Observation of Software for Distributed Systems with RCL *

Alexander I. Tomlinson and Vijay K. Garg

email: {alext,vijay}@pine.ece.utexas.edu
homepage: http://maple.ece.utexas.edu/
Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas 78712

Abstract. Program observation involves formulating a query about the behavior of a program and then observing the program as it executes in order to determine the result of the query. Observation is used in software development to track down bugs and clarify understanding of a program's behavior, and in software testing to ensure that a program behaves as expected for a given input set. RCL is a recursive logic built upon conjunctive global predicates. Computational structures of common paradigms such as butterfly synchronization and distributed consensus can be expressed easily in RCL. A nonintrusive decentralized algorithm for detecting RCL predicates is developed and proven correct.

1 Introduction

Posets have a recursive structure that has not been exploited much in research on observation predicates. This paper presents a poset predicate logic which exploits this recursive structure. The result, RCL, is a logic which is simple yet powerful. Recursive logics are intuitive because there are fewer constructs and rules to remember. They are powerful because the full power of the logic is available at each level of recursion. Boolean logic is an example of a recursive logic: it is simple, elegant and powerful.

RCL is based upon *conjunctive global predicates* (CGP). A CGP is defined to be a conjunction of local predicates. For example, let l_i be a predicate on the local state of process i. Then we can define a CGP g to be $l_1 \wedge l_2 \wedge l_3$. All CGPs, including g, are evaluated on global states. For example if c is a global state containing local states $\sigma_1, \sigma_2, \sigma_3$, then g is true in global state c if and only if l_i is true in local state l_i, for $1 \leq i \leq 3$.

An RCL formula takes a set of CGPs and specifies a pattern in which the individual CGPs must occur in a computation in order for the formula to be "true" in that computation. Some patterns which can be specified with RCL include butterfly synchronization, data collection, and distributed consensus. These examples are demonstrated in this paper.

* Research supported in part by NSF Grant CCR 9110605, TRW faculty assistantship award, GM faculty fellowship, a grant from IBM, and an MCD University Fellowship.

We begin with a review of related work, and then continue with a description of the computation model and notation. Then we define the syntax and semantics of RCL, and give examples of RCL formulas and how they can be applied in the observation of distributed programs. We present a distributed algorithm for online detection of RCL formulas and prove its correctness. The algorithm is based on existing algorithms for detecting CGPs, which are not considered in detail in this paper. Due to space constraints, the complexity of the algorithm is not discussed in detail. See [19] for a complete analysis of the algorithm's complexity.

2 Related Work

There has been much work in observing unstable global states of distributed computations. Babaoğlu and Marzullo [1] and Schwartz and Mattern [18] both survey recent work on detecting consistent global states in a distributed system. Recently, Chase and Garg [2] have shown that global predicate detection is an NP-complete problem. In that paper they define the property of *linearity* and show that there exists a polynomial detection algorithm for any linear predicate. CGP (discussed later) is an example of a linear predicate.

Cooper and Marzullo [4] present algorithms for online detection of three types of global predicates. The first type is "global predicate g was *possibly* true at some point in the past". The second type is "g was *definitely* true at some point in the past", and the third type is "g is *currently* true". The third type may require delaying certain processes of the execution.

Observation of general global predicates is very expensive. As a result, researchers have devised classes of global predicates which can be efficiently observed. One such class is *relational global predicates* as described by Tomlinson and Garg [20].

Another such class is *conjunctive global predicates* (CGP) as described by Garg and Waldecker [10]. Garg and Waldecker present strong and weak [11, 10, 12] forms of CGP which correspond to *possibly* and *definitely* of Cooper and Marzullo [4]. The *weak CGP* is true in a computation if there exists a global state in the computation which satisfies the CGP. The *strong CGP* is true in a computation if all runs consistent with the computation must enter a global state in which the CGP is true.

Some researchers have taken the idea of conjunctive global predicates (CGP) and extended them to form poset predicates. One can define an ordering relation on global states and then define a sequence of CGP. There have been several approaches to this that differ primarily in their definition of the ordering on global states.

Chiou and Korfage [3] define *event normal form* predicates which are sequences of CGP. In their sequencing relation, global state a precedes global state b if and only if each local state in a happens-before all local states in b.

Haban and Weigel [13] gave an early attempt to define poset predicates with recursive structure. They used local events (essentially the same as local predicates) as primitives and build global events from them with a set of binary relations that include alternation, conjunction, happens-before, and concurrency. All events (global and local) have vector timestamps which are used to determine if two events are related according to one of the four relations. The new global event inherits a timestamp from one of the constituent events. For example, consider alternation: if $e_1 \mid e_2$ is a global event which is said to occur whenever either e_1 or e_2 occurs. The event $e_1 \mid e_2$ inherits the vector time of whichever event actually occurred (i.e., either e_1 or e_2). Even though their definitions lead to ambiguities (resulting from timestamp inheritance) as demonstrated in [14], the work was noteworthy in that it was an early attempt to develop a recursive poset predicate logic.

