Expressing and Detecting Control Flow Properties of Distributed Computations

Vijay Garg, Alex Tomlinson, Eddy Fromentin, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Expressing and Detecting Control Flow Properties of Distributed Computations

Vijay K Garg, Alex Tomlinson, Eddy Fromentin, Michel Raynal

N° 2384

Octobre 1994

PROGRAMME 1

ISSN 0249-6399

Expressing and Detecting Control Flow Properties of Distributed Computations

Vijay K Garg*, Alex Tomlinson*, Eddy Fromentin**, Michel Raynal**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projets Adp

Rapport de recherche n° 2384 — Octobre 1994 — 19 pages

Abstract:

Properties of distributed computations can be either on their global states or on their control flows. This paper addresses control flow properties. It first presents a simple yet powerful logic for expressing general properties on control flows, seen as sequences of local states. Among other properties, we can express invariance, sequential properties (to satisfy such a property a control flow must match a pattern described as a word on some alphabet) and non-sequential properties (these properties are on several control flows at the same time).

A decentralized detection algorithm for properties described by this logic is then presented. This algorithm, surprisingly simple despite the power of the logic, observes the underlying distributed computation, does not alter its control flows and uses message tags to carry detection-related information.

Key-words: behavioral property, distributed computation, distributed debugging, on the fly detection, control flows.

(Résumé : tsvp)

*Dept of ECE – University of Texas at Austin, Austin TX, e-mail:{vijay, alext}@pine.ece.utexas.edu
**IRISA – Campus de Beaulieu, 35042 RENNES cedex – FRANCE, e-mail:{fromenti, raynal}@irisa.fr

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

Expression et détection de propriétés sur les flots de contrôle d'une exécution répartie

Résumé :

Les propriétés d'une exécution répartie peuvent être exprimées sur ses états globaux ou bien sur ses flots de contrôle. Nous nous intéressons dans ce rapport aux propriétés sur les flots de contrôle. Dans un premier temps, est présentée une logique simple (quoique puissante) permettant d'exprimer des propriétés sur les flots de contrôle vus comme des séquences d'états locaux. Au moyen de cette logique, nous pouvons notamment exprimer des propriétés telles l'invariance, les propriétés séquentielles (un flot de contrôle satisfait une propriété séquentielle si et seulement si il peut être mis en correspondance avec un motif décrit comme un mot sur un alphabet) et les propriétés non séquentielles (ces propriétés ne considèrent pas qu'un seul flot de contrôle au même instant).

Nous présentons ensuite un algorithme décentralisé permettant la détection de toute propriété décrite au moyen de cette logique. Cet algorithme, bien que simple en dépit de sa puissance d'expression, observe l'exécution répartie sous-jacente en en préservant les flots de contrôle (les informations nécessaire à la détection sont transmises dans les messages de l'application).

Mots-clé : propriété comportementale, exécution répartie, déverminage de programmes répartis, detection au vol, flots de contrôle.

1 Introduction

Contrary to model checking, which works at "compile-time" on representations of all possible executions of a distributed program [4], run-time detection of properties of distributed executions is concerned by a single but real execution. Two classes of *run-time properties* have been identified: ones that are on global states and ones that are on control flows of a distributed execution. In the first case, global states have to be computed. If the property is stable (once true it remains true forever), a snapshot algorithm can be used to detect it [3]. When the property is not stable, all the global states, through which the computation could have passed, have to be considered. These states constitute a lattice [1, 15]; a node of the lattice represents a possible global state of the distributed computation and an edge represents an event that changes the global state of the computation. A general method to detect such unstable properties consists in building the lattice associated with the distributed execution [5, 6] and then in traversing it to detect the property [2, 12]; this can be done on the fly by pipelining the construction and the traversal of the lattice with the execution. The basic problem with the detection of general (unstable) properties on global states is that the size of the lattice can be exponential with respect to the number of processes [1, 15]. However for some specific properties on global states such as conjunction of local predicates [9], relational global predicate [16] or inevitable global states [7, 9] the lattice construction is not necessary.

