OBSERVATION AND VERIFICATION OF SOFTWARE FOR DISTRIBUTED SYSTEMS

APPROVED BY

DISSERTATION COMMITTEE:

Supervisor: ______________________

____

____

____

____

OBSERVATION AND VERIFICATION OF SOFTWARE FOR DISTRIBUTED SYSTEMS

by

ALEXANDER IVOR TOMLINSON, B.S, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1995

Acknowledgments

I would like thank Professor Vijay K. Garg for supervising this research and providing guidance along the way. I also thank my supervising committee (Professors J.C. Browne, C.M. Chase, M.G. Gouda and C.L. Wu) for allocating their valuable time to this endeavor.

I am also grateful for financial support from the following: MCD University Fellowship, University Continuing Fellowship, NSF Grant CCR 9110605, TRW Faculty Assistantship Award, GM Faculty Fellowship, and a grant from IBM.

My parents cannot be thanked enough for all that they have done for me. And lastly, I am most grateful to my future wife, Ivonne Mercado, for being patient with me and for providing encouragement.

Alexander Ivor Tomlinson

The University of Texas at Austin
August, 1995

OBSERVATION AND VERIFICATION OF SOFTWARE FOR DISTRIBUTED SYSTEMS

Publication No. _______________

Alexander Ivor Tomlinson, Ph.D.
The University of Texas at Austin, 1995

Supervisor: Vijay Kumar Garg

The main contribution of this dissertation is the development of new theory and techniques in the areas of program observation and verification. These areas are important parts of the specification, design, development and testing phases of a software project's lifecycle. Program observation involves formulating a query about the behavior of a program and then monitoring the program as it executes in order to determine the result of the query. Observation is used in the development phase to track down bugs and clarify understanding of a program's behavior, and in the testing phase to ensure that a program behaves as expected for a given input set. Program verification involves proving that a design meets its specification. This is done during the design phase to ensure that the specifications are satisfied. The fundamental difference between observation and verification is that observation applies to individual executions of a program and verification applies to all executions.

Three observation systems are developed in this dissertation: global functions, recursive logic (RCL), and DAG expressions. A global function has the form of a sum of local variables. The observed program is monitored to determine if the value of the global function exceeds some constant. Two special cases of this problem are considered. 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. DAG expressions are an extension of regular expressions from strings to directed acyclic graphs. While a regular expression specifies patterns of letters in a string, a DAG expression specifies patterns of letters in a DAG. The letters in a DAG expression represent local predicates. Nonintrusive observation algorithms for each of these three systems are developed, proven and analyzed for complexity.

A system for specifying and verifying distributed programs is developed. A new proof technique is also introduced which uses induction on the complement of the causally precedes relation. These specification and verification techniques are demonstrated on two well known algorithms (vector clocks and mutual exclusion) and one lesser known algorithm (direct dependency clocks).

Table of Contents

List of Figures

Chapter 1

Introduction

There are many indications that severe problems exist in the software development process. The cost of software development can be as much as a staggering $900 per line of code. The average schedule slip in software projects is 60%, and more than 50% of software projects run over budget. Up to 50% of large projects and 10% of small projects are canceled[24]. Software engineering was rated as the number one strategic concern among the corporate members on the “board of directors” of the Electrical and Computer Engineering Department at The University of Texas at Austin. All these numbers are symptoms of what is commonly referred to as the “software crisis”.

Software is a $93 billion dollar industry in the US alone, thus there is enormous incentive to improve all aspects of the development process. Unfortunately, there are several trends which continuously complicate the problem, making the need for improvements even more crucial. These trends include the emergence of distributed systems, the growth of program size, and the use of software in safety critical applications. Distributed systems are prevalent in society and are inherently much more complex than sequential programs. The incessant improvements in hardware technology enable larger and larger software systems, which also adds complexity. In addition, software is used in many economically and socially critical applications where failure cannot be tolerated. These critical applications are found in the financial industry, transportation systems, medical equipment, military equipment and nuclear power facilities. There is a clear need for improvement in the software development process, and that need is becoming more and more critical due to these trends.

The main contribution of this dissertation is the development of new theory and techniques in the areas of program observation and verification, which are important parts of a software project's lifecycle.

Program observation is most useful in the testing phase of a project, but its theory has application in the specification phase as well. It involves formulating a query about the behavior of a program and then executing the program, augmented with an observation algorithm, in order to determine the result of the query. For example, a query which could detect errors in a reservation system is "does there exist a global state in which one reservation is allocated to more than one person" (this is an example of mutual exclusion). This query could be used during testing to ensure that such an error is not possible. It could also be used during specification to insist that such an error is not permissible.

Program verification is most useful in the design phase, but its theory also has application in testing. It involves proving that a design meets some specification. This is done during the design phase to ensure that the specifications are satisfied. An example assertion which one might want to verify is: "if a customer initiates a transaction, then eventually the transaction will be completed" (this is an example of progress). This assertion could also be used in testing to make sure the run time behavior obeys the assertion.

The fundamental difference between observation and verification is that observation applies to individual executions of a program and verification applies to all possible executions. This is an important distinction and should be kept in mind while reading this dissertation. Both observation and verification can be thought of as functions. In observation the argument to the function is a single execution and the result is valid for that execution and no others. In verification the argument to the function is the program text and the result applies to all executions of that program.

This dissertation presents three different observation systems and one verification system, all four of which are tailored for distributed systems. In our model, a distributed system is a set of processes which communicate via message channels only (i.e., no shared memory). Observation algorithms for each of observation systems are described, proven and analyzed for complexity. The algorithms do not alter the behavior of the observed program. There is the unavoidable overhead created by attaching the observer algorithm to the observed program, but the observed program is never blocked or interfered with in any way other than this overhead.

The first of the three observation systems, global functions, is described in chapter 4. Global functions have the form of a sum of local variables. The observed program is monitored to determine if the value of the global function exceeds some constant. Two special cases of this problem are considered, one with two local variables that can take any value, and the other with any number of variables that can take on only two values. In neither case is the number of processes limited – only the number of relevant variables or the values of the variables are limited.

Chapter 5 describes RCL, the second system for program observation. 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.

DAG expressions, the third observation system, are presented in chapter 6. DAG expressions are an extension of regular expressions from linear orders to directed acyclic graphs. While a regular expression specifies patterns of symbols in a linear order, a DAG expression specifies patterns of symbols in a directed acyclic graph. The symbols in DAG expressions represent local predicates.

Finally, in chapter 7, a system for specifying and verifying distributed programs is described. In this system, a *program* is a specification of the relationship between local states which are adjacent to each other in the poset. A *property* is a statement regarding the structure of a poset. A program satisfies a property if each poset produced by the program satisfies the property. The goal is to prove that a given program satisfies a given property. The system also introduces a new proof technique, induction on the complement of the causally precedes relation, which is useful in proving poset properties. The verification techniques are demonstrated on two well known algorithms (vector clocks and mutual exclusion) and one lesser known algorithm (direct dependency clocks).

The next two chapters review research related to observation and verification, and describe the model and notation used throughout this dissertation.

Chapter 2

Related Work

2.1 Observation

Observation systems can be categorized with respect to three independent attributes: stable vs. unstable, functions vs. predicates, and global-state-based vs. poset-based. Figure 2.1 shows this categorization.

The function vs. predicate dimension simply refers to whether or not the observation gives a boolean result. Since predicates are essentially boolean functions, they are a special case of the "functions" category.

The global-state-based vs. poset-based dimension refers to whether or not the observation is based on global states. Invariants and safety properties, which state that some property is true in all global states, are in the global-state-based category. A familiar example is mutual exclusion.

A poset-based observation is one which depends on additional struc-

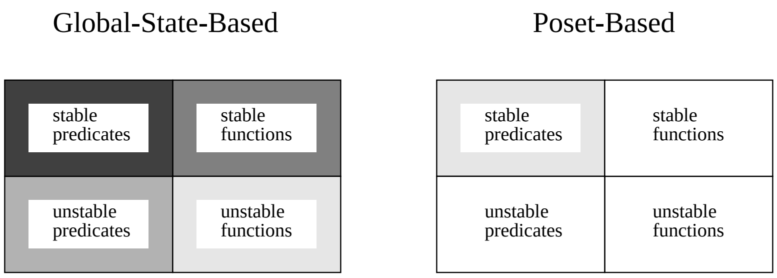

Figure 2.1: Research Areas in Program Observation

ture of the poset. An example is the sequence of local predicates which is expressed as 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[34]) such that local predicate p_i is true in local state σ_i. This type of predicate cannot be evaluated on individual global states.

A stable predicate stays true once it becomes true while an unstable predicate may oscillate between true and false. A stable function (or monotonic function) is one which increases with time. An example of a stable predicate is: "each process has received the token at least once". Clearly, in any execution, once this predicate becomes true, it stays true. An example of an unstable predicate which would be useful in analyzing an implementation of the two phase commit protocol is: "all processes are in the *ready* state".

In figure 2.1, each of the eight areas are shaded according to the amount of past research in that area. Darker shading indicates more research, no shading indicates that it is essentially an unexplored area. Stable global predicates have received widespread attention. The work of this dissertation falls into unstable global functions, and stable poset predicates. The remainder of this section is devoted to describing previous research in each of the areas shown in this figure.

2.1.1 Stable Global Predicates

This type of global function is commonly known as stable properties and has been studied extensively in the literature. Since this dissertation does not address stable global functions, the review of related work will be brief.

Chandy and Lamport [7] invented the global snapshot, which is the basis for almost all subsequent research on detecting stable predicates. A global snapshot is a freeze-frame picture of a distributed computation. The picture corresponds to a global state of the computation with one caveat: the global state is not guaranteed to have occurred. However, it is guaranteed to be consistent with at least one global state that did occur.

The snapshot approach to detecting stable predicates is conceptually simple. Snapshots of the system are taken repeatedly until one of them determines that the predicate is true or until the computation ends. This approach works because the predicate stays true once it becomes true.

Many researchers have developed more efficient algorithms [27, 50, 22, 40, 45, 1, 51], and others have developed algorithms with different underlying assumptions such as non-FIFO message channels [37] and synchronous message channels [6].

Chodrow and Gouda [10] describe a facility for observing a concurrent program in order to detect safety and progress properties. Their system is based on global snapshots, and as such, is most useful for stable properties. They also describe an annotation language which uses quantification to specify properties on nonscalar data structures.

The snapshot approach used for stable predicates does not work for unstable predicates because the predicate may become true and then false again between two snapshots. An entirely different approach is required for unstable predicates.

2.1.2 Unstable Global Predicates

There has been much work in observing global states of distributed computations. Babaoğlu and Marzullo [3] and Schwartz and Mattern [49] both survey recent work on detecting consistent global states in a distributed system. Recently, Chase and Garg [8] have shown that unstable 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. Weak CGP (discussed later) is an example of a linear predicate.

Cooper and Marzullo [11] present algorithms for online detection of three types of unstable 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 the execution of one or more processes.