The above systems are based on global predicates, but many systems have been designed around the local predicate too. One of the early works in this area was Miller and Choi's *sequence of local predicates* [17]. These are an ordered list of local predicates $p_1, \ldots p_k$. This predicate is true in an execution if and only if there exists a sequence of local states $\sigma_1, \ldots \sigma_k$ (sequenced by Lamport's happens-before relation) such that local predicate p_i is true in local state σ_i.

Hurfin, Plouzeau and Raynal [15] extended the sequence of local predicates to the *atomic sequence of local predicates*. In this class, occurrences of local predicates can be forbidden between adjacent predicates in a sequence of local predicates. The example given above for linked predicates could be expanded to include: "local predicate r_i never occurs in between local predicates p_i and p_{i+1}". Each local predicate can belong to a different process in the computation.

Fromentin, Raynal, Garg and Tomlinson [6] developed *regular patterns*, which are based upon regular expressions. A regular pattern is specified by a regular expression of local predicates. For example pq^*r is true in a computation if there exists a sequence of consecutive local states $(s_1, s_2, \ldots, s_n)$ such that p is true in s_1, q is true in $s_2, \ldots, s_{n-1}$, and r is true in s_n. Note that the states in the sequence need not belong to the same process – two states are consecutive if they are adjacent in the same process or one sends a message and the other receives it. In [9], the same authors extend regular patterns to allow patterns on directed acyclic graphs instead of just strings.

3 Model and Notation

We use the following notation for quantified expressions:

$$(\text{ op free\_var\_list : range\_of\_free\_vars : expr })$$

For example, $(\forall i : 0 \leq i \leq 10 : i^2 \leq 100)$ means that for all i such that $0 \leq i \leq 10$, we know that $i^2 \leq 100$. The operator "op" need not be restricted to universal or existential quantification. It can be any commutative associative operator (e.g., $min, \cup, +$). For example, if S_i is a finite set, then $(+u : u \in S_i : 1)$ equals the cardinality of S_i.

Any distributed computation can be modeled as a decomposed partially ordered set (deposet) of process states [5]. A deposet is a partially ordered set $(P, \rightsquigarrow)$ such that:

1. P is partitioned into N sets P_i, $1 \leq i \leq N$.
2. Each set P_i is a total order under some relation $\prec_{im}$.
3. $\prec_{im}$ does not relate two elements which are in different partitions.
4. Let $\rightarrow$ be the transitive closure of $\prec_{im} \cup \rightsquigarrow$. Then $(P, \rightarrow)$ is an irreflexive partial order.

An execution that consists of N processes can be modeled by a deposet where P_i is the set of local states at process i which are sequenced by $\prec_{im}$ the $\rightsquigarrow$ relation represents the ordering induced by messages; and $\rightarrow$ is Lamport's *happens before* relation[16]. For convenience, we use P_i to represent two quantities: the set of local states at process i (as it was defined), and the process i itself. Similarly, we use P to denote both the set of all local states and the set of all processes.

The concurrency relation on P is defined as $u \| v \;=\; (u \not\rightarrow v) \wedge (v \not\rightarrow u)$. $\preceq$ denotes the reflexive transitive closure of $\prec_{im}$. For convenience, $s.next = t$ and $t.prev = s$ whenever $s \prec_{im} t$.

A global state is a subset $c \subseteq P$ such that no two elements of c are ordered by $\rightarrow$. We define $\overline{P}$ to be the set of all global states in $(P, \rightarrow)$. We also use the terms "cut" and "antichain" to refer to an element of $\overline{P}$. A "chain" is a set of states which are totally ordered by $\rightarrow$. For example, each set P_i is a chain.

All formulas in RCL are evaluated on closed posets. Evaluating a formula on a poset which is not closed is not a defined operation. A poset P is closed if and only if every state which is ordered in between two elements of P is also in P. Another way of saying this is that P is closed if and only if its prefix-closure intersected with its suffix-closure equals P. Prefix and suffix closure of a poset A are denoted by $\overleftarrow{A}$ and $\overrightarrow{A}$.

$$\begin{aligned}
\overleftarrow{A} &\triangleq \{x \mid (\exists y : y \in A : x \rightarrow y \vee x = y)\} \\
\overrightarrow{A} &\triangleq \{x \mid (\exists y : y \in A : y \rightarrow x \vee x = y)\} \\
\mathit{closed}(A) &\triangleq A = (\overleftarrow{A} \cap \overrightarrow{A})
\end{aligned}$$

We define another closure operation which performs closure between any two subposets of a poset. For example, $[A..B]$ is the poset which includes posets A and B and all in between local states. Usually, A and B are cuts, but it is convenient to use the more general definition that they are subposets. This allows us to say, for example, that $[c..P]$ is the set of all states in or after cut c but still in poset P. We also define $(A..B)$ to be an open-ended version of $[A..B]$.

$$\begin{aligned}
[A..B] &\triangleq \overrightarrow{A} \cap \overleftarrow{B} \\
(A..B) &\triangleq \overrightarrow{A} \cap \overleftarrow{B} - (A \cup B)
\end{aligned}$$

The *cutset* of a poset P and a formula f is the set of all cuts c of P such that $[P..c]$ satisfies f. The expression $\Psi(P, f)$ refers to this set and is defined as follows:

$$\Psi(P, f) \triangleq \{c \mid c \in \overline{P} \wedge [P..c] \models f\}$$

Cutsets will be used to prove the correctness of the RCL detection algorithm. It turns out that cutsets are lattices. The correctness proof shows that, given a computation P and a formula f, the detection algorithm returns the infimum of $\Psi(P, f)$, which is the unique first cut c of P such that $[P..c]$ satisfies f.