The subject of this paper is the second class of properties, namely the ones on control flows of a distributed computation. A control flow is a sequence of causally related events produced by a distributed computation or, equivalently, the sequence of local states produced by these events. It is important to note that, due to messages exchanges, control flows visit processes, merge and fork. The set of all control flows can easily be determined from Lamport's partial order relation on events of a distributed computation [13] (usually called *happened before* or *causal precedence*). One of the first proposal to express properties on control flows was introduced in [14], under the name *linked predicates*. Linked predicates describe a causal sequence of local states where each state in the sequence satisfies a specific local predicate. The behavior "an occurrence of local predicate p is causally followed by an occurrence of local predicate q" is an example of a linked predicate. Algorithms for linked predicates appear in [9, 11, 14]. A generalization of linked predicates to a broader class called *atomic sequences of predicates* has been proposed in [11]. In this class, occurrences of local predicates can be forbidden between adjacent predicates in linked predicates. The example given above for linked predicates could be expanded to include: "q follows p and r never occurs in between" (note that p,q, and r could all

occur in different processes). More general *regular patterns* were introduced in [8]; a property is then specified by a regular expression of local predicates. For example $pq^\star r$ is true in a computation if there exists a sequence of local states $(s_1, s_2, \cdots, s_n)$ such that p is true in s_1, q is true in $s_2, \cdots, s_{n-1}$ and r is true in s_n. Note that the states in the sequence need not belong to the same process. Regular patterns are *sequential*, which means that they can be expressed as a set of words on some alphabet (elements of the alphabet are local predicates which must be satisfied by local states).

This paper introduces a simple but powerful logic that can express general properties on control flows of which sequential properties are a special case. A labeled poset of local states is used to model a distributed computation. Each state in the poset has a set of labels which represent boolean expressions (local predicates) which are true in that state. In this model the global past of any local state s forms a labeled directed acyclic graph such that s is a root in the graph. We call these structures LRDAGs (labeled rooted DAGs). Formulas in the logic express properties of LRDAGs. Thus a formula can be thought of as a boolean function whose argument is an LRDAG. Moreover in a labeled poset there is a one to one relationship between states and LRDAGs, thus we can also think of a formula as a boolean function on local states.

The paper also presents a decentralized, yet surprisingly simple, algorithm to detect the formulas expressed with this logic; so this detection algorithm includes as special cases the ones described in [14, 11, 8]. The detection algorithm is superimposed on the distributed computation. It is passive in the sense it can only observe the computation (it can neither initiate or inhibit the sending or receiving of messages nor alter the control flow of the observed computation). The memory and time overhead of the detection algorithm is a function of the formula being detected, and not of the number of processes. Typically, this overhead is quite low.

The paper is divided into four main Sections. Section 2 presents the model of distributed executions. Section 3 introduces the LRDAG logic to express general properties on control flows. Section 4 presents some uses of this logic. Section 5 presents a decentralized algorithm that detects properties expressed as formulas of this logic.

2 Model of Distributed Executions

2.1 Distributed Programs

A distributed program consists of N processes (denoted $P_1, P_2, \ldots, P_N$) which can communicate with each other only via messages. It is assumed that messages cannot be forged — that is, if process P_1 receives a message that appears to be sent from state s in process P_2, then P_2 did send that message from state s. We assume that information can be piggybacked on messages. Message channels need not be reliable or FIFO.

2.2 Partial Order on States

Each process, P_i, consists of a sequence of interleaved states and events: $(s_i^0, e_i^1, s_i^1, e_i^2, s_i^2 \ldots)$. Initially, P_i is in state s_i^0. Event e_i^k transforms state s_i^{k-1} into state s_i^k. (Throughout the paper we use i to index processes, and k to index sequences of states. We use s_i^k to denote a specific state at a specific process, and s to denote non-specific states.)

Let S be the set of states of all the processes. (The term "state" always refers to a local state of a single process). We assume that $s_i^x = s_j^y$ if and only if $x = y$ and $i = j$. We define two relations on $S \times S$ as follows:

- *local predecessor* relation: $\prec$

$$s_i^x \prec s_j^y \iff i = j \ \wedge \ y = x + 1$$

- *remote predecessor* relation: $\rightsquigarrow$

$$s_i^x \rightsquigarrow s_j^y \iff e_i^x \text{ is the sending of message } m \text{ and } e_j^y \text{ is the reception of } m$$

The tuple $\tilde{S} \triangleq (S, \prec, \rightsquigarrow)$ models a distributed computation. The *causally precedes* relation $\rightarrow$ is defined as the transitive closure of $\prec \cup \rightsquigarrow$. This relation is Lamport's relation [13] applied to states. The set of states S is partially ordered by $\rightarrow$. Any execution of any distributed program can be modeled by a partially ordered set of local states.

Figure 1 shows a distributed execution and Figure 2 shows the resulting state poset. In these figures, black circles are events and white squares are states. Arrows represent messages in Figure 1, while they represent the relation $\rightsquigarrow$ in Figure 2.