Their algorithms construct the lattice associated with a distributed execution [33] and then analyze it to detect the specified behavior. A node of this lattice represents a possible global state of the distributed computation and an edge represents an event that changes the global state of the computation. This approach can detect a very broad class of behaviors. The drawback is that the cost is high since it requires building a lattice, which has a time complexity that is exponential in the number of local states in the computation. Marzullo and Neiger [39] improve the performance of this approach by limiting the number of global states.

Manabe and Imase [38] present an alternative method of detecting "*possibly* g" which uses two identical runs of the observed program. The relative order of events is recorded in the first run and during the second run, each process is controlled in order to produce a global state which satisfies g, if it occurred.

Observation of general unstable global predicates is very expensive. As a result, researchers have devised classes of unstable global predicates which can be efficiently observed. One such class is *conjunctive global predicates* (CGP) as described by Garg and Waldecker [18]. A CGP consists of a conjunction of local predicates such as $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, where p_i is evaluated in process i. The CGP is evaluated on a global state and is defined to be true if and only if, for all i, p_i is true in process i's component of the global state.

Garg and Waldecker present strong and weak [19, 18, 20] forms of CGP which correspond to *possibly* and *definitely* of Cooper and Marzullo [11]. 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.

The most recent improvement of weak CGP by Garg and Chase [21] results in a fully decentralized token based algorithm which does not require the use vector clocks. Garg, Chase, Mitchell and Kilgore [16], extend weak conjunctive predicates to include predicates on the state of message channels.

Venkatsen and Dathan [57] present fully distributed offline algorithms for detecting CGP. Their algorithms assume that reproducible execution is possible and that message channels are FIFO.

2.1.3 Stable Global Functions

Stable global functions are essentially monotonic functions on the global states of a distributed program. Therefore, much of the work on stable global *predicates* apply to stable global *functions* as well.

2.1.4 Unstable Global Functions

Stability and monotonicity are equivalent notions, and since monotonicity is more commonly used when referring to functions, it may help to think of non-monotonic functions instead of unstable functions. Very little research from this area has appeared in the literature.

Tomlinson and Garg [55, 54] defined *relational global predicates* which are based on non-monotonic functions. That work is the basis for chapter 4 of this dissertation, which presents work on monitoring unstable functions on global states. In another work [53], they described mechanisms for maintaining the value of a global function within specified bounds.

Chase and Garg [8] present a method for transforming a global function based on summation (a special case of the global functions considered in this dissertation) into the well known graph theoretic maximum-flow problem.

2.1.5 Poset Predicates

Global predicates consist of a set of local states which have a special causal relationship – they are concurrent with each other. Poset-based observations also consist of a set of local states with specific causal relationships, however the relationships are not restricted to concurrency. Poset predicates can be as complex as path expressions or as simple as a sequence of local predicates (both are described below).

One of the early works in this area was Miller and Choi's *sequence of local predicates* [42]. These poset predicates are expressed as an ordered list of local predicates $p_1, \dots p_k$. This predicate is true in an execution if and only if there exists a sequence of local states $\sigma_1, \dots \sigma_k$ (sequenced by Lamport's happens-before relation) such that local predicate p_i is true in local state σ_k.

Hurfin, Plouzeau and Raynal [30] 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 a sequence of local 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, Tomlinson and Garg [14] 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 $(s_1, s_2, \dots, s_n)$ such that p is true in s_1, q is true in $s_2, \dots, s_{n-1}$, 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 [23], the same authors extend regular patterns to allow patterns on directed acyclic graphs instead of just strings. This work forms the basis for DAG patterns presented in chapter 6 of this dissertation.

Each type of poset predicate mentioned so far consists of sets of local states which are related to each other by Lamport's happens-before relation. Conceptually, they are quite simple, and they are efficient to detect because all the information needed for detection can be piggybacked on existing messages.

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 [9] 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.

The logic presented in chapter 5 of this dissertation, *recursive logic* (RCL), is also based on conjunctive global predicates. RCL combines CGP in a recursive fashion so that at each level of recursion, the full power of RCL is available. This is similar to boolean expressions where each subexpression is a boolean expression. This is in contrast to many poset predicate logics which have no such recursive powers.

Haban and Weigel [25] 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 [28], the work was noteworthy in that it was an early attempt to develop a recursive poset predicate logic.

There are other forms of poset predicates which start to cross the boundary into program specification. For example, temporal logic is a well known mechanism for program specification, and it can also be used to define poset predicates.

Harter, Heimbigner, King [26] developed the Interactive Distributed Debugger (IDD) which uses *interval logic* to specify program behavior. IDD observes the program being debugged and halts it when the interval logic specification is violated. Interval logic uses the familiar temporal operators for *always* and *eventually*, but allows them to have limited scope in the computation. The syntax of a specification is $I[R]$ which asserts the temporal formula R must hold the interval defined by I. IDD and the associated interval logic have two major drawbacks. One is that the only detectable events are broadcasts on a shared media. This indicates that it might not be possible to efficiently implement their work in systems which don't support a single broadcast medium. The second drawback is that one can write specifications which their algorithms are not capable of detecting.

Baiardi, de Francesco and Vaglini [4] describe the debugger used with the distributed programming language ECSP. Their system requires one to completely specify the communication and synchronization behavior of each process in the system. The specifications can refer to communication activity as well as process state. Each specification can only refer to events from a single process.

Hyder in his dissertation [32] describes a distributed debugger which considers the entire debugging cycle. The cycle begins with the program text and progresses to the user's expectation of the program behavior, followed by the user's observation of the program. If the observation does not match the expected behavior, then the cycle is repeated.

Bates and Wileden [5] describe the *event definition language* (EDL) which provides a mechanism for defining multiple levels of abstract events from underlying primitive events. The major drawback is that an accurate global clock is assumed to be available, and that ambiguities may arise when low level events are used in multiple high level events.

2.2 Verification

The work presented in this dissertation is based on previous work by Garg and Tomlinson [17, 52] and contributes three main ideas which are useful for reasoning about distributed programs. The first idea is that of induction on the $\not\rightarrow$ relation. Proving properties of distributed programs is very difficult for all but the simplest of programs, and induction on $\not\rightarrow$ provides a new option for such proofs. Any new tools for proving distributed programs is a welcome addition due to the complexity of the task.

The second idea is the "window" model of messages in a distributed program. Usually, messages are modeled as tuples of data values that are set by one local state and read by another. A different approach is proposed here: a message is modeled as a window into the local state in which the message originated. In the window model, a message is a mechanism for granting the receiver read-only access to the local state from which the message was sent. When a message is sent from state s and received in state t, state t can read components of state s directly. For example, the statement $t.x := s.y$ sets variable x in state t to the value that y had in state s (clearly this statement is only allowed if a message from s to t exists). The advantage of the window model is that the causal relationship between sender and receiver is explicit, which usually results in simpler proofs. Another advantage is that there is less mathematical machinery to work around when designing proofs. The only disadvantage is that it is one step removed from implementation, which is minor in comparison to the benefits gained.

The third contribution is that our system for reasoning about distributed programs is static. Most program logics are dynamic because the truth value of a formula is determined by the current location of the program. The vast majority of distributed program logics (see chapters 14, 15, 16 in [56]) are descendants of either Hoare Logic [29] or Temporal Logic [47] and as such, define the current location to be a global state. Thus, a formula is interpreted in the context of a current global state.

In our system, local states are given names so that their variables can be referenced. This allows us to abandon the notion of a current location and to define a static program logic. For example, the property that a variable x is monotonic would be stated as follows in our system:

$$s \rightarrow t \Rightarrow s.x \leq t.x$$

where s and t are *local* states, $s.x$ is the value of variable x in local state s, and $\rightarrow$ is Lamport's happens-before relation. The above formula is static. It need not be interpreted with respect to the current location of the program. In fact, in our system, the current location of a program is not even defined.

Our system is based strictly on the partial order of local states; global states are not given any special significance. This contrasts virtually all other program logics which are dynamic and define the current location to be the current global state.

Chapter 3

Model and Notation

3.1 Quantified expressions

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.

3.2 Posets

Any distributed computation can be modeled as a decomposed partially ordered set (deposet) of process states [13]. A deposet is a partially ordered set $(P, \leadsto)$ 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[34]. 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 $s \prec_{im} t$.

3.3 Global States, Cuts, Antichains, Chains

A antichain 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 antichains in $(P, \rightarrow)$. We also use the terms "cut" and "global state" 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.

3.4 Local Variables

Given a local state $\sigma \in P$, we denote the value of a variable, say x, in state σ by $\sigma.x$. We reserve the special variable $\sigma.p$ to denote the process in which σ occurs. That is, $\sigma.p = i$ if and only if $\sigma \in P_i$. Similarly, $\sigma.v$ is used to denote the value of a vector clock in σ.

3.5 Pred and Succ functions

If $(u \to v)$ then $max(u, v) = v$ and $min(u, v) = u$. Since max and min are commutative and associative, the maximum and minimum element of any chain in $(P, \to)$ are well-defined. The unit elements of the max and min operators are $\bot$ and $\top$ respectively. Thus max applied to a zero length chain returns $\bot$. We require that $(\forall u : u \in P : \bot \to u \wedge u \to \top)$, and also that $\bot \to \bot$ and $\top \to \top$.

The predecessor and successor functions are defined as follows for $u \in P$ and $1 \leq i \leq N$:

$$pred.u.i = (\max v : v \in P_i \wedge v \to u : v)$$

$$succ.u.i = (\min v : v \in P_i \wedge u \to v : v)$$

Thus if $(pred.u.i = v)$ then v is the maximum element in $(P_i, \to)$ which happens before u, or $\bot$ if no element in P_i happens before u.

3.6 External Events, Intervals

An external event is the sending or receiving of a message. The n^{th} interval in P_i (denoted by (i, n)) is the subchain of $(P_i, \to)$ between the $(n-1)^{th}$ and n^{th} external events. For a given interval (i, n), if n is out of range then (i, n) refers to $\bot$ or $\top$. The notion of intervals is useful because the relation of two states belonging to the same interval is a congruence with respect to $\to$.

Thus, for any two states s and s' in the same interval and any state u which is not in that interval: $(s \rightarrow u \iff s' \rightarrow u)$ and $(u \rightarrow s \iff u \rightarrow s')$.

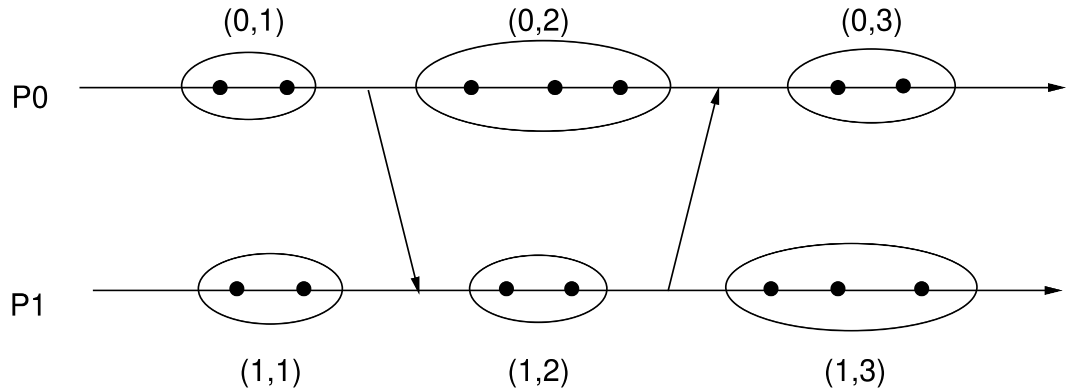

Figure 3.1: Intervals in a Poset

We take advantage of intervals in some algorithms by assigning a single timestamp to all states belonging to the same interval. Due to this congruence, the *pred* and *succ* functions and the $\|$ relation are well-defined on intervals. Figure 3.1 shows an example of poset intervals. It is clear from this figure that causality (and therefore concurrency) is preserved from states to intervals.