We also define two ordering relations on subposets: weak and strong precedes. The ordering relations are usually used on cuts, but the more general relation suffices. Subposet A weakly precedes subposet B if and only if B is entirely contained within the suffix closure of A and they have no elements in common.

$$A \prec B \;\stackrel{\triangle}{=}\; B \subseteq \overrightarrow{A} \;\wedge A \cap B = \emptyset$$

Strong precedes requires not only that B must be contained in the suffix closure of A, but also that each element in A must happen-before every element in B. Clearly, this implies that A weakly precedes B as well. Strong precedes corresponds to barrier synchronization: there is a barrier synchronization between A and B if and only if A strongly precedes B. It is defined as follows:

$$A \prec\!\!\prec B \;\stackrel{\triangle}{=}\; (\forall a, b : a \in A \wedge b \in B : a \to b)$$

4 Syntax and Semantics

A formula in RCL is evaluated on a poset. One can think of a formula as a boolean function whose argument is a poset. The rules for constructing well formed formulas are given by the syntactic definitions shown below:

$$\begin{aligned} f &= S \;\mid\; f \triangle f \\ S &= g \;\mid\; g\langle f\rangle S \;\mid\; g\langle\!\langle f\rangle\!\rangle S \end{aligned}$$

The basic component of a formula is a conjunctive global predicate (CGP), which is represented by the terminal symbol g. The symbol S is a sequence of CGP formulas. The symbol f is a conjunction of these sequences, and the $\triangle$ operator is similar to boolean AND operator.

When S is fully expanded, it has the form $g\langle f\rangle g\langle f\rangle g \ldots g\langle f\rangle g$. When such a sequence is true on a poset, then each g corresponds to an antichain. The regions in between these antichains are subposets upon which the f's in the sequence are evaluated. This is explained in more detail in the section on semantics.

The symbol 'g' represents any global state based predicate which meets certain requirements. Mathematically, g is a set containing exactly those antichains upon which the predicate (that g represents) is true. One of these requirements is that this set forms a lattice.

Another requirement is that the antichains in g cannot contain any extraneous local states. For example, if g represents a conjunctive global predicate with components at process 1 and 2, then each antichain in g can only contain local states from these two processes. No others are needed to evaluate the predicate g. The reason for this requirement is to ensure the proper interpretation of sequences of g's – the ordering should be based on the necessary states only.

It is clear from the syntax that g is a valid RCL formula. The truth of such a formula is determined by the following rule:

$$P \models g \triangleq \mathit{closed}(P) \wedge (g \cap \overline{P} \neq \emptyset)$$

This rule states that g is true in P if and only if P is closed and some antichain in g is also in P. This is similar to saying that there exists a global state in P in which the global predicate g is true.

The $\triangle$ operator is essentially the same as the boolean AND operator. We use $\triangle$ in order to avoid confusion with its boolean counterpart. This is especially useful in proofs where the two operators frequently appear in the same expression.

$$P \models f_1 \triangle f_2 \triangleq (P \models f_1) \wedge (P \models f_2)$$

The last two rules are the heart of RCL. They show how to evaluate a recursive formula. The only difference between them is the ordering between the cuts.

$$P \models g\langle f \rangle S \triangleq (\exists a, b : a, b \in \overline{P} : a \prec b \wedge a \models g \wedge (a..b) \models f \wedge [b..P] \models S)$$

The strong-precedes counter part to the above formula is:

$$P \models g\langle\langle f \rangle\rangle S \triangleq (\exists a, b : a, b \in \overline{P} : a \prec\!\!\prec b \wedge a \models g \wedge (a..b) \models f \wedge [b..P] \models S)$$

Now consider the following formula:

$$g_1\langle f_2 \rangle g_2 \langle f_3 \rangle g_3 \ldots \langle f_n \rangle g_n$$

This formula holds on a poset P if and only if there exist cuts a_i in P such that $a_{i-1} \prec a_i$ and $a_i \models g_i$ and $(a_{i-1}..a_i) \models f_i$. Figures 1 and 2 show some examples.