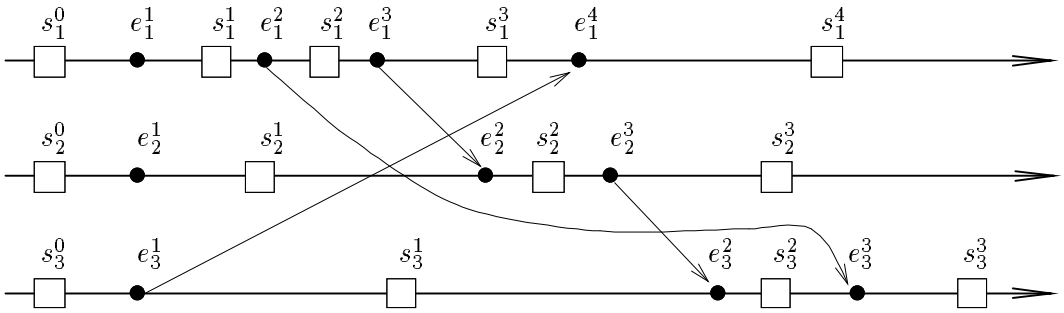

Figure 1: A distributed execution $\tilde{S}$

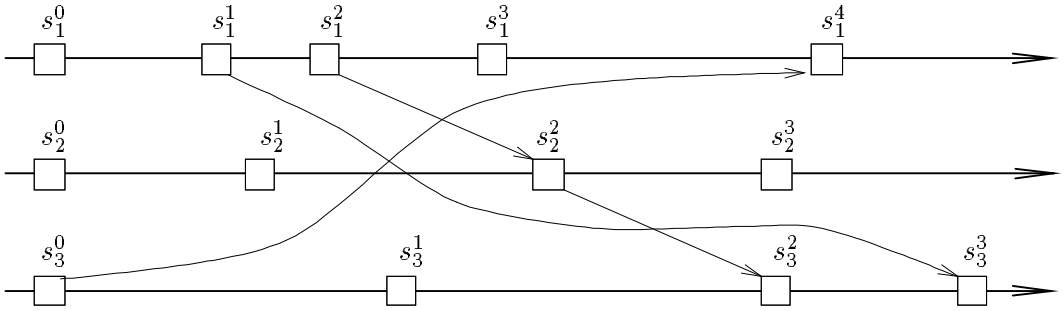

Figure 2: State poset

2.3 Labeled states and Labeled Rooted DAGs

Consider the subposet formed by taking a local state $s \in S$ and all local states which causally precede s. This subposet, called the past (prefix closure) of s and denoted $dag(s)$ forms a rooted directed acyclic graph (DAG) whose root is s (in this DAG edges are directed towards the root):

$$dag(s) \stackrel{\triangle}{=} \tilde{S} \quad \text{restricted to} \quad \{s\} \cup \{s' \mid s' \rightarrow s\}$$

Let A be a set of labels and let λ be a function from S to 2^A. Then each local state s has a set of labels $\lambda(s)$ associated with it. These labels represent boolean expressions evaluated in the local state s; presence of a label in $\lambda(s)$ means that the associated boolean expression (local predicate) evaluates to true in s. For example, an expression such as $(v \leq 10)$ where v is a variable in s can be associated with a label p; then $p \in \lambda(s)$ if and only if $(v \leq 10)$ in state s. If a local state s satisfies no predicate then $\lambda(s) = \{\}$. According to the properties we want to express several local predicates can be associated with the same label. Since each local state has labels, we call these structures labeled rooted DAGs, or LRDAGs.

3 LRDAG Logic

This logic allows one to specify, for any local state $s \in S$, properties as formulas on the associated labeled $dag(s)$. The kind of properties that can be expressed are very general and includes invariance, existence, sequential and non-sequential properties. Section 4 gives examples illustrating this logic. (A sequential property can be expressed as a set of words on some alphabet, the labels; a non-sequential property is more powerful since it can be on several paths of $dag(s)$ at the same time.)