3.7 Vector Clocks

In a system with N processes, a vector clock [41, 13, 48] is a function which associates a vector of N non-negative integers with each state of the poset. Vector clocks are useful because they implement the *pred* function, and they encode Lamport's happens before relation. Given a state σ, we denote its vector clock value by $\sigma.v$. The important property of vector clocks is

$$\sigma \rightarrow \sigma' \iff \sigma.v < \sigma'.v$$

where the $<$ relation on vectors u and v is defined as:

$$u < v \iff (\forall i :: u[i] \leq v[i]) \wedge (\exists i :: u[i] < v[i])$$

The algorithm for implementing vector clocks is quite simple. We describe it here for process P_i; the algorithm is the same for each process. Initially $v[i]$ equals 1 and all other components equal zero. Upon sending a message, the message is tagged with the current value of v and then $v[i]$ is incremented by one. Upon receiving a message tagged with u (all received messages will have a message tag) execute $v[j] := max(v[j], u[j])$ for all j and then increment $v[i]$. This is the entire algorithm.

Chapter 4

Monitoring Functions on Global States

A global state function takes any global state and returns a value in some ordered domain D. This chapter presents algorithms for monitoring the value of a global state function during the execution of a distributed program.

We begin this chapter by giving informal examples of some global functions that these algorithms are capable of monitoring. Then, the global functions are defined precisely, and we revisit the same examples using the formal notation. And finally, algorithms for two cases of the global state function are described.

4.1 Examples

Using the notation that x_i is a variable in process i, the following examples should give you an idea of the kinds of things we will be able to determine about an execution.

1. Find the maximum value of $x_1 + x_2$ over all global states, where x_i ranges over the set of real numbers.

2. Determine if there exists a global state in which more than, say, one third of the processes are idle.

3. Determine if $x_1 \vee x_2$ is true in *all* global states, where x_i is boolean.

4. Determine if $x_1 \wedge x_2$ is true in *some* global state, where x_i is boolean.

4.2 Definitions

The *global state function*, *gf*, takes a global state and returns a value in domain D, which is totally ordered by the $<$ relation. Given a global state $C \in \overline{P}$, the value of $gf(C)$ depends upon the local states which make up C, and upon the local state function. The *local state function* takes any local state and returns a value in $D_l \subseteq D$. If σ is a local state, then $\sigma.x$ denotes the value of the local state function applied to σ. You can think of a local state function as a variable, x, that exists in each process and always has some value in D_l.

The value of $gf(C)$ is the "sum" of the values of x in the local states which make up the global state. We use *oplus* ($\oplus$) to denote this summation:

$$gf(C) = (\oplus\sigma : \sigma \in C : \sigma.x) \tag{GF}$$

The only restrictions on $\oplus$, D, and D_l are that they meet the following requirements:

set D :	D is totally ordered by $<$
set D_l :	$D_l \subseteq D$
oplus operator:	$\oplus : D \times D \mapsto D$
oplus commutative:	$a \oplus b = b \oplus a$
oplus associative:	$a \oplus (b \oplus c) = (a \oplus b) \oplus c$
oplus monotonic:	$a < b \Rightarrow (a \oplus c) < (b \oplus c)$

We give algorithms for monitoring special cases of GF. The cases are referred to as "Two Variables" and "Bivalent Variables".

Two Variables: GF is restricted so that its value depends only upon the local states of two processes. The program is monitored in order to determine the maximum value of GF.

Bivalent Variables: D_l is restricted to be a set with exactly two elements (e.g., a boolean domain). The program is monitored to determine if the value of GF ever exceeds some constant in D.

4.3 Examples Revisited

We now revisit the examples listed earlier and express each one formally. In each example, σ_1 and σ_2 range over P_1 and P_2 such that $\sigma_1 \parallel \sigma_2$.

1. Find the maximum value of $x_1 + x_2$ over all global states, where x_i ranges over the set of real numbers.

   Algorithm: Two Variables.

   Solution: Let $(D_l, D, \oplus) = (Reals, Reals, +)$.

   Monitor $gf(\{\sigma_1, \sigma_2, ...\}) = \sigma_1.x + \sigma_2.x$.

2. Determine if there exists a global state in which more than, say, one third of the processes are idle.

   Algorithm: Bivalent Variables.

   Solution: Let $(D_l, D, \oplus) = (\{0, 1\}, Integers, +)$. Let $x = 1$ when idle.

   Monitor $gf(C) = (+\sigma : \sigma \in C : \sigma.x)$ to see if it ever exceeds $N/3$.

3. Determine if $x_1 \vee x_2$ is true in *all* global states, where x_i is boolean.

Algorithm: Two Variables.

Solution: Let $(D_l, D, \oplus) = (\{false, true\}, \{false, true\}, \vee)$. Let $true < false$.

Monitor $gf(\{\sigma_1, \sigma_2, ...\}) = \sigma_1.x \vee \sigma_2.x$ to find max value.

4. Determine if $x_1 \wedge x_2$ is true in *some* global state, where x_i is boolean.

   Algorithm: Two Variables.

   Solution: Let $(D_l, D, \oplus) = (\{false, true\}, \{false, true\}, \wedge)$. Let $false < true$.

   Monitor $gf(\{\sigma_1, \sigma_2, ...\}) = \sigma_1.x \wedge \sigma_2.x$ to find max value.

4.4 Two Variables

In this case, we need only to consider the local state function on two processes. We assume that these two processes are P_1 and P_2 and we use σ_1 and σ_2 to denote local states in P_1 and P_2. Note that the computation may contain more than two processes, but the only two of them have local variables which affect the value of the global state function. When limited to these two processes, the global state function can be simplified:

$$gf(\{\sigma_1, \sigma_2, ...\}) = \sigma_1.x \oplus \sigma_2.x$$

In addition, it is clear that two states σ_1 and σ_2 appear in some global state $C \in \overline{P}$ if and only if $\sigma_1 \parallel \sigma_2$. This means that the original problem statement, GF, and can be simplified to GF1 as follows:

$$GF1 \triangleq (max\ \sigma_1, \sigma_2 : \sigma_1 \in P_1 \wedge \sigma_2 \in P_2 \wedge \sigma_1 \parallel \sigma_2 : \sigma_1.x \oplus \sigma_2.x)$$

We present two algorithms for monitoring GF1. The decentralized algorithm runs concurrently with the underlying program and can be used for monitoring the program in real time. The centralized version is decoupled from the underlying program and can run concurrently with the underlying program or after the underlying program terminates. We formally prove that both algorithms are sound and complete (ie, the algorithm results are consistent with what really happened). Before describing the centralized and decentralized algorithms, we describe the ideas behind them, and present some results which are necessary for proving the algorithms.

4.4.1 The main idea behind the algorithms

Consider a situation in which P_1 and P_2 do not send or receive any messages. Then every state in P_1 is concurrent with every state in P_2. In this case we could evaluate $maxx(P_1) \oplus maxx(P_2)$, where $maxx(P_i)$ is the maximum value of x_i in the set of states P_i. This expression would equal GF1. We use this basic idea in our algorithm, but there are complications resulting from message communication. Care must be take to ensure that $x_1 \oplus x_2$ is evaluated in a global state if and only if the global state is valid.

Each time a message m is received at P_1, or P_2, we compute a value K_m such that the maximum value of K_m for all such messages is equal to GF1 (Let $Kmax$ refer to this maximum value). For example, suppose a message m is received at P_1. To compute K_m we first compute sequences of states, $P_1^m \subseteq P_1$ and $P_2^m \subseteq P_2$, such that every state in P_1^m is concurrent with every state in P_2^m. Refer to figure 4.5 for a picture of this. Then K_m is defined as follows:

$$K_m \triangleq maxx(P_1^m) \oplus maxx(P_2^m)$$

If, for every pair of concurrent states $\sigma_1 \parallel \sigma_2$, there exists a message m such that $\sigma_1 \in P_1^m$ and $\sigma_2 \in P_2^m$, then we could conclude that $Kmax >= \sigma_1.x \oplus \sigma_2.x$.

Furthermore, if, for every message m, all states in P_1^m are concurrent with all states in P_2^m, then we could conclude that there exists a pair of concurrent states $\sigma_1 \parallel \sigma_2$, such that $Kmax = \sigma_1.x \oplus \sigma_2.x$.

Together, the previous two paragraphs demonstrate that $Kmax$ is indeed equal to $GF1$. At this point we have a basis from which to develop and prove an algorithm. The tasks that need to be accomplished are listed below along with the sections or theorems which address that task.

1. Show how to determine P_1^m and P_2^m for each message m which is received at P_1 or P_2 (section 4.4.2).
2. Prove that $maxx(P_1^m) \oplus maxx(P_2^m)$ is equal to the maximum value of $\sigma_1.x \oplus \sigma_2.x$ for all $\sigma_1 \in P_1^m$ and $\sigma_2 \in P_2^m$ (lemma 4.3).
3. Prove that $\sigma_1 \parallel \sigma_2$ if and only if there exists a message m (rcvd at P_1 or P_2) such that $\sigma_1 \in P_1^m$ and $\sigma_2 \in P_2^m$ (lemma 4.4).
4. Prove that $Kmax = GF1$ (theorem 4.1).

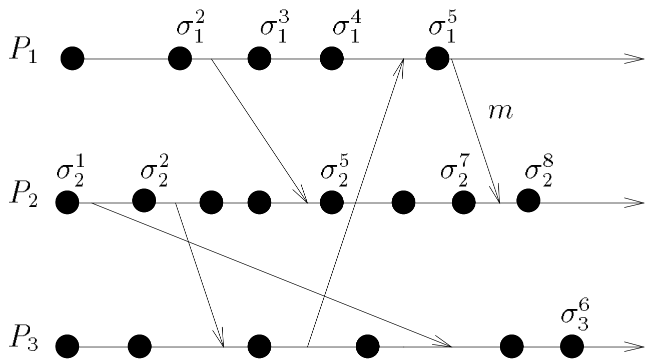

Figure 4.1: Demonstration of the *pred* function: $pred.\sigma_1^5.2 = \sigma_2^2$, $pred.\sigma_3^6.2 = \sigma_2^2$, $pred.\sigma_1^4.2 = \perp$.

