification and detection of behavioral patterns in distributed computations", *Journal of Parallel and Distributed Computing 28*, 1995.

[3] Ö. Babaoğlu, E. Fromentin, and M. Raynal: "A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations", *The Journal of Systems and Software*, special issue on Software Engineering for Distributed Computing, To appear 1996.

[4] R. Cooper, and K. Marzullo: "Consistent Detection of Global Predicates", In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 163–173, Santa Cruz, California, May 1991.

[5] C. Diehl, C. Jard, and J. X. Rampon: "Reachability analysis on distributed executions", In *Theory and Practice of Software Development*, pp. 629–643, TAPSOFT, Springer Verlag, LNCS 668, Gaudel and Jouannaud editors, April 1993.

[6] E. Fromentin, Cl. Jard, G. Jourdan, and M. Raynal: "On-the-fly Analysis of Distributed Computations", *Information Processing Letters 54*, pp. 267–274, 1995.

[7] E. Fromentin, M. Raynal, V.K. Garg, and A.I. Tomlinson: "On the fly testing of regular patterns in distributed computations", In *Proc. of the the* 23^{rd} *Int. Conf. on Parallel Processing*, pp. 73–76, St. Charles, IL, August 1994.

[8] V.K. Garg, A.I. Tomlinson, E. Fromentin, and M. Raynal: "Expressing and Detecting General Control Flow Properties of Distributed Computations", In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 432–438, San-Antonio (USA), October 1995.

[9] V.K. Garg and B. Waldecker: "Detection of Unstable Predicates in Distributed Programs", In *Proc. of the 12*th *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag, LNCS 652, pp. 253–264, New Delhi, India, December 1992.

[10] M. Hurfin, N. Plouzeau, and M. Raynal: "Detecting atomic sequences of predicates in distributed computations", In *Proc. ACM workshop on Parallel and Distributed Debugging*, pp. 32–42, San Diego, May 1993. (Reprinted in SIGPLAN Notices, Dec. 1993).

[11] J. E. Hopcroft and J. D. Ullman. "Introduction to Automata Theory, Languages, and Computation", Addison-Weslay Publishing Company, 418 pages, 1979.

[12] D. Haban and W. Weigel. "Global Events and Global Breakpoints in Distributed Systems", In *Proc.* 21^{st} *Hawaii Int. Conf. on System Sciences*, pp 166–175, January, 1988.

[13] Cl. Jard, T. Jeron, G.V. Jourdan, and Rampon J.X. "A general approach to trace-checking in distributed computing systems", In *Proc. IEEE Int. Conf. on DCS*, Poznan, Poland, June, 1994.

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

[15] B.P. Miller and J. Choi. "Breakpoints and halting in distributed programs", In *Proc.* 8^{th} *IEEE Int. Conf. on Distributed Computing Systems, San Jose*, pp 316–323, July 1988.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399