3.1 Syntax

In the following syntax definition A is a set of labels and X is a set of logic variables whose purpose will be described later.

$$\begin{array}{lcl} p & \in & \mathrm{A} \\ x & \in & \mathrm{X} \\ f & = & p \mid \Diamond_l\, x \mid \Diamond_m\, x \mid (\neg f) \mid (f \wedge f) \end{array}$$

This syntax can be easily understood by noticing that a formula f is syntactically correct if and only if f is a boolean expression over the following set:

$$\mathrm{A} \cup \{\Diamond_l\, x \mid x \in \mathrm{X}\} \cup \{\Diamond_m\, x \mid x \in \mathrm{X}\}$$

Define $B \stackrel{\triangle}{=} |\mathrm{X}|$ and $\mathrm{X} \stackrel{\triangle}{=} \{x_1, x_2 \ldots x_B\}$. Then a property is defined as a set of B equations which define each logic variable in X (f_i are formulas).

$$\begin{aligned} x_1 &:= f_1 \\ &\vdots \\ x_B &:= f_B \end{aligned}$$

3.2 Semantics

The forms x_j exist so that we can name formulas in an equation. This allows recursion such as: $x_1 := p \wedge \Diamond_l\, x_1$. We call x_j a logic variable (it is a variable of the detection algorithm, not the underlying computation). The logic variable x_j is true in some state $s \in S$ if and only if formula f_j is true in s. This can be stated formally as follows:

$$s \models x_j \quad \stackrel{\triangle}{=} \quad s \models f_j$$

$\Diamond_l$ and $\Diamond_m$ are temporal operators which provide the power of this logic. In a state s, $\Diamond_l\, x_j$ means that x_j is true in the local predecessor of s, and $\Diamond_m\, x_j$ means that x_j is true in the remote predecessor of s. Note that in an initial state, both forms are false since there are no predecessors; and for states in which the preceding event is not a receive event, $\Diamond_m\, x_j$ is *false*.

$$\begin{aligned} s \models \Diamond_l\, x_j &\quad \stackrel{\triangle}{=} \quad (\exists s' : s' \prec s : s' \models x_j) \\ s \models \Diamond_m\, x_j &\quad \stackrel{\triangle}{=} \quad (\exists s' : s' \rightsquigarrow s : s' \models x_j) \end{aligned}$$

Recall that a label p represents a boolean expression evaluated on some state in S, and that $p \in \lambda(s)$ means that the boolean expression is true in s.

$$s \models p \quad \stackrel{\triangle}{=} \quad p \in \lambda(s)$$

The remaining semantic definitions are straightforward.

$$s \models (f \ \wedge \ f') \quad \triangleq \quad (s \models f) \ \wedge \ (s \models f')$$
$$s \models (\neg f) \quad \triangleq \quad \neg(s \models f)$$

3.3 Predefined predicates

It is useful to define predicates on local states whose truth values depend on the position of the local state s in $\tilde{S}$. These built-in predicates can be used to specify properties which take into account the structure of $\tilde{S}$. The predicate labeled *initial* is true only in initial states, and the predicate labeled *receive* is true only if the preceding event is a message receive.

$$s \models \mathit{initial} \quad \triangleq \quad \neg(\exists s' :: s' \prec s)$$
$$s \models \mathit{receive} \quad \triangleq \quad (\exists s' :: s' \rightsquigarrow s)$$

Moreover, let a predicate labeled *send* be true only if the immediately preceding event was the sending of a message. We can take advantage of other boolean operators (such as $\Rightarrow$ and $\vee$) since they can be expressed in terms of $\wedge$ and $\neg$. Finally, let a predicate labeled *external* be defined as *send or receive*; in other words:

$$s \models \mathit{external} \ \triangleq \ (s \models \mathit{send}) \vee (s \models \mathit{receive})$$

4 Examples

This section gives examples demonstrating the power and flexibility of LRDAG logic.

4.1 Invariance and Existence in the Local Past

If x_1 is defined as:

$$x_1 \ := \ p \ \wedge \ (\neg \mathit{initial} \Rightarrow \Diamond_l \, x_1)$$

then x_1 is true in s_i^k iff $s_i^k \models (p \ \wedge \ (\mathit{initial} \ \vee \ \Diamond_l \, x_1))$. If s_i^k is an initial state, then x_1 is true iff p is true. If s_i^k is not an initial state, then x_1 is true iff p is true and x_1

is true in the locally preceding state. It follows, by induction, that x_1 is true in s_i^k iff p is true in s_i^x for all $0 \leq x \leq k$.