4.4.2 Determining P_1^m and P_2^m

P_1^m is a sequence of states at P_1, $(\sigma_1^{lo_1+1}, \ldots, \sigma_1^{hi_1})$, and P_2^m is a sequence at P_2, $(\sigma_2^{lo_2+1}, \ldots, \sigma_2^{hi_2})$. Given a message m which has been received at P_1 (or P_2), we need to be able to determine values for lo_1, hi_1, lo_2, hi_2. This is accomplished by keeping track of predecessors of states in P_1 and P_2. Consider a state $\sigma_i \in P_i$. The predecessor of σ_i in P_j, denoted $pred.\sigma_i.j$, is the latest state in P_j which happens before σ_i. Figure 4.1 shows an example. Notice that the predecessor or σ_i^n on P_i is just the previous state σ_i^{n-1}.

Suppose a message m is received in some state $\sigma \in P_1 \cup P_2$. Then we define lo_1, hi_1, lo_2, hi_2 as follows:

$$\begin{array}{rclcrcl} \sigma_1^{hi_1} & = & pred.\sigma.1 & \qquad & \sigma_2^{lo_2} & = & pred.\sigma_1^{hi_1}.2 \\ \sigma_2^{hi_2} & = & pred.\sigma.2 & \qquad & \sigma_1^{lo_1} & = & pred.\sigma_2^{hi_2}.1 \end{array}$$

Applying these definitions to the message labeled m in figure 4.1, we see that

$P_1^m = (\sigma_1^3, \sigma_1^4, \sigma_1^5)$, and $P_2^m = (\sigma_2^3, \sigma_2^4, \sigma_2^5, \sigma_2^6, \sigma_2^7)$, Thus, if we can evaluate the above *pred* functions, then we can determine P_1^m and P_2^m. We use a 2×2 matrix clock to enable us to evaluate these expressions.

4.4.3 Finding lo_1, hi_1, lo_2, hi_2 with a 2×2 matrix clock

Instead of associating values for lo_1, hi_1, lo_2, hi_2 with each state, we associate these values with intervals. Thus P_1^m and P_2^m now represent sequences of intervals, which themselves are sequences of states.

To evaluate the *pred* functions, we use a 2×2 matrix clock as described by Raynal [48]. In this section we present an algorithm for maintaining the matrix clock and prove that from it we can determine lo_1, hi_1, lo_2, hi_2. Once we have these values, we can find P_1^m and P_2^m, and then determine if GF1 is satisfied.

The matrix clock algorithm is presented in figure 4.3. The algorithm is easier to understand by noticing the vector clock algorithm embedded within it. If the row index is held constant, then it reduces to the vector clock algorithm. Figure 4.2 shows values of the matrix clock and message tags on an example run. Note that row 1 of P_1's matrix is a traditional vector clock restricted to indices 1 and 2, and row 2 equals the value of P_2's vector clock at a state in the "past" of P_1. Similar properties hold for P_2's matrix clock.

Let M_k^n denote the value of the matrix clock in interval (k, n). The following description applies to an $N \times N$ matrix clock in a system with N processes. The 2×2 matrix is the upper left submatrix of the $N \times N$ matrix. We describe the information contained in a matrix clock M_k^n at interval (k, n)

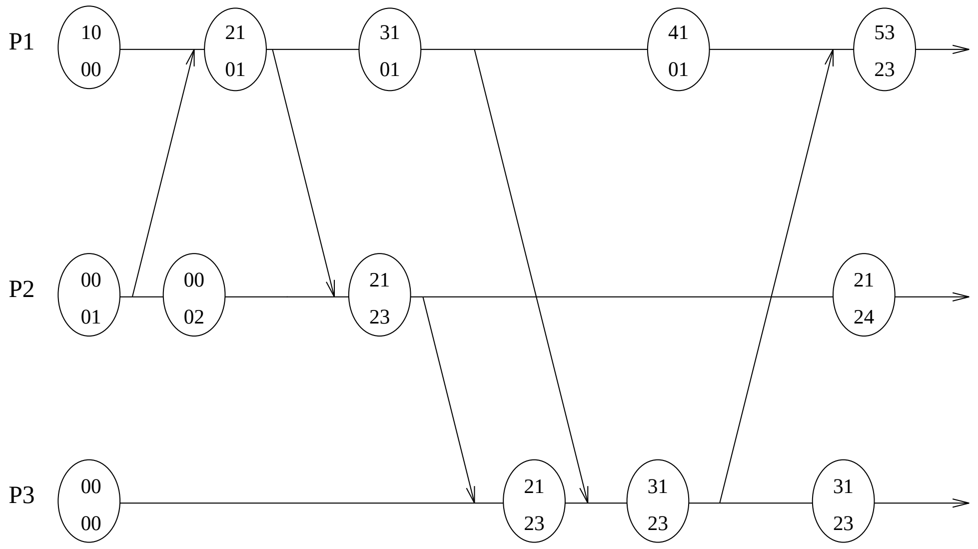

Figure 4.2: Matrix clock example.

in three steps. First we explain the meaning of the k^{th} diagonal element. Then we explain the meaning of the k^{th} row, followed by the entire matrix.

The k^{th} diagonal element, $M_k^n[k,k]$, is an interval counter at P_k. Thus for any interval (k,n), the following is true: $M_k^n[k,k] = n$. Row k of M_k^n is equivalent to a traditional vector clock. In fact, when we say "P_k's vector clock" we are referring to row k of P_k's matrix clock. Since the vector clock implements the *pred* function, row k of M_k^n can be used to find predecessors of $(k,n) = (k, M_k^n[k,k])$ as follows: the predecessor of $(k, M_k^n[k,k])$ on process j is the interval $(j, M_k^n[k,j])$. In fact, this applies to all rows: the predecessor of $(i, M_k^n[i,i])$ on process j is the interval $(j, M_k^n[i,j])$. Furthermore, row k of M_k^n equals the diagonal of M_k^n. Thus we can use row k to find the $pred.(k,n).j$ for $j \neq k$, and then use row j to find $pred.(pred.(k,n).j).i$ for $i \neq j$.

The matrix clock can be summarized in three simple rules. First, the

```
To initialize:
        M_k[·,·] := 0;
        if (k = 1 ∨ k = 2) then
            M_k[k,k] := M_k[k,k] + 1;
        end_if

To send a message:
        Tag message with M_k[·,·];
        if (k = 1 ∨ k = 2) then
            M_k[k,k] := M_k[k,k] + 1;        increment local clock
        end_if

Upon receipt of a message tagged with W[·,·]:
        for i := 1 to 2 do
            if (M_k[i,i] < W[i,i]) then
                M_k[i,·] := W[i,·];   copy vector for (i, W[i,i])
            end_if
        end_for
        if (k = 1 ∨ k = 2) then
            M_k[k,k] := M_k[k,k] + 1;        increment local clock
            M_k[k,·] := diagonal(M_k);
        end_if
```

Figure 4.3: Algorithm for maintaining matrix clock $M_k[1..2, 1..2]$ at P_k, $1 \leq k \leq N$

k^{th} diagonal element corresponds to some interval on P_k. Second, row i is the value of P_i's vector clock in interval $(i, M_k[i,i])$ and third, row k of M_k equals the diagonal. These three rules are stated and proven in lemma 4.1.

Lemma 4.1 *(M1)* $M_k^n[k,k] = n$

(M2) $i \neq j \;\Rightarrow\; (j, M_k^n[i,j]) = pred.(i, M_k^n[i,i]).j$

(M3) $M_k^n[i,i] = M_k^n[k,i]$

Proof: Each part is proven by induction. In the base case, $n = 1$ and (k,n) is an initial interval. Induction applies to $n > 1$.

(M1) **Base** $(n = 1)$: Initial value of $M_k^n[k,k] = 1$.

Induction $(n > 1)$: We assume $M_k^n[k,k] = n$ and show that $M_k^{n+1}[k,k] = n+1$. P_k enters interval $(k, n+1)$ only after sending or receiving a message. From the program text it is clear the k^{th} diagonal element is incremented by one. Thus $M_k^{n+1}[k,k] = M_k^n[k,k] + 1 = n + 1$.

(M2) **Base** $(n = 1)$: Since $n = 1$, M_k^n is the initial value of the matrix clock in P_k. Thus, $i \neq j$ implies that $M_k^n[i,j] = 0$, and hence $(j, M_k^n[i,j]) = (j, 0) = \perp$. Thus we need to show $pred.(i, M_k^n[i,i]).j = \perp$ also. Suppose $i \neq k$, then by initial value of M_k^n, we know $(i, M_k^n[i,i]) = (i, 0)$, thus $pred.(i, M_k^n[i,i]).j = pred.(i, 0).j = \perp$. Now suppose $i = k$, then by initial value of $M_k^n[i,i]$ we know that $(i, M_k^n[i,i]) = (i, 1)$. Since no interval precedes $(i, 1)$, we know $pred.(i, 1).j = \perp$.

Induction $(n > 1)$: Assume true for (k,n) and every interval in the past of (k,n). Suppose the event initiating $(k, n+1)$ is a message send, then $M_k^{n+1} = M_k^n$ except for $M_k^{n+1}[k,k] = M_k^n[k,k] + 1$. Since the event is a message send, (k, n) and $(k, n+1)$ have the exact same predecessors. Thus $pred.(i, M_k^{n+1}[i,i]).j$ equals $pred.(i, M_k^n[i,i]).j$, which by induction hypothesis, equals $(j, M_k^n[i,j])$, which is equal to $(j, M_k^{n+1}[i,j])$ since $i \neq j$ and only element $M_k^n[k,k]$ changes.

Suppose the event initiating $(k, n+1)$ is a message receive tagged with matrix clock W. By the induction hypothesis, W satisfies M2. Note also that M2 relates elements in a single row of M, thus by copying rows from W to M, M2 is not violated. Since all rows $i \neq k$ are either copied from W or not changed, then M2 holds for rows $i \neq k$ in M_k^{n+1}. It remains to show that M2 holds for row k of M_k^{n+1}. The predecessor of $(k, n+1)$ on P_j is the greater of $M_k^n[k,j]$ and $W_k^n[j,j]$. This is exactly the value assigned to $M_k^{n+1}[k,j]$. Thus $(k, M_k^{n+1}[k,j]) = pred.(k, M_k^{n+1}[k,k]).j$ and M2 holds for row k.

(M3) **Base** $(n = 1)$: True by initial assignment to matrix.

Induction $(n > 1)$: Assume $diag(M_k^n) = M_k^n[k,\cdot]$, show $diag(M_k^{n+1}) = M_k^{n+1}[k,\cdot]$. P_k enters interval $(k, n+1)$ only after sending or receiving a message. In the case of a send, $M_k^n[k,k]$ is incremented by one. Thus the diagonal will still be equal to row k. In the case of a message receive, the last statement sets row k to the diagonal.

■

Lemma 4.2 shows how to use the matrix clock M_1^n to determine lo_1, hi_1, lo_2, and hi_2 for any interval $(1, n)$. The procedure for any interval on P_2 is similar.