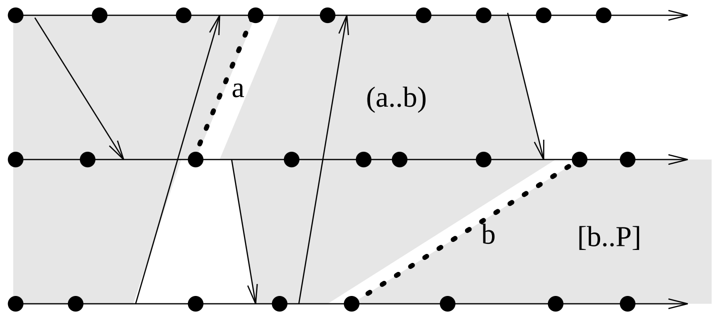

Fig. 1. Example of a poset structure which could satisfy $g\langle f \rangle g$. The cuts a and b divide the poset into regions as indicated by the shading. Notice that each region is closed.

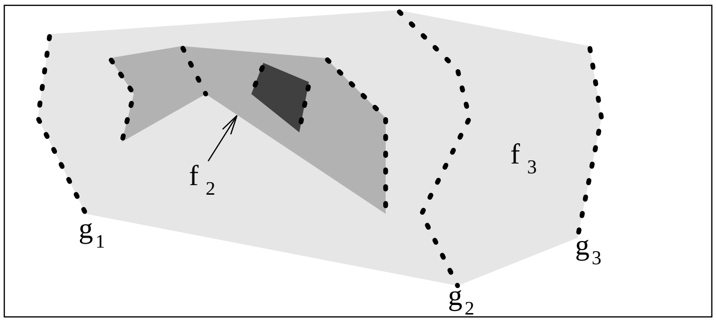

Fig. 2. Example of a structure which might satisfy $g_1\langle f_2\rangle g_3\langle f_4\rangle g_5$. The CGPs, g_i, need not be full width antichains, which is why they are shown in a "free-form" manner. Each f_i is another RCL formula. Two levels of recursion are shown for f_2, where f_2 has the structure $g\langle f\rangle g\langle g\langle f\rangle g\rangle g$.

5 Examples

This section gives examples of some useful RCL formulas. The first three examples show previous debugging logics which are special cases of RCL. In each example, p_i is a predicate on the state of a single process.

Sequence of local predicates: Consider the sequence of local predicates as defined in [17]: $(p_1, p_2, \ldots p_n)$. Each local predicate p_i is a special case of CGP. Therefore, this sequence of local predicates is equivalent to the RCL formula $p_1\langle true\rangle p_2 ... \langle true\rangle p_n$.

Event Normal From (ENF): Chiou and Korfage [3] define *event normal form* predicates which are sequences of CGP. In their sequencing relation, global state a precedes global state b if and only if each local state in a happens-before all local states in b. This ordering is equivalent to the $\prec\!\!\prec$ ordering on cuts. Thus, an ENF formula which consists of a sequence of CGP could be represented in RCL as follows: $g_1\langle\!\langle true\rangle\!\rangle g_2\langle\!\langle true\rangle\!\rangle g_3 ... \langle\!\langle true\rangle\!\rangle g_n$.

Barrier Synchronization: The strong precedes relation is equivalent to barrier synchronization: a barrier synchronization exists between two cuts if and only if they are related by $\prec\!\!\prec$. Suppose two global states could be characterized by the predicates g_1 and g_2. The RCL formula $g_1\langle\!\langle true\rangle\!\rangle g_2$ is true if and only if a barrier synchronization takes place between two cuts which satisfy g_1 and g_2.

Butterfly Synchronization: Butterfly synchronization is an implementation of barrier synchronization. Its structure can be defined recursively. Let X denote a set of process identifiers and let X_l and X_h be a partition of this set into upper and lower halves. $BF(X)$ will be defined to be an RCL formula which is true when there exists a butterfly synchronization between the processes named in X.

$BF(X)$ is the formula *true* if the size of X equals 1. Otherwise, $BF(X)$ equals the formula $g_X\langle\langle BF(X_l) \triangle BF(X_h)\rangle\rangle g_X$. Figure 3 shows an example.

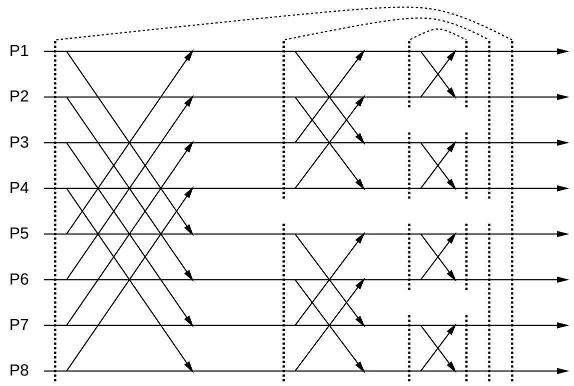

Fig. 3. Butterfly synchronization example. See section 5.

Distributed Consensus: Consider a fixed connection network. A phase consists of a message exchange on each edge. After phase i, each node has data from all nodes within distance i.