In a similar way, existence (as opposed to invariance) of p in the local past of s_i^k is specified by the following formula:

$$x_1 \;:=\; p \vee \Diamond_l\, x_1$$

4.2 Invariance and Existence in the Global Past

The global past of a local state s is the set $\{s' \mid s' \in dag(s)\}$. A label p is invariant in the global past of s iff $p \in \lambda(s')$ for all $s' \in dag(s)$. This can be expressed in a way similar to invariance in the local past:

$$x_1 \;:=\; p \wedge (\neg initial \Rightarrow \Diamond_l\, x_1) \wedge (receive \Rightarrow \Diamond_m\, x_1)$$

In this case, x_1 is true in non-initial state s iff p is true in s, x_1 is true in the local predecessor, and x_1 is true in the remote predecessor if a message has just been received.

If x_1 is defined in the following way:

$$x_1 \;:=\; p \vee \Diamond_l\, x_1 \vee \Diamond_m\, x_1$$

then $s \models x_1$ iff p is true in some state in $dag(s)$.

4.3 Interval Abstraction

It is sometimes useful to consider a distributed execution modeled as a poset of intervals [16] instead of a poset of local states. Recall *external* is true iff the preceding event was a send or receive. Consider the sequence of states in process $P_i : (s_i^0 s_i^1 \ldots s_i^n)$. This sequence is partitioned into subsequences by external events. These subsequences are *intervals*. For example, in Figure 1, the states of P_2 are partitioned into three intervals: $\{s_2^0, s_2^1\}$, $\{s_2^2\}$, and $\{s_2^3\}$.

Consider the following pattern specification:

$$x_1 \;:=\; p \vee (\neg external \wedge \Diamond_l\, x_1)$$

x_1 is true in s iff p is true in at least one local state since the previous external event. The label *external* resets x_1 each time a new interval begins.

4.4 Regular Expressions[1]

A sequential property is defined as a language (set of strings) on some finite alphabet (the labels). For a local state s, the property is satisfied if one of labeled paths of $dag(s)$ belongs to the language. The set of all labeled paths of $dag(s)$ is denoted $STRINGS(s)$. Formally, the string $(\alpha^0\alpha^1 \ldots \alpha^n)$ is in $STRINGS(s)$ if and only if there exists a sequence of states $(\sigma^0\sigma^1 \ldots \sigma^n)$ such that:

1. $(\exists i :: \sigma^0 = s_i^0)$ (i.e., σ^0 is an initial state.)
2. **for** $(0 \leq k < n), \sigma^k \prec \sigma^{k+1} \ \vee \ \sigma^k \rightsquigarrow \sigma^{k+1}$
3. $\sigma^n = s$
4. **for** $(0 \leq k \leq n)$, **if** $\lambda(\sigma^k) \neq \{\}$ **then**
   $\alpha^k \in \lambda(\sigma^k)$
   else
   $\alpha^k = \epsilon$
   fi

We consider here sequential properties defined by regular expressions, or equivalently, by a finite state automaton M. Given a state s, we can specify and determine if there exists a string in $STRINGS(s)$ which is accepted by M. Linked predicates [14], atomic sequences of predicates [11] an regular patterns [8] are special cases of sequential properties that can be described by the LRDAG logic.

A non deterministic finite state machine M is defined by a tuple:

$$M = (Q, \mathrm{A}, q_1, Q_F, \delta) \text{ with } \begin{cases} \text{set of states: } Q = \{q_1, \cdots, q_B\} \ (\mid Q \mid = B) \\ \text{A: set of input symbols (labels)} \\ q_1 \in Q & \text{(initial state)} \\ Q_F \subseteq Q & \text{(set of final states)} \\ \delta : Q \times \mathrm{A} \mapsto 2^Q & \text{(transition function)} \end{cases}$$

Such an automaton recognizes a set of strings on A that can be specified in LRDAG logic by a set of B equations defining x_j for $1 \leq j \leq B$, such that x_j is true in s iff there exists a string in $STRINGS(s)$ which would place M in state q_j.

[1] Two kinds of local states are considered in this section: those of the state machine and those of the distributed execution. Context should clarify which type of state we are referring to when we use the term "state".