Lemma 4.2 *The following expressions hold for any interval $(1,n)$ with matrix clock M_1^n:*

$$
\begin{array}{lclcl}
(1,hi_1) & = & pred.(1,n).1 & = & M_1^n[1,1]-1 \\
(2,hi_2) & = & pred.(1,n).2 & = & M_1^n[1,2] \\
(1,lo_1) & = & pred.(2,hi_2).1 & = & M_1^n[2,1] \\
(2,lo_2) & = & pred.(1,hi_1).2 & = & M_1^{n-1}[1,2]
\end{array}
$$

Proof: The equivalences on the left are the previously stated definitions for hi_i and lo_i.

hi_1: The predecessor for $(1,n)$ on P_1 is $(1,n-1)$. Thus, by M1, $hi_1 = M_1^n[1,1]-1$.

hi_2: Also by M1, we know $pred.(1,n).2 = pred.(1,M_1^n[1,1]).2$, which by M2, equals $(2,M_1^n[1,2])$. Thus $hi_2 = M_1^n[1,2]$.

lo_1: We need to evaluate $pred.(2,M_1^n[1,2]).1$. Since the diagonal equals row 1, this is equal to $pred.(2,M_1^n[2,2]).1$. And by M2, this equals $(1,M_1^n[2,1])$. Thus $lo_1 = M_1^n[2,1]$.

lo_2: We need to evaluate $pred.(1,M_1^n[1,1]-1).2$, which equals $pred.(1,n-1).2$. Using the matrix clock M_k^{n-1} in the preceding interval, this equals $pred.(1,M_1^{n-1}[1,1]).2$, which by M2, equals $(2,M_1^{n-1}[1,2])$. Thus $lo_2 = M_1^{n-1}[1,2]$.

■

We have shown how to determine the values of lo_1, hi_1, lo_2, and hi_2 in any interval at P_1 or P_2. Thus for the remainder of this chapter we refer directly to lo_1, hi_1, lo_2, and hi_2 instead of referring to the matrix clock or the *pred* function.

4.4.4 Preliminary Results

The following lemma is important in developing efficient algorithms. It indicates that we need not maintain the value of x_i in every state at process P_i, but instead we can maintain the maximum value of x_i in each interval. As before, we use the notation $maxx.(i, n_i)$ to represent the maximum value of x_i in interval (i, n_i). Formally, $maxx.(i, n_i) = (max\, \sigma : \sigma \in (i, n_i) : \sigma.x)$. Figure 4.4 shows an example of this lemma in action.

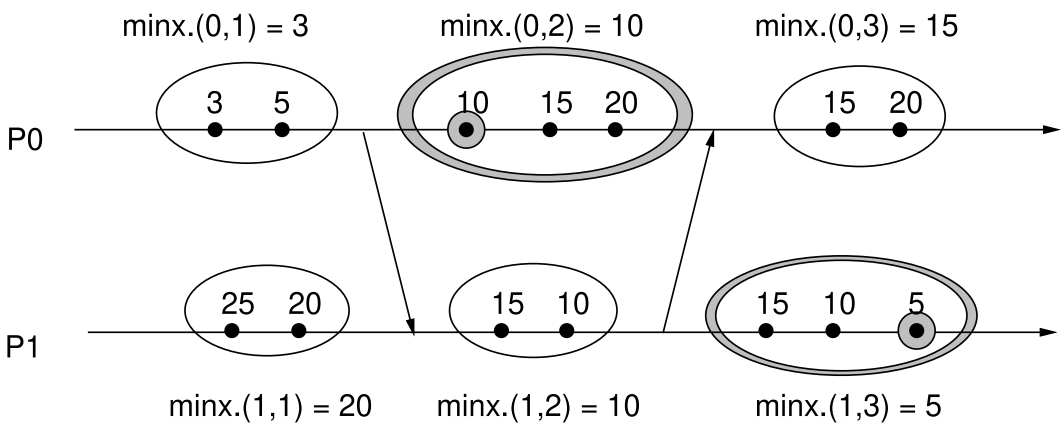

Figure 4.4: Illustration of lemma 4.3.

Lemma 4.3 $(max\, \sigma_1, \sigma_2 \,:\, \sigma_1 \in P_1 \wedge \sigma_2 \in P_2 \wedge \sigma_1 \parallel \sigma_2 \,:\, \sigma_1.x \oplus \sigma_2.x) = (max\, n_1, n_2 : (1, n_1) \parallel (2, n_2) : maxx.(1, n_1) \oplus maxx.(2, n_2))$

Proof:

$(max\, \sigma_1, \sigma_2 : \sigma_1 \in P_1 \wedge \sigma_2 \in P_2 \wedge \sigma_1 \parallel \sigma_2 : \sigma_1.x \oplus \sigma_2.x)$

$= \left\{ \begin{array}{l} \sigma_1 \parallel \sigma_2 \text{ if and only if there exist } n_1, n_2\text{, such that } \sigma_x 1 \in \\ (1, n_1),\ \sigma_2 \in (2, n_2)\text{, and } (1, n_1) \parallel (2, n_2). \end{array} \right\}$

$(max\, \sigma_1, \sigma_2, n_1, n_2 : \sigma_1 \in (1, n_1) \wedge \sigma_2 \in (2, n_2) \wedge (1, n_1) \parallel (2, n_2) : \sigma_1.x \oplus \sigma_2.x)$

$$= \left\{ \begin{array}{l} \text{Since } max \text{ is commutative and associative, we can rearrange} \\ \text{the order of evaluation.} \end{array} \right\}$$

$(max\, n_1, n_2 : (1, n_1) \parallel (2, n_2) : (max\, \sigma_1, \sigma_2 : \sigma_1 \in (1, n_1) \wedge \sigma_2 \in (2, n_2) : \sigma_1.x \oplus \sigma_2.x))$

$$= \left\{ \begin{array}{l} \text{In the previous step, we are finding the } max \text{ of the sum of} \\ \text{all pairings of elements in } (1, n_1) \text{ and } (2, n_2). \text{ Since } \oplus \text{ is} \\ \text{monotonic, this is the same as finding the } max \text{ of } (1, n_1) \\ \text{and } (2, n_2) \text{, and then adding them.} \end{array} \right\}$$

$(max\, n_1, n_2 : (1, n_1) \parallel (2, n_2) : (max\, \sigma_1 : \sigma_1 \in (1, n_1) : \sigma_1.x) \oplus (max\, \sigma_2 : \sigma_1 \in (2, n_2) : \sigma_2.x))$

$$= \{ \text{ By definition of } maxx. \qquad \}$$

$$(max\, n_1, n_2 : (1, n_1) \parallel (2, n_2) : maxx.(1, n_1) \oplus maxx.(2, n_2))$$

■

The next lemma justifies our use of P_1^m and P_2^m in monitoring GF1. It states that if two intervals $(1, i)$ and $(2, j)$ are concurrent then there exists a message m which defines sequences P_1^m and P_2^m and that $(1, i) \in P_1^m$ and $(2, j) \in P_2^m$. It also states that from each message m we can find P_1^m and P_2^m and that every interval in P_1^m is concurrent with every interval in P_2^m. This lemma proves that our approach is sound and complete. See figure 4.5 for a graphical representation of the case (in the proof) where KEY is in P_1.

The lemma requires that P_2 sends a message to P_1 immediately before P_2 terminates and delivery of this message is delayed until immediately before P_1 terminates. This ensures that there is at least one state (the last state in P_1) which is capable of "seeing" the entire computation. We call this message the `<FINAL>` message.

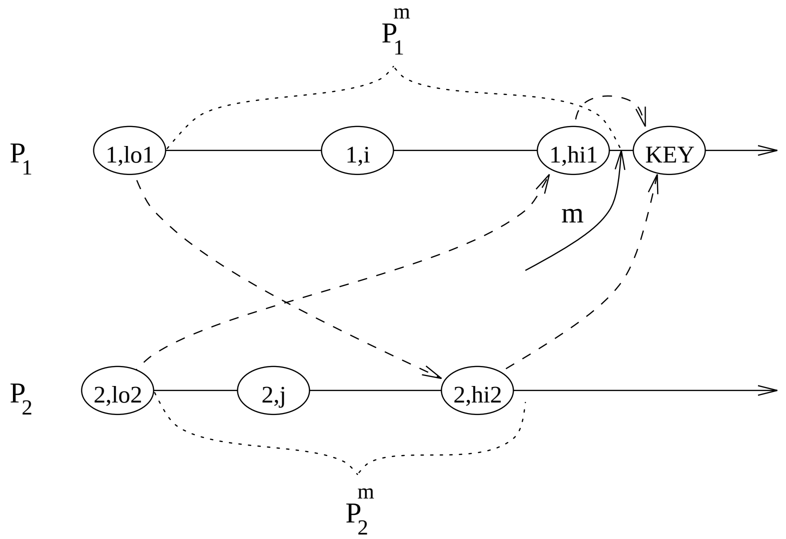

Figure 4.5: Relationship among intervals when KEY is in P_1. Dashed arrows represent the *pred* function, and the solid arrow represents some message being received at P_1.

Lemma 4.4 *Given that the* `<FINAL>` *message exists: two intervals* $(1, i)$ *and* $(2, j)$ *are concurrent* if and only if *there exist a message received just before some interval, say* KEY*, at* P_1 *or* P_2 *such that* $(1, i) \in P_1^m$ *and* $(2, j) \in P_2^m$ *where:*

$$\begin{aligned}
P_1^m &= (\ (1, lo_1 + 1), \ldots (1, hi_1)\ ) \\
P_2^m &= (\ (2, lo_2 + 1), \ldots (2, hi_2)\ ) \\
(1, hi_1) &= pred.KEY.1 \\
(2, hi_2) &= pred.KEY.2 \\
(1, lo_1) &= pred.(2, hi_2).1 \\
(2, lo_2) &= pred.(1, hi_1).2
\end{aligned}$$

Proof:

Proof of $\Rightarrow$: Assume $(1,i) \| (2,j)$.

Due to the <FINAL> message, we know that every state in P_2 happens before some state in P_1. This implies that $succ.(2,j).1$ exists. If $succ.(1,i).2$ exists and $succ.(1,i).2 \rightarrow succ.(2,j).1$, then let $KEY = succ.(1,i).2$ (case 1), otherwise let $KEY = succ.(2,j).1$ (case 2). We prove case 2 (the proof for case 1 is nearly identical). At this point we know that: KEY is the first interval on P_1 such that $(2,j) \rightarrow KEY$, and $succ.(1,i).2$ doesn't precede KEY. Note that $succ.(1,i).2$ may not exist, but it is still true that it $succ.(1,i).2$ doesn't precede KEY.

Proof of $i \leq hi_1$: We know $(1,i)$ must occur before KEY (otherwise $(1,i) \rightarrow (2,j)$). This is equivalent to saying $i \leq hi_1$ where $(1,hi_1) = pred.KEY.1$.

Proof of $lo_2 < j$: Since KEY is the first interval on P_1 such that $(2,j) \rightarrow KEY$, and since $(1,hi_1)$ precedes KEY, we know $(2,j) \not\rightarrow (1,hi_1)$. This is equivalent to saying $lo_2 < j$ where $(2,lo_2) = pred.(1,hi_1).2$.

Proof of $j \leq hi_2$: By definition of predecessors, any interval on P_2 which occurs after KEY's predecessor on P_2, say $(2,hi_2)$, does not happen before KEY. Then since $(2,j) \rightarrow KEY$ we know that $(2,j)$ does not happen after $(2,hi_2)$. This is equivalent to saying $j \leq hi_2$ where $(2,hi_2) = pred.KEY.2$.