Let $g_{\{1,2\}}$ denote a CGP which is true on all antichains which contain exactly one state from each of process 1 and 2. Consider the example shown in figure 4. The communication structure of distributed consensus on the edge between nodes 1 and 2 that network can be captured by the following RCL formula:

$$g_{\{1,2,3,4,5\}}\langle\langle g_{\{1,2,3\}}\langle\langle g_{\{1,2\}}\langle\langle true\rangle\rangle g_{\{1\}}\rangle\rangle g_{\{1\}}\rangle\rangle g_{\{1\}}$$

The innermost form, $g_{\{1,2\}}\langle\langle true\rangle\rangle g_{\{1\}}$, is true after phase 1. The form that surrounds that becomes true after phase 2. The entire formula becomes true after phase 3 at which time consensus is complete since number of phases equals maximum distance between any two nodes.

Data Collection: It is common practice in distributed computing to scatter data among a set of computers, have each computer perform some operation on the data, and then collect the results. The collection phase of this operation can characterized with an RCL formula. Using the notation defined above, the formula is shown below.

$$g_{\{1,2,3,4,5,6,7,8\}}\langle\langle true\rangle\rangle g_{\{2,4,6,8\}}\langle\langle true\rangle\rangle g_{\{4,8\}}\langle\langle true\rangle\rangle g_{\{8\}}$$

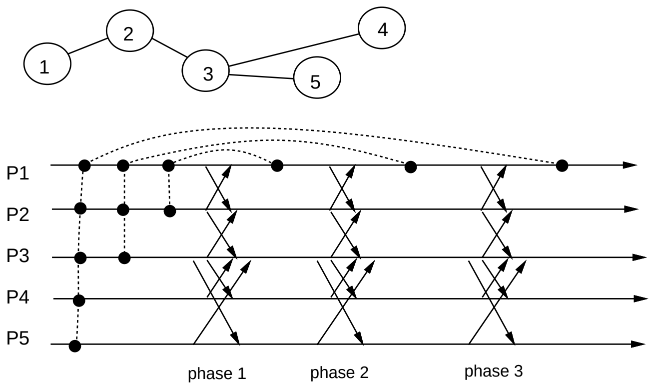

Fig. 4. Distributed consensus example. See section 5.

6 Algorithm

The algorithm is implemented by the function *fc*(). Given a cut a, and a formula f, the function $fc(a, f)$ finds the first cut b such that $[a..b]$ satisfies f. If there is no such cut, then $fc(a, f)$ returns $\top$.

The function $fc(a, f)$ calls three subroutines ($findCGP$, *advance*, and *sup*) as it parses an RCL formula. The pattern of subroutine calls depends on the syntactic structure of the formula.

Function $findCGP$: Function $findCGP$ finds the first cut in a poset that satisfies a given conjunctive global predicate (CGP). An efficient, decentralized, token-based algorithm for detecting CGP appears in [8]. The subroutine $findCGP$ is assumed to be a procedural interface to this algorithm. Thus, any process can call $findCGP$ to spawn the distributed token based algorithm described in [8]. The calling process is blocked until the underlying distributed algorithm completes at which point the result is returned by $findCGP$. Note that the blocked process is part of the RCL detection algorithm, not the underlying computation. In [8], it is shown that $findCGP$ has $O(NM)$ message, time and space complexity, where N is the number of processes, and M is the number of application messages.

Function sup: Function *sup* is takes one or more cuts as input and returns the supremum of those cuts. Cuts are represented as vector clock values, and the *sup* function takes the component-wise maximum of the vector clock values. The *sup* function can be implemented with computational complexity $O(NB)$, where B is the number of cuts input to *sup*, and N is the number of processes.

Function advance(a, b): Function *advance*(a, b) advances cut b until cut a strongly precedes it. It then returns the advanced cut. The *advance* function can be implemented with $O(N^2)$ computational complexity.

The function $fc(a, f)$ takes a cut a and a formula f and returns the first cut b such that $[a..b]$ satisfies f. If there is no such cut, then $fc(a, f)$ returns $\top$. Four definitions of $\Psi(a, f)$ are shown below, one for each syntactic form of f.

$\underline{fc(a, g\langle f\rangle S)}$

$a_1 = fc(a, g);$
$a_2 = fc(a_1, f);$
$a_3 = fc(a_2, S);$
return a_3;

$\underline{fc(a, g\langle\langle f\rangle\rangle S)}$

$a_1 = fc(a, g);$
$a_2 = fc(a_1, f);$
$a_3 = advance(a_1, a_2);$
$a_4 = fc(a_3, S);$
return a_4;

$\underline{fc(a, g)}$

return $findCGP(a, g)$;

$\underline{fc(P, f_1 \triangle f_2)}$

return $sup(fc(P, f_1), fc(P, f_2))$;

The algorithm is recursive and it mirrors the syntax structure of RCL. The recursion always bottoms out in the $fc(a, g)$ function. The poset which is being searched is a global read only structure and is not shown in the above descriptions. Only the subroutines $findCGP$ and $advance$ need access to the poset structure.