Let $\Diamond\, x_k$ be a short form for $(\Diamond_l\, x_k \ \vee\ \Diamond_m\, x_k)$[2]. Let

$$T_j = \{(\alpha \ \wedge\ \Diamond\, x_k) \mid q_j \in \delta(q_k, \alpha) \wedge \alpha \in A \wedge k \in \{1, \cdots, B\}\}$$

(T_j represents all transitions of the automaton entering q_j). The B equations are defined in the following way:

$$x_1 := \mathit{initial} \ \vee\ f_1$$

$$x_j := f_j \qquad \forall j \in \{2, \cdots, B\}$$

with

$$f_j = \bigvee_{t \in T_j} t$$

f_j has the form $(f_j^1 \ \vee\ f_j^2 \ \vee\ f_j^3 \ldots)$. When we consider a pictorial representation of the state machine each arrow pointing to q_j has a label and defines one of the disjuncts in f_j. As an example, consider an arrow incident on q_j and suppose it defines f_j^1. Let q_k be the state on the other end of the arrow (i.e. we are considering the edge (q_k, q_j)). Let α be the label on this edge. Then $f_j^1 = \alpha \ \wedge\ \Diamond\, x_k$. Thus f_j^1 is true in s if the state machine could have traveled edge (q_k, q_j) in the previous step.

The following example illustrates this construction by considering a state machine (Figure 3) implementing the regular expression $a + cb^*c$ (q_3 is the only final state).

First consider f_1. No arrows enter q_1. Therefore the disjunct list is empty and $f_1 = false$.

Next consider f_2. One incoming arrow is labeled c and comes from state q_1. Thus one disjunct is $c \ \wedge\ \Diamond\, x_1$. The other incoming arrow is labeled b and comes from q_2. Thus another disjunct is $b \ \wedge\ \Diamond\, x_2$. Thus $f_2 = (c \ \wedge\ \Diamond\, x_1) \ \vee\ (b \ \wedge\ \Diamond\, x_2)$.

Next consider f_3. There is an incoming arrow from q_1 labeled a, and another one from q_2 labeled c. Thus $f_3 = (a \ \wedge\ \Diamond\, x_1) \ \vee\ (c \ \wedge\ \Diamond\, x_2)$.

Therefore, in our logic, this regular expression can be specified as follows:

$$\begin{aligned} x_1 &:= \mathit{initial} \\ x_2 &:= (c \ \wedge\ \Diamond\, x_1) \ \vee\ (b \ \wedge\ \Diamond\, x_2) \\ x_3 &:= (a \ \wedge\ \Diamond\, x_1) \ \vee\ (c \ \wedge\ \Diamond\, x_2) \end{aligned}$$

[2] As we can see a sequential property does not distinguish between local and remote predecessors of local states.

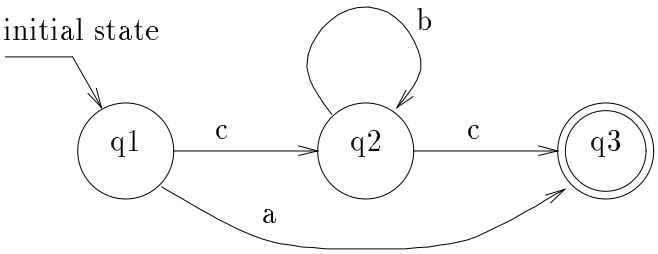

Figure 3: State machine implementing $a + cb^*c$

x_j is true in poset state s iff there is a string in $STRINGS(s)$ which would place M in state q_j. Thus, x_3 is true in s iff there is a string which matches the regular expression $a + cb^*c$.

4.5 Non-sequential properties

The previous examples demonstrated how LRDAG logic can express sequential properties of control flows. A non-sequential property cannot be expressed as a set of independent words on some alphabet. Such a property is on several control flows at the same time.

A non-sequential control flow can be demonstrated with the scatter and collect operations which are commonly used to distribute a workload and collect the results. Suppose there is a matrix D partitioned into N submatrices D_i, for $1 \leq i \leq N$. The matrix, which is initially owned by P_1, is distributed among the processes so that P_i owns D_i. After process P_i performs some operation on D_i, P_1 collects the results and then owns the entire matrix once again. Let $i.owns.D_j$ and $1.owns.D$ be labels such that:

$$\begin{aligned} i.owns.D_j \in \lambda(s) &\iff s \in S_i \ \wedge \ s \text{ owns } D_j \\ 1.owns.D \in \lambda(s) &\iff s \in S_1 \ \wedge \ s \text{ owns the entire matrix } D \end{aligned}$$

The scatter-and-collect control flow is then characterized by the following pattern specification. We use y_1 and z_1 as logic variables in addition to x_i, and $\diamond$ is the same short form used in section 4.4 (a local state of P_i satisfies x_i just after the

scattering, a local state of P_1 satisfies z_1 just after the collection of the results):

$$\begin{aligned} y_1 &:= 1.owns.D \\ x_1 &:= 1.owns.D_1 \wedge \Diamond\, y_1 \\ x_2 &:= 2.owns.D_2 \wedge \Diamond\, y_1 \\ &\vdots \\ x_N &:= N.owns.D_N \wedge \Diamond\, y_1 \\ z_1 &:= 1.owns.D \wedge (\forall i : 1 \leq i \leq N : \Diamond\, x_l) \end{aligned}$$

In this pattern specification, z_1 is true in P_1 when it owns D after the scatter-and-collect operation.