Proof of $lo_1 < i$: Since $succ.(1,i).2$ does not precede KEY (as a result of assuming case 2; also note: it may be the case that $succ.(1,i).2$ doesn't exist) and since $(2, hi_2) \rightarrow KEY$, we know that $(1, i)$ cannot precede any interval which precedes $(2, hi_2)$. Therefore $lo_1 < i$ where $(1, lo_1) = pred.(2, hi_2).1$. (If we assumed case 1, this same argument would apply).

Proof of $\Leftarrow$: Assume $lo_1 < i \leq hi_1$ and $lo_2 < j \leq hi_2$.
Since $(1, lo_1) = pred.(2.hi_2).1$ we know that any interval on P_1 which follows lo_1 cannot happen before $(2, hi_2)$. Therefore since $(1, lo_1) \rightarrow (1, i)$ (by $lo_1 < i$) we know $(1, i) \not\rightarrow (2, hi_2)$. And since $j \leq hi_2$ we know that $(1, i) \not\rightarrow (2, j)$.
Similarly, $(2, j) \not\rightarrow (1, i)$. ∎

4.4.5 Overview of Algorithms

The centralized and the decentralized algorithms each gather the same information from the underlying program as it executes. They differ in what they do with the information. This section explains what the information is, and how it is gathered. Everything in this section applies to both algorithms (i.e., both algorithms for the two variable problem).

The 2×2 matrix clock is maintained as described earlier. Each time a message is received at P_1 or P_2, the matrix clock is used to determine the values lo_1, lo_2, hi_1, and hi_2. These values define the sequences of intervals P_1^m and P_2^m which have the property that every interval in P_1^m is concurrent with every interval in P_2^m. This allows us to efficiently determine the maximum value of the global function when restricted to cuts which contain local states in P_1^m and P_2^m. This value is referred to as K_m. The value of $GF1$ is the maximum value of K_m over all messages m received at P_1 or P_2.

Both the centralized and decentralized algorithms use this approach, however only the decentralized algorithm explicitly calculates each value of K_m.

4.4.6 Decentralized Algorithm

The decentralized algorithm is quite simple. We show how to evaluate K_m for any message m received at P_1. A mirror image procedure is used for messages received at P_2. The maximum of all K_m (i.e., over all messages received at P_1 and P_2) equals $GF1$.

<u>Upon receiving an application message m at P_1</u>

Update 2×2 matrix clock;

Calculate lo_1, lo_2, hi_1, and hi_2 from 2×2 matrix clock;

$temp_1 := (\ \max\ i\ :\ lo_1 < i \leq hi_1\ :\ maxx.(1, i)\ );$

send system message $< FinishCalc, temp_1, lo_2, hi_2 >$ to P_2;

<u>Upon receiving system message $< FinishCalc, temp_1, lo_2, hi_2 >$ at P_2</u>

$temp_2 := (\ \max j\ :\ lo_2 < j \leq hi_2\ :\ maxx.(2, j)\ );$

$K_m := temp_1 \oplus temp_2;$

In the next theorem, it is proven that the value produced by this algorithm is equal to $GF1$.

Theorem 4.1 *Let $Kmax$ be the maximum value of K_m for all messages m which are received by P_1 or P_2. Then $Kmax = GF1$.*

Proof: We prove parts 1 and 2, which together imply the desired result.

Part 1: $\sigma_1 \parallel \sigma_2$ implies $Kmax \geq \sigma_1.x \oplus \sigma_2.x$.

Part 2: There exists σ_1, σ_2 such that $\sigma_1 \parallel \sigma_2$ and $Kmax = \sigma_1.x \oplus \sigma_2.x$.

Part 1: Suppose $\sigma_1 \parallel \sigma_2$. Then there exist concurrent intervals $(1, i)$ and $(2, j)$ which contain σ_1 and σ_2. And by lemma 4.4, there exist a message m such that $lo_1 < i \leq hi_1$ and $lo_2 < j \leq hi_2$. After the above code is executed as a result of receiving m, $temp_1 \geq \sigma_1.x$ and $temp_2 \geq \sigma_2.x$. Thus, by monotonicity of *oplus*, K_m, which equals $temp_1 \oplus temp_2$, must be greater than or equal to $\sigma_1.x \oplus \sigma_2.x$. And since $Kmax \geq K_m$, then $Kmax \geq \sigma_1.x \oplus \sigma_2.x$.

Part 2: The value for $Kmax$ must equal K_m for some message m. This message m defines sequences of intervals P_1^m and P_2^m in which all intervals in P_1^m are concurrent with all intervals in P_2^m. Due to the relationship between intervals and local states, all local states in P_1^m are concurrent with those in P_2^m. Let σ_1 be the state with the smallest value of x_1 for all states in P_1^m. Define σ_2 similarly. Then $temp_1 = \sigma_1$ and $temp_2 = \sigma_2.x$. Therefore, since $K_m = temp_1 \oplus temp_2$, then $Kmax = \sigma_1.x \oplus \sigma_2.x$. ∎

Overhead and Complexity: Let A_i be the number of application messages sent and received by P_i, and let R_i be the number of application messages received by P_i. The message overhead consists of the number and size of the debug messages, and the size of message tags on application messages. P_1 sends one debug message to P_2 in each of the R_1 receive intervals at P_1. Similarly, P_2 generates R_2 debug messages. Thus the total number of debug messages generated by the decentralized algorithm is $R_1 + R_2$. The size of each debug message is 3 integers. Each application message carries a tag of 4 integers. The debug messages can be combined to reduce message overhead, however this will increase the delay in the evaluation of the global function.

The memory overhead in P_1 arises from the need to maintain $maxx.(1,\cdot)$ for each of A_1 intervals. This can be reduced (for the average case) by a clever implementation since the elements of the array are accessed in order (i.e., the lower elements can be discarded as the computation proceeds). Likewise, the memory overhead for P_2 is A_2. Other processes incur only the overhead needed to maintain the matrix clock (i.e., 4 integers).

The computation overhead at P_1 consists of monitoring the local variable x_1, and evaluation of the expression $(max\, i : lo_1 < i \leq hi_1 : maxx.(1,i))$ for each debug message sent (R_1) and received (R_2). The aggregate complexity of this is at most $A_1(R_1 + R_2)$ since there are A_1 elements in $maxx.(1,\cdot)$. P_2 has similar overhead. Other processes have neither the overhead of monitoring local variables nor of computing the expression.

4.4.7 Centralized Algorithm

This version of the algorithm can be used as a checker process which runs concurrently with the underlying program, or which runs post-mortem. We describe the post-mortem version which reads data from trace files generated by P_1 and P_2. Since the trace files are accessed sequentially, the algorithm can be easily adapted to run concurrently with the underlying program by replacing file I/O with message I/O. First we explain what data is stored in the trace files, then we show how the global function can be evaluated by one process which has access to both trace files.

Let R_1 be the number of receive intervals in P_1 and let $Q1[k]$, $1 \leq k \leq R_1$, be a record containing the values of lo_1,hi_1,lo_2, and hi_2 in the k^{th} receive interval. Define R_2 and $Q2[1 \ldots R_2]$ similarly. The elements of both $Q1$ and $Q2$ must be checked to determine if one of the elements represents a key interval (i.e., satisfies the requirements of KEY in lemma 4.4.

By virtue of their construction, both $Q1$ and $Q2$ are already sorted in terms of all their fields. That is, for $Q = Q1$ or $Q = Q2$, and for every component $f \in \{lo_1, lo_2, hi_1, hi_2\}$, $Q[\cdot].f$ is a sorted array. This results from the fact that the elements are generated in order on a single process; thus the receive interval represented by $Q[k]$ happens before $Q[k+1]$.

P_1's trace file contains two arrays of data: $maxx.(1, i)$ for each interval $(1, i)$, and $Q1[1..R_1]$. Likewise P_2's trace file contains $maxx.(2, j)$ and $Q2[1..R_2]$. We have already demonstrated how to determine lo_1,hi_1,lo_2 and hi_2 for any interval in P_1 or P_2, and generating the values for the $maxx$ function is straight forward.

The trace files are analyzed in two independent passes. We describe a function $check(Q[1 \ldots R])$ such that K equals the maximum of $check(Q1[1 \ldots R_1])$ and $check(Q2[1 \ldots R_2])$. $Check$ uses two heaps: $heap_1$ and $heap_2$. $Heap_p$ contains tuples of the form $\langle n, maxx.(p, n)\rangle$ where (p, n) is an interval in P_p. The first element of a tuple h is accessed via $h.interval$; the second element is accessed via $h.value$. The heap is sorted with the $value$ field.

The algorithm maintains the following properties (HEAP holds at all times; I1 and I2 hold after S4 and before S5; k is a program variable):

$$\begin{aligned} HEAP &\equiv (\forall h, p : h \in heap_p : heap_p.top().value \geq h.value) \\ I1 &\equiv (\forall i, p : Q[k].lo_p < i \leq Q[k].hi_p : \langle i, maxx.(p,i)\rangle \in heap_p) \\ I2 &\equiv (\forall p :: Q[k].lo_p < heap_p.top().interval \leq Q[k].hi_p) \end{aligned}$$

HEAP is an inherent property of heaps: the top element, heap.top(), is the maximum element in the heap. Heaps are designed to efficiently maintain the maximum element of an ordered set. Statements S1 and S2 ensure I1, which states that in the k^{th} iteration of the *for* loop, all intervals in the sequences defined by $Q[k]$ are represented in the heaps. Statements S3 and S4 ensure I2, which states that the top of $heap_p$ is in the sequence defined by $Q[k]$. The text of the *check* function is shown in figure 4.6 and theorem 4.2 proves its correctness.

Theorem 4.2 *There exists a value for program variable k such that at statement S5* $(heap_1.top().value \oplus heap_2.top().value > K)$ if and only if $(\exists \sigma_1, \sigma_2 : \sigma_1 \in P_1 \wedge \sigma_2 \in P_2 \wedge \sigma_1 \parallel \sigma_2 : \sigma_1.x \oplus \sigma_2.x > K)$

Proof:

Proof of $\Rightarrow$: Let $\langle i, maxx.(1,i)\rangle = heap_1.top()$ and $\langle j, maxx.(2,j)\rangle = heap_2.top()$. By lemma 4.3, it suffices to show that $(1,i)\|(2,j)$. By I2 we know that $Q[k].lo_1 < i \leq Q[k].hi_1$ and $Q[k].lo_1 < i \leq Q[k].hi_1$. Then by lemma 4.4 we know that $(1,i)\|(2,j)$.

Proof of $\Leftarrow$: Assume $(\exists \sigma_1, \sigma_2 : \sigma_1 \in P_1 \wedge \sigma_2 \in P_2 \wedge \sigma_1 \parallel \sigma_2 : \sigma_1.x \oplus \sigma_2.x > K)$.