The correctness proof consists of showing that $fc(a, f) = inf\, \Psi(P, f)$, where $a = inf\, \overline{P}$. There are several properties of RCL which enable us to prove this. For example, RCL is monotonic with respect to set inclusion. Using this property of monotonicity, it can be shown that a cutset is a lattice. The lattice property allows us to implement $fc(a, g\langle f\rangle S)$ in a greedy fashion: first finding g, then f and finally S.

7 RCL Properties and Algorithm Proof

The logic is monotonic with respect to set inclusion over closed posets. If $P \subseteq R$, R is closed, and P satisfies some formula, then R also satisfies that formula. This definition of monotonicity is more encompassing than other commonly used definitions which use the "advancement of time" as the ordering relation instead of set inclusion. Using set inclusion has the benefit that if a formula is true for a given subcomputation, then it remains true not only when (local) states are added to the end of the subcomputation, but also when states are included from before the computation or even concurrent with it. That is, the poset which represents the computation can "grow" in any direction (as long as it remains closed). Lemma 1 proves that RCL is monotonic.

Lemma 1. *Monotonicity:* $P \subseteq R \wedge closed(R) \wedge P \models f \Rightarrow R \models f$

Proof: Appears in [19]. ■

Given any poset P and formula f, the cutset of (P, f) forms a lattice. Cuts can be represented with vector clocks, and the infimum of two cuts is the component-wise minimum of their vector clocks. The supremum is the component-wise maximum. The same operators are used in cutset lattices.

Lemma 2. *Lattice:* $a, b \in \Psi(P, f) \ \Rightarrow \ inf(a, b) \in \Psi(P, f) \wedge sup(a, b) \in \Psi(P, f)$

Proof: Appears in [19]. ■

The following is a statement that the algorithm is correct:

$$c = inf\, \overline{P} \Rightarrow fc(c, f) = inf\, \Psi(P, f)$$

This statement is proven by structural induction on f. First we show that it is correct for g, then we show that if it is correct for f_1 and f_2, then it is correct for $f_1 \triangle f_2$, and finally we show that if it is correct for g, f and S, then it is correct for $g\langle f\rangle S$ and $g\langle\langle f\rangle\rangle S$.

Lemma 3. $fc(c, g) = inf\, \Psi(P, g)$, *where* $c = inf\, \overline{P}$

Proof: Proof of a token based algorithm for the case when g is defined to be a CGP appears in [7]. ■

Lemma 4. $fc(c, f_1 \triangle f_2) = inf\, \Psi(P, f_1 \triangle f_2)$, *where* $c = inf\, \overline{P}$

Proof:

$fc(c, f_1 \triangle f_2)$
$= \{$ from the algorithm $\}$
$sup\{fc(c, f_1), fc(c, f_2)\}$
$= \{$ by induction, and since $c = inf\, \overline{P}$ $\}$
$sup\{inf\, \Psi(P, f_1), inf\, \Psi(P, f_2)\}$
$= \left\{ \begin{array}{l} \text{the } sup \text{ of the first cut to satisfy } f_1 \text{ and the first cut to} \\ \text{satisfy } f_2 \text{ is equal to the first cut which satisfies both } f_1 \\ \text{and } f_2. \end{array} \right\}$
$inf\{a \mid a \in \overline{P} \wedge [P..a] \models f_1 \wedge [P..a] \models f_2\}$
$= \{$ semantics $\}$
$inf\{a \mid a \in \overline{P} \wedge [P..a] \models f_1 \triangle f_2\}$
$= \{$ defn $\Psi(\ )$ $\}$
$inf\, \Psi(P, f_1 \triangle f_2)$

■

The next lemma is used in the proof of $fc(P, g\langle f\rangle S)$ several times. Presenting it here greatly simplifies the presentation of the proof for $fc(P, g\langle f\rangle S)$. Figure 5 shows the structure of the posets in this lemma.

Lemma 5. $Z(a_1, a_2, b_1, b_2, f, P)$, *which is defined as:*

$a_1, a_2, b_1, b_2 \in \overline{P} \wedge a_1 \preceq b_1 \wedge [b_1..b_2] \models f \wedge a_2 = inf\, \Psi([a_1..P], f)$
$\Rightarrow$
$a_2 \preceq b_2 \wedge [a_1..a_2] \models f$

Proof:

The following are assumed:

$a_1, a_2, b_1, b_2 \in \overline{P}$
$a_1 \preceq b_1$

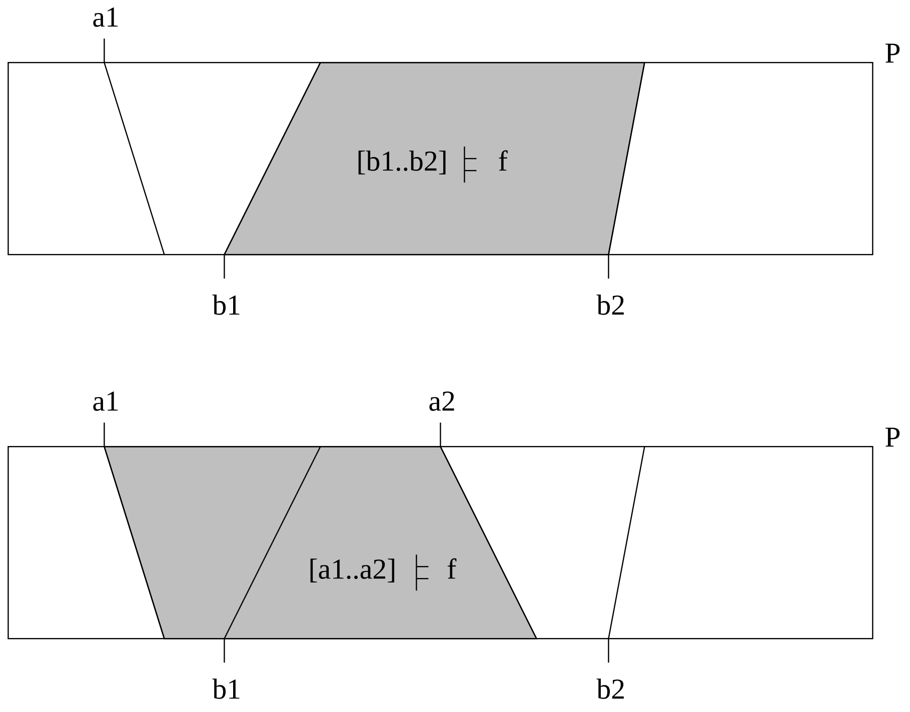

Fig. 5. Structure of posets in lemma 5.

$[b_1..b_2] \models f$
$a_2 = inf\,\Psi([a_1..P], f)$

Note that $[b_1..b_2] \models f$ implies that $b_1 \preceq b_2$, which in turn implies that $[b_1..b_2] \subseteq [a_1..b_2]$. Thus, by monotonicity, $[a_1..b_2] \models f$. This implies that $b_2 \in \Psi([a_1..P], f)$. And since $a_2 = inf\,\Psi([a_1..P], f)$, then $a_2 \preceq b_2$. ∎

Lemma 6. *$fc(c, g\langle f\rangle S) = inf\,\Psi(P, g\langle f\rangle S)$, where $c = inf\,\overline{P}$*

Proof: From the algorithm, it is clear that $fc(c, g\langle f\rangle S) = a_3$, where:
$a_1 = fc(c, g)$
$a_2 = fc(a_1, f)$
$a_3 = fc(a_2, S)$
By structural induction, we also know that:
$a_1 = inf\,\Psi([c..P], g)$
$a_2 = inf\,\Psi([a_1..P], f)$
$a_3 = inf\,\Psi([a_2..P], S)$

We must show that $a_3 = inf\,\Psi(P, g\langle f\rangle S)$. The proof is divided into two cases:
Case 1: $P \models g\langle f\rangle S$
Case 2: $P \not\models g\langle f\rangle S$
Case 1: $P \models g\langle f\rangle S$
In order to show $a_3 = inf\,\Psi(P, g\langle f\rangle S)$, it suffices to show:

Claim 1.1: $a_3 \in \Psi(P, g\langle f\rangle S)$, and
Claim 1.2: $x \in \Psi(P, g\langle f\rangle S) \Rightarrow a_3 \prec x$.

$P \models g\langle f\rangle S$
$\left\{ \begin{array}{l} \text{Let } x \text{ be any element in } \Psi(P, g\langle f\rangle S). \\ \text{It exists since } P \models g\langle f\rangle S. \end{array} \right\}$
$x \in \Psi(P, g\langle f\rangle S)$
{ By definition of cutsets: }
$x \in P \wedge [P..x] \models g\langle f\rangle s$
{ Semantics tell us b_1 and b_2 exist such that: }
$b_1, b_2 \in \overline{[P..x]} \wedge b_1 \prec b_2 \wedge b_1 \models g \wedge (b_1..b_2) \models f \wedge [b_2..x] \models S$
{ The preconditions of $Z(\bot, a_1, \bot, b_1, g, P)$ are satisfied: }
$a_1 = inf\,\Psi(P, g) \wedge [\bot..b_1] \models g \wedge \bot \preceq \bot$
{ The consequents are: }
$a_1 \preceq b_1 \wedge [\bot..a_1] \models g$
$\left\{ \begin{array}{l} \text{Now we do the same thing with } Z(a_1, a_2, b_1, b_2, f, [a_1..P]). \\ \text{The preconditions are:} \end{array} \right\}$
$a_2 = inf\,\Psi([a_1..P], f) \wedge (b_1..b_2) \models f \wedge a_1 \preceq b_1$
{ The consequents are: }
$a_2 \preceq b_2 \wedge (a_1..a_2) \models f$
$\left\{ \begin{array}{l} \text{One more time with } Z(a_2, a_3, b_2, x, S, [a_2..P]). \\ \text{The preconditions are:} \end{array} \right\}$
$a_2 \preceq b_2 \wedge a_3 = inf\,\Psi([a_2..P], S) \wedge [b_2..x] \models S$
{ The consequents, one of which proves claim 1.2, are: }
$a_3 \preceq x \wedge [a_2..a_3] \models S$
{ We have collected the following true statements: }
$a_1 \preceq a_2 \wedge a_1 \models g \wedge (a_1..a_2) \models f \wedge [a_2..a_3] \models S$
{ And by semantics: }
$a_3 \in \overline{P} \wedge [P..a_3] \models g\langle f\rangle S$
{ Which leads us to claim 1.1: }
$a_3 \in \Psi(P, g\langle f\rangle S)$