5 Decentralized Detection Algorithm

In this section there are algorithm variables and logic variables. A logic variable is still referred to as x_j and the corresponding algorithm variable is named X_j. The variables X_j^m and X_j^l are algorithm variables which store the values of $\Diamond_m\, x_j$ and $\Diamond_l\, x_j$ respectively.

5.1 Description of the Algorithm

Given a property definition we want to evaluate each logic variable x_j in any state s. In the property definition, $x_j := f_j$. Recall that f_j is a boolean expression over the set

$$\mathrm{A} \cup \{\Diamond_l\, x \mid x \in \mathrm{X}\} \cup \{\Diamond_m\, x \mid x \in \mathrm{X}\}$$

Recall that a label p is an element of $\lambda(s)$ if and only if the local predicate which p represents evaluates to true in state s. The form $\Diamond_l\, x_j$ represents the value of logic variable x_j in the local predecessor. For initial states, $\Diamond_l\, x_j$ is false. In states where a message has just been received, $\Diamond_m\, x_j$ represents the value of logic variable x_j in the state that sent the message. If no message has just been received, then $\Diamond_m\, x_j$ is false.

Each process P_i is augmented with boolean variables $X_1, X_2, \ldots X_B$; X_j is the concrete representation of the logic variable x_j. In any state, the values of x_j in predecessor states must be known. Message tags are used to carry the values of x_j to the receiving process so that the forms $\Diamond_m\, x_j$ can be evaluated (their values are kept in local variables X_j^m of the receiving process). The forms $\Diamond_l\, x_j$ can be

evaluated easily since they are from the same process (their values are kept in local variables X_j^l). The algorithm uses a macro $eval_j$ such that:

$$eval_j(\Diamond_l\, x_1,\ \Diamond_l\, x_2,\ \ldots\ \Diamond_l\, x_B,\ \ \Diamond_m\, x_1,\ \Diamond_m\, x_2,\ \ldots\ \Diamond_m\, x_B)$$

expands to f_j. For example, consider the logic variables x_1, x_2 and x_3 from the regular expression $a + cb^*c$ discussed in Section 4.4:

$$\begin{aligned}
eval_1(\Diamond_l\, x_1, \Diamond_l\, x_2, \Diamond_l\, x_3, \Diamond_m\, x_1, \Diamond_m\, x_2, \Diamond_m\, x_3) &:= \mathit{initial} \\
eval_2(\Diamond_l\, x_1, \Diamond_l\, x_2, \Diamond_l\, x_3, \Diamond_m\, x_1, \Diamond_m\, x_2, \Diamond_m\, x_3) &:= (c \wedge \Diamond\, x_1) \vee (b \wedge \Diamond\, x_2) \\
eval_3(\Diamond_l\, x_1, \Diamond_l\, x_2, \Diamond_l\, x_3, \Diamond_m\, x_1, \Diamond_m\, x_2, \Diamond_m\, x_3) &:= (a \wedge \Diamond\, x_1) \vee (c \wedge \Diamond\, x_2)
\end{aligned}$$

A formal description of the algorithm follows:

Local Variables

boolean: X_j, X_j^l, X_j^m **for** $j \in 1 \ldots B$;

Initially

(S1) **for** $j := 1$ **to** B **do**
$X_j := eval_j(\mathit{false}, \ldots \mathit{false})$;

Upon sending a message

(S2) Tag message with $(X_1, X_2, \ldots X_B)$;

Upon entering new local state s

(S3) **if** (previous event was message receive) **then**
$(X_1^m, X_2^m, \ldots X_B^m) :=$ message tag
else
$(X_1^m, X_2^m, \ldots X_B^m) := (\mathit{false}, \ldots, \mathit{false})$
fi;

(S4) $(X_1^l, X_2^l, \ldots X_B^l) := (X_1, X_2, \ldots, X_B)$

(S5) % evaluation of each f_j in state s%
for $j := 1$ **to** B **do**
$X_j := eval_j(X_1^l, X_2^l, \ldots X_B^l, X_1^m, X_2^m, \ldots X_B^m)$;

The above algorithm has storage overhead proportional to B bits per process. No additional messages are introduced by the algorithm, however each message produced by the underlying computation is tagged with B bits. The time complexity is also proportional to B operations per event per process. In an actual implementation, only states deemed relevant to the behavior being detected would be interrupted. In most cases this would drastically reduce the time complexity.