Figure 4.6: *Check* function used in centralized algorithm for two integer variables.

```
function check(Q[1...R])
    K := unit element of max;
    n1 := 1;   n2 := 1;
    for k := 1 to R do
S1:     while (n1 ≤ Q[k].hi1) do
            heap1.insert( ⟨n1, maxx.(1, n1)⟩ );
            n1 := n1 + 1;
        end_while
S2:     while (n2 ≤ Q[k].hi2) do
            heap2.insert( ⟨n2, maxx.(2, n2)⟩ );
            n2 := n2 + 1;
        end_while
S3:     while (heap1.top().interval ≤ Q[k].lo1) do
            heap1.removetop();
        end_while
S4:     while (heap2.top().interval ≤ Q[k].lo2) do
            heap2.removetop();
        end_while
S5:     if (heap1.top().value ⊕ heap2.top().value > K) then
            K := heap1.top().value ⊕ heap2.top().value;
        end_if
    end_for
    return K;
end_function
```

By lemma 4.3 we know $(\exists i, j : (1,i) \parallel (2,j) : maxx.(1,i) \oplus maxx.(2,j) > K)$. Referring to lemma 4.4, if the KEY interval is in P_1 then let $Q = Q1$ and let k equal the number of messages received at P_1 before KEY. Likewise for KEY in P_2. Then $Q[k]$ corresponds to KEY. Then by lemma 4.4 $Q[k].lo_1 < i \leq Q[k].hi_1$ and $Q[k].lo_2 < j \leq Q[k].hi_2$. By invariant I1, $\langle i, maxx.(1,i)\rangle$ is in $heap_1$, and $\langle j, maxx.(2,j)\rangle$ is in $heap_1$. And by invariant HEAP, $heap_1.top().value \geq maxx.(1,i)$, and $heap_2.top().value \geq maxx.(2,j)$. Thus in iteration k, $heap_1.top().value \oplus heap_2.top().value > K$. ∎

Overhead and Complexity: If we consider each record written to a trace file to be a debug message then the message complexity analysis is identical to the decentralized algorithm (except that the debug messages have a different destination).

P_1 and P_2 do not need to maintain $maxx.(\cdot,\cdot)$, thus the only memory overhead for each application processes is the 4 integers needed for the matrix clock.

The computation overhead consists of monitoring the local variables. The rest of the computation is offloaded to the checker process which uses the following data: $Q1[1 \ldots R_1]$, $Q2[1 \ldots R_2]$, $maxx.(1,i)$ for $1 \leq i \leq A_1$, and $maxx.(2,j)$ for $1 \leq j \leq A_2$. Recall from section 4.4.6 that R_i is the number of messages received by P_i and A_i equals R_i plus the number of messages sent by P_i .

Consider the call $check(Q1[1 \ldots R_1])$. On a heap of N elements, $insert()$ and $removetop()$ each cost $\Theta(lg\, N)$ and $top()$ costs $\Theta(1)$. Each element of $maxx.(1,\cdot)$ is inserted at most once and removed at most once from $heap_1$ for a total cost of $\Theta(A_1\, lg\, A_1)$. Similarly, the total cost of operations on $heap_2$ is $\Theta(A_2\, lg\, A_2)$. The outer loop executes R_1 times but is added to the cost of the heap operations since the heap operations are spread out through all R_1 iterations. Thus the total cost of $check(Q1[1\ldots R_1])$ is $\Theta(R_1 + A_1\, lg\, A_1 + A_2\, lg\, A_2)$. Since $R_1 \leq A_1$, this simplifies to $\Theta(A\, lg\, A)$ where $A = max(A_1, A_2)$. Since $check$ is only called twice, the total complexity is $\Theta(A\, lg\, A)$.

4.5 Bivalent Variables

In this section we consider the case when D_l contains exactly two elements, $\{V_0, V_1\}$, such that V_0 is the identity element of $\oplus$ (i.e., $V_0 \oplus v = v$). We want to determine if the value of GF ever exceeds some constant $z \in D$. First we show this problem is equivalent to determining if there exists an antichain of size K, which is conceptually a much simpler problem. First we demonstrate this equivalence, and then go about developing an algorithm and proof for finding the largest antichain.

The goal is to determine if there exists an antichain C such that $gf(C) > z$. Let $\#C$ denote the number of V_1's in C (i.e., the size of the set $\{\sigma \mid \sigma \in C \wedge \sigma.x == V_1\}$). Since V_0 is the identity element, the value of $gf(C)$ depends only on $\#C$. Assume that $V_0 < V_1$ (a similar argument to that which follows can be made for the case where $V_0 > V_1$). Then

$$\#C_1 > \#C_2 \iff gf(C_1) > gf(C_2)$$

It is easy to find a value K such that

$$\#C > K \iff gf(C) > z$$

Thus we can solve the problem by finding any antichain C such that $\#C > K$. Now suppose we remove all local states from P except those which have $\sigma.x == V_1$. Now for any antichain in P,

$$|C| > K \iff gf(C) > z$$

Therefore we can use the following as the problem statement for this section:

$$(\exists C : C \in \overline{P} : |C| \geq K) \tag{GF2}$$

The techniques used to solve GF2 are based on Dilworth's theorem [12] which shows that the maximum size antichain in a poset equals the minimum number of chains which cover the poset. It is known that if the poset is given in the form of an acyclic graph, then max-flow techniques [36] can be used to compute the width [2]. However, to the best of our knowledge, the number of comparisons required to compute the width is not known if the poset is given in a chain-decomposed form. Two special cases of this problem ($K = 2$ and $K = N$) are solved in [15].

This is one of the fundamental problems in deposets. It has many applications in distributed debugging. For example, we use the solution of the above problem to detect the unstable predicate of the form

$$x_1 + x_2 + ... + x_N \geq K$$

where $x_i \in \{0, 1\}$ is a variable in the process P_i for all i. As an example, consider the K-mutual exclusion problem. This problem requires a synchronization protocol such that no more than K processes are ever in the critical section. If x_i represents the fact that process P_i is in the critical section, then detection of the global predicate $x_1 + x_2 + ... + x_N \geq K + 1$ is equivalent to detection of violation of K-mutual exclusion. This predicate is also useful if the number of resources (for example, the number of copies for a copyrighted program) in a network is constrained to be at most K.

4.5.1 Algorithm

We are given P already partitioned into N chains, P_i, and we need to determine if there exists an antichain of size at least K. It follows from Dilworth's theorem [12] that P can be partitioned into $K-1$ chains if and only if there does not exist an antichain of size at least K. Therefore, we can solve GF2 by trying to partition P into $K-1$ chains. If we succeed then GF2 is false. If we fail then GF2 is true.

So our problem is now reduced to taking the N chains P_i, $1 \leq i \leq N$, and trying to merge them into $K-1$ chains. The approach we take is to choose K chains and try to merge them into $K-1$ chains. After this step we have $N-1$ chains left. We do this step ("choose K chains, merge into $K-1$ chains") $N-K+1$ times. If we fail on any iteration, then P could not be partitioned into $K-1$ chains. Thus there exists an antichain C such that $|C| > K$. If we succeed then we have reduced P to $K-1$ chains and there does not exist an antichain C such that $|C| > K$.

The algorithm uses queues to represent chains. Each queue is stored in increasing order so the head of a queue is the smallest element in the queue.

An element represents a state, and a smaller state "happens before" a larger state (this can be determined by comparing the vector clocks of the states). The algorithm uses the following operations on queues (q represents a queue):

insert(q,e)	insert element e in q
deletehead(q)	remove the head of q
empty(q)	true if q is empty
head(q)	return the first item in q, or a maximal value if q is empty

The algorithm $FindAntiChain$ (figure 4.7) calls the $Merge$ function $N - K + 1$ times. The $Merge$ function takes K queues as input and returns K queues: $Q_1, \ldots, Q_K$. If Q_k is returned empty, then the merge was successful (the result is in $Q_1, \ldots, Q_{K-1}$) and $Merge$ continues to the next iteration. If Q_k is not empty then an antichain has been found and is given by the heads of the returned queues.

There are two important decisions in this algorithm. The first is how to choose the K chains for the merge operation. The answer follows from classical merge techniques used for sorting. We choose the chains which have been merged the fewest number of times. This is why the algorithm rotates $Qlist$ after each merge operation. This reduces the number of comparisons required by the algorithm.

The second and more important decision is how to implement *Merge*. The merge is performed by repeatedly removing an element from one of the K input chains and inserting it in one of the $K - 1$ output chains. Output chain Q_K is only used if $Merge$ fails, thus when we say "output chain" we are only referring the chains which contain successfully merged output: $Q_1, \ldots, Q_{K-1}$. We move the smallest elements first (that is, the ones which represent states

```
function FindAntiChain (Qlist : list of queues of vector clocks,
                        K:integer, ) : antichain;
begin
    assume: 1 ≤ K ≤|Qlist|
    assume: q ∈ Qlist ⇒ ¬empty(q)
    for (n :=|Qlist|; n ≥ K; n := n − 1) do
        (p_1, .., p_N) := Qlist;
        (q_1, ..., q_K) := Merge(p_1, ..., p_K);
        if (empty(q_k))
            then Qlist := (p_{K+1}, .., p_n, q_1, .., q_{k−1});        rotate list
            else return ( {head(q_i) | 1 ≤ i ≤ K} );
        end_if
    end_for
    return(∅);
end_function
```

Figure 4.7: Function that determines if an antichain of size K exists in the poset encoded by the queues of vectors listed in $Qlist$

that occurred earlier in the computation). An element will be moved from the input to the output if it is smaller than an element on the head of some other chain. The problem is deciding on which output chain to place the element.

Note that this is trivial for $K = 2$, when two input queues are merged into a single output queue as is done in the merge sort. For $K > 2$, the number of output queues is greater than one, and the algorithm needs to decide in which output queue the element needs to be inserted. The simple strategy of inserting the element in any output queue does not work as shown in figure 4.9 where there are three input queues (P_1, P_2, P_3) which need to be merged into two output queues (Q_1, Q_2). Suppose we use a simple strategy which results in the operations listed below. Initially Q_1 and Q_2 are empty. Each operation moves an element from the head of P_1, P_2 or P_3 to one of the two output queues. Note that we can only move the head of P_i if it is smaller than the head of some other queue P_j. The operations we perform are:

1. $(1, 0, 0) < (2, 0, 0)$. So move $(1, 0, 0)$ to an output queue, say Q_1.
2. $(0, 1, 0) < (1, 1, 0)$. So move $(0, 1, 0)$ to an output queue, say Q_2.
3. $(1, 1, 0) < (2, 2, 0)$. So move $(1, 1, 0)$ to an output queue, say Q_1.
4. $(1, 2, 0) < (2, 2, 0)$. So move $(1, 2, 0)$ to an output queue, say Q_1.
5. $(2, 0, 0) < (2, 2, 0)$. So move $(2, 0, 0)$ to an output queue, but which one?

Notice that we have worked ourselves into a corner because when we decide to move $(2, 0, 0)$ there is no output queue in which we can insert it. The output queues must be sorted since they represent chains. This is done by inserting the elements in increasing order, but $(2, 0, 0)$ is not larger than the tails of any of the output queues. Thus we have nowhere to insert it.

This situation does not imply that the input queues cannot be merged. In fact in this case they can be merged into two queues as shown on the right side of figure 4.9. It merely implies that we did not intelligently insert the elements in the output queues. The function $FindQ$ chooses the output queue without running into this problem. Discussion of $FindQ$ is deferred until after we describe the details of the $Merge$ function.