Case 2: $P \not\models g\langle f\rangle S$

$P \not\models g\langle f\rangle S$
{ semantics }
$\neg(\exists c_1, c_2 :: c_1, c_2 \in \overline{P} \wedge c_1 \prec c_2 \wedge c_1 \models g \wedge (c_1..c_2) \models f \wedge [c_2..P] \models S)$
{ Instantiate c_1, c_2 with a_1, a_2 and apply de Morgan's law: }
$a_1 \notin \overline{P} \vee a_2 \notin \overline{P} \vee a_1 \not\prec a_2 \vee a_1 \not\models g \vee (a_1..a_2) \not\models f \vee [a_2..P] \not\models S$
$\left\{ \begin{array}{l} \text{Since } (a_1 \notin \overline{P} \Rightarrow a_1 \not\models g) \text{ and } (a_1 \not\prec a_2 \Rightarrow (a_1..a_2) \not\models f) \text{ and} \\ (a_2 \notin \overline{P} \Rightarrow [a_2..P] \not\models S)\text{, then:} \end{array} \right\}$
$a_1 \not\models g \vee (a_1..a_2) \not\models f \vee [a_2..P] \not\models S$
{ By definition of a_1, a_2 and a_3: }
$a_1 = \top \vee a_2 = \top \vee a_3 = \top$
{ By definition of a_3: }
$a_3 = \top$

■

Lemma 7. *$fc(c, g\langle\langle f\rangle\rangle S) = inf\,\Psi(P, g\langle\langle f\rangle\rangle S)$, where $c = inf\,\overline{P}$*

Proof: Similar to the proof for weak sequences: $g\langle f\rangle S$ ■

8 Conclusions

An RCL formula is essentially a specification of a pattern of CGPs. If each CGP occurs in a computation, and they occur in the correct pattern, then the RCL formula is true in that computation.

RCL is a recursive logic, which means that the patterns can be recursive. logics are intuitive because there are fewer constructs and rules to remember. They are also powerful because the full power of the logic is available at each level of recursion. As a result of these properties of recursive logics, RCL is simple yet powerful. Computational structures of common paradigms such as butterfly synchronization and distributed consensus can be expressed easily in RCL.

A high level algorithm for detecting whether or not a computation satisfies a given RCL formula was presented. The complexity of the algorithm was not analyzed due to space constraints, but it is shown in [19] to be quite efficient – about the same as the complexity of detecting a single CGP.

9 Acknowledgments

We are grateful to Don Pazel at IBM for many fruitful discussions on the topic of breakpoint logic.

References

1. Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems: fundamental concepts and mechanisms, in Distributed Systems*, chapter 4. ACM Press, Frontier Series. (S. J. Mullender Ed.), 1993.
2. C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem. In *Proc. of the Workshop on Distributed Algorithms*, France, September 1995.
3. H. K. Chiou and W. Korfhage. Enf event predicate detection in distributed systems. In *Proc. of the Principles of Distributed Computing*, pages 91–100, Los Angeles, CA, 1994. ACM.
4. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.
5. C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.
6. E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc. of the 23rd Intl. Conf. on Parallel Processing*, St. Charles, IL, August 1994.
7. V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.
8. V. K. Garg, C. Chase, J. R. Mitchell, and R. Kilgore. Detecting conjunctive channel predicates in a distributed programming environment. In *Proc. of the 28th*

Hawaii International Conference on System Sciences, pages 232–241, Vol II, January 1995.
9. 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*, San Antonio, TX, October 1995.
10. V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.
11. V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.
12. V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, Submitted.
13. D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proc. of the* 21^{st} *International Conference on System Sciences*, volume 2, pages 166–175, January 1988.
14. G. Hoagland. A debugger for distributed programs. Master's thesis, University of Texas at Austin, Dept. of Electrical and Computer Engineering, Austin, TX, August 1991.
15. M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
17. B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the* 8^{th} *International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, July 1988. IEEE.
18. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
19. A. I. Tomlinson. *Observation and Verification of Software for Distributed Systems.* PhD thesis, University of Texas at Austin, Dept. of Electrical and Computer Engineering, Austin, TX, August 1995.
20. A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. (Reprinted in SIGPLAN Notices, Dec. 1993).

This article was processed using the LaTeX macro package with LLNCS style