5.2 Correctness Proof

The proof consists in showing the following equivalence, where s is any state in $\tilde{\mathrm{S}}$. (Recall that the term "state" always refers to a local state).

$$(\mathrm{P}) \quad s \models x_j \iff \text{Local variable } X_j \text{ is true in state } s$$

The implication "$\Leftarrow$" states that detection is sound (safety) and the implication "$\Rightarrow$" states that detection is complete (liveness).

Proof: The proof is done by induction on the rank of the state s in the poset $\tilde{\mathrm{S}}$. Let $rank(s)$ be the length of the longest path in $dag(s)$ from some initial state to s.

1. Base case.
   For all s such that $rank(s) = 0$, s is an initial state. Statement $S1$ ensures P is true for initial states.

2. Induction case.
   Assume P is true for all states v such that $rank(v) \leq k$ and consider a state s in P_i such that $rank(s) = k + 1$. First we show that X^l and X^m have correct values.

   - s has one local predecessor s_l whose rank is less than k. By the induction hypothesis and by statement $S4$:

     $$s_l \models x_j \iff X_j^l \text{ is true in state } s$$

   - If a message is received just before state s then s has one remote predecessor, s_m whose rank is less than k. Thus by the induction hypothesis and by statement $S3$:

     $$s_m \models x_j \iff X_j^m \text{ is true in state } s$$

     If no message is received then s has no remote predecessor and by $S3$ each variable X_j^m is false in state s.

Since X^l and X^m have correct values, and by definition of $eval_j$, we know that statement $S5$ computes new values for X_j^m such that:

$$s \models x_j \iff X_j \text{ is true in state } s$$

□

6 Conclusion

A simple but powerful logic to express a wide class of properties of control flows of distributed computations has been presented. This class includes sequential properties (of which linked predicates [14], atomic sequences of predicates [11] and regular patterns [8] are special cases) and more sophisticated non-sequential properties. These properties, expressed as formulas on the global past of local states, are useful for analyzing, testing or debugging executions of distributed programs.

A decentralized algorithm that detects the properties, concurrently with the underlying computation, has been presented and proved correct. This algorithm is surprisingly simple despite the power of the logic. The algorithm does not alter the control flows or the causal structure of the computation (so it does not inhibit or initiate sending or receiving of messages) and it is efficient in the sense it uses message tags of B bits where B depends only on the property being detected (and so is independent on the number of processes). Algorithms already proposed to detect sequential properties on control flows [14, 11, 8] are particular instances of this algorithm which additionally can detect more sophisticated properties.

An implementation of this algorithm is currently being developed in the context of a debugging facility for distributed programs [10].

References

[1] Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems: fundamental concepts and mechanisms, in Distributed Systems*, chapter 4, pages 55–93. *ACM Press, Frontier Series*, (S.J. Mullender Ed.), 1993.

[2] Ö. Babaoğlu and M. Raynal. Specification and detection of behavioral patterns in distributed computations. In *Proc. of 4th IFIP WG 10.4 Int. Conference on Dependable Computing for Critical Applications*, Springer Verlag Series in Dependable Computing, San Diego, January 1994.

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

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Toplas*, 8(2):244–263, 1986.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, Santa Cruz, California, May 1991.

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

[7] E. Fromentin and M. Raynal. Inevitable global states: a new concept to detect unstable properties of distributed computations in an observer independent way. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Oct. 1994.

[8] 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* 23^{rd} *International Conference on Parallel Processing*, 2:73–76, St. Charles, IL, August 1994.

[9] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Twelfth International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Springer Verlag, LNCS 625, New Delhi, India, December 1992.

[10] M. Hurfin, N. Plouzeau, and M. Raynal. A debugging tool for distributed Estelle programs. *Journal of Computer Communications*, 16(5):328–333, May 1993.

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

[12] C. Jard, T. Jeron, G.V. Jourdan, and J.X. Rampon. A general approach to trace-checking in distributed computing systems. In *Proc. 14th IEEE Int. Conf. on DCS*, Poznan, Poland, pp. 396–403, June 1994.

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

[14] 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*, pages 316–323, July 1988.

[15] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations : in search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[16] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. (Reprinted in SIGPLAN Notices, Dec. 1993).

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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