The $Merge$ function compares the heads of each input queue with the heads of all other queues. Whenever it finds a queue whose head is less than the head of another queue, it marks the smaller of the two to be deleted from its input queue and inserted in one of the output queues. It repeats this process until no elements can be deleted from an input queue. This occurs when the heads of all the queues are incomparable, that is, they form an antichain. Note that it may be the case that some input queues are empty. If none are empty, then we have found an antichain of size K. The heads of the input queues form the antichain. If one or more are empty, the merge operation (which is not complete yet) will be successful. All that is left to do is to take the non-empty input queues and append them to the appropriate output queues. This is done by the *FinishMerge* function whose implementation is not described in detail because it is straight forward.

The *Merge* algorithm is shown in figure 4.8. Note that it only compares the heads of queues which have not been compared earlier. It keeps track of this in the variable ac, which is a set of indices indicating those input queues whose heads are known to form an antichain. Initially ac is empty. The $Merge$ algorithm terminates when either ac has K elements or one of the input queues is empty.

The first *for* loop in *Merge* compares the heads of all queues which are not already known to form an antichain. That is, we compare each queue not in ac to every other queue. This avoids comparing two queues which are already in ac. Suppose $e_i = head(P_i)$ and $e_j = head(P_j)$ and inside the first *for* loop it is determined that $e_i < e_j$. This implies that e_i is less than all elements in P_j. Thus, e_i cannot be in an antichain with any element in P_j and therefore cannot be in any antichain of size K which is a subset of the union of the input queues. Thus, we can safely move e_i to an output queue, which eliminates it from further consideration. The set $moved$ records which elements will be moved from an input queue to an output queue. The array $bigger$ records the larger of the two elements which were compared. In this example, $bigger[i]$ equals j, implying that the head of P_j is bigger than the head of P_i. This information is used by $FindQ$ to choose the output queue where the head of P_i will be inserted.

The second *for* loop just moves all elements in $move$ to an output queue. Consider the state of the program just before the second *for* loop begins. If the head, e, of an input queue is not marked to be moved, then e is not less than the head of any other input queue or else it would have been marked to be moved. This implies that any two elements which are not moved are concurrent, which in turn implies that the set of heads which are not moved form an antichain. This antichain is recorded in ac for the next iteration of the *while* loop.

We now return to describing how $FindQ$ works. The formal description of $FindQ$ is shown in figure 4.11. Given the queue which contains the element to be moved, and the queue with which this element was compared (it must be smaller than the head of another queue in order to move it), the procedure $FindQ$ determines which output queue to use. The $FindQ$ function takes three parameters:

G	:	an undirected graph called *queue insert graph*
i	:	the input queue from which the element x is to be deleted
j	:	the queue in which all elements are bigger than x

A "queue insert graph" is used to deduce the queue in which the next element is inserted. It has K vertices and exactly $K-1$ edges. An edge corresponds to an output queue and a vertex corresponds to an input queue. Therefore, each edge (i,j) has a label, $label(i,j) \in \{1,\ldots,K-1\}$, which identifies the output queue it corresponds with. No labels are duplicated in the graph, thus each output queue is represented exactly once. Similarly, each input queue is represented exactly once.

An edge (i,j) between vertex i and vertex j means that the heads of P_i and P_j are both bigger than the tail of $Q_{label(i,j)}$. The goal is to ensure that for any input queue (i.e. any vertex) there exists an output queue (i.e. an edge) in which the head of input queue can be inserted. This constraint is equivalent to the requirement that every vertex has at least one edge adjacent to it. It is also equivalent to the requirement that the graph is acyclic since there are K nodes and $K-1$ edges.

```
function Merge(P_1, ..., P_K:queues) : Q_1, ..., Q_K: queues;
const all = {1,... K};
var  ac,move: subsets of all;
     bigger: array[1..k] of 1..k;
     G: initially any acyclic graph on k-1 vertices;
begin
     ac := ∅;
     while (|ac| ≠ K ∨ ¬(∃i : 1 ≤ i ≤ K : empty(P_i))) do
          move := {};
          for i ∈ all - ac and j ∈ all do
               if head(P_i) < head(P_j) then
                    move := move ∪ {i};
                    bigger[i] := j;
               end_if;
               if head(P_j) < head(P_i) then
                    move := move ∪ {j};
                    bigger[j] := i;
               end_if;
          end_for
          for i in move do
               dest := FindQ(G,i,bigger[i]);
               x := deletehead(P_i)
               insert(Q_dest, x);
          end_for
          ac := all - move;
     end_while
     if (∃i :: empty(P_i)) then
          FinishMerge(G, P_1, ... P_K, Q_1, ... Q_{K-1});
R1:       return (Q_1, ..., Q_{K-1}, ∅);
     else
R2:       return (P_1, ..., P_K);
     end_if
end_function
```

Figure 4.8: Generalized Merge Procedure for deposets

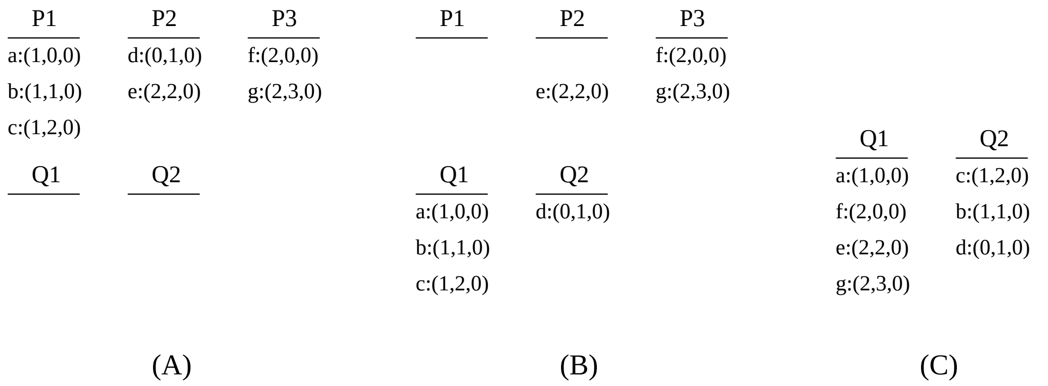

Figure 4.9: An example of a failed naive strategy. Diagram A shows the initial configuration. Diagram B shows the point at which the strategy fails: there is no where to insert $(2, 0, 0)$. Diagram C shows that this example can be merged into two chains.

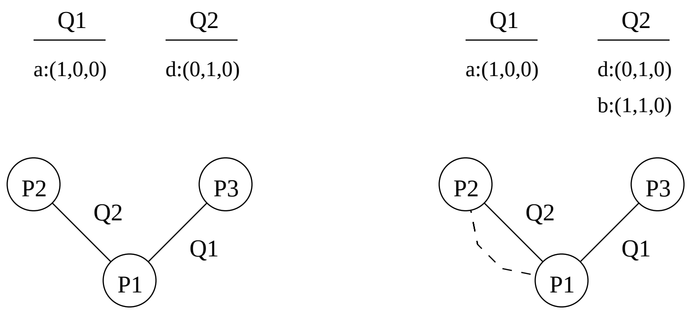

Figure 4.10: Using a queue insert graph to find the output queue.

FindQ uses the queue insert graph as follows. Consider the function call $FindQ(G, i, j)$. The element to be deleted is $e_i = head(P_i)$, and it is smaller than $e_j = head(P_j)$ (i.e., $bigger[i] = j$). *FindQ* adds the edge (i, j) to G. Since G was acyclic before adding edge (i, j), it now contains exactly one cycle which includes (i, j). Let (i, k) be the other edge incident on vertex i which is part of this cycle (it is possible that $k = j$). *FindQ* deletes (i, k) and adds (i, j), and then sets the label of (i, j) equal to the label of (i, k). *FindQ* then returns the label associated with the new edge. This label indicates which queue e_i will be inserted in.

Consider our previous example with the naive algorithm. It made a bad decision when it placed element $(1, 1, 0)$ on Q_1. Figure 4.10 shows the state of the queues and of the graph before and after the correct decision made by *FindQ*. Element $(1, 1, 0)$ is in P_1 and is less than the head of P_2. Thus the edge $(1, 2)$ is added to the graph (dashed line in figure 4.10). It forms a cycle with the other edge $(1, 2)$ which is labeled with Q_2. We copy this label to the new edge, delete the old edge and return Q_2, indicating that $(1, 1, 0)$ should be inserted in Q_2.

An important invariant of the queue insert graph is that given any edge (i, k) and the output queue associated with it, the queue is empty or the tail of the queue is less than all elements in input queues P_i and P_k. This is stated and proven in lemma 4.5 and is used later to show that the output queues are always sorted.

Lemma 4.5 $(i, k) \in G \Rightarrow (\ empty(Q_{label(i,k)}) \vee (\forall e : e \in P_i \cup P_k : tail(Q_{label(i,k)}) \leq e)\ )$

```
function FindQ(G: graph; i,j:1..K) : label;
      add edge (i, j) to G;
      (i, k) := the edge such that (i, j) and (i, k) are part
                 of the same cycle in G;
      remove edge (i, k) from G;
      label(i, j) := label(i, k);
      return label(i, j);
end_function
```

Figure 4.11: Function that finds which output queue to insert an element which is being deleted from the head of input queue P_j. The chosen output queue is $Q_{label(i,j)}$.

Proof: Initially, the lemma holds since all output queues are empty. Now assume that the lemma holds and we need to insert $e_i = head(P_i)$ into output queue Q_l where $l = FindQ(G, i, j)$ and $j = bigger[i]$. Since $j = bigger[i]$, we know $e_i \leq head(P_j)$. Since P_i and P_j are sorted, we know $e_i \leq e$ for any element $e \in P_i \cup P_k$. After moving e_i to Q_l, the tail of Q_l will be e_i and the lemma will still hold. ∎

The merge operation must produce sorted output queues. Lemma 4.6 proves that our algorithm meets this requirement.

Lemma 4.6 *If the elements are inserted in the output queues using* FindQ*, then all output queues are always sorted.*

Proof: Initially all output queues are empty. Now assume each output queue is sorted and we need to move $e_i = head(P_i)$ to an output queue. The $FindQ$ procedure will return $Q_{label(i,j)}$, where (i, j) is some edge in G. By lemma 4.5, the tail of this queue is less than or equal to e_i. Thus after inserting e_i in Q_l, Q_l is still sorted. No other output queues are modified, thus the lemma still holds. ∎

Lemma 4.7 *Suppose* $(Q_1, \ldots Q_K) = Merge(P_1, \ldots, P_K)$ *and each queue* P_i *is sorted. Then*

(A) $empty(Q_K) \Rightarrow P_1, \ldots P_K$ *have been merged into* $K-1$ *sorted queues* $Q_1, \ldots Q_{K-1}$.

(B) $\neg empty(Q_K) \Rightarrow \{head(P_i) \mid 1 \leq i \leq K\}$ *is an antichain of size* K.

(C) $\neg empty(Q_K) \iff$ *there exists an antichain of size* K.

Proof: We use the following invariant of the *while* loop:

$$i \in ac \wedge j \in ac \wedge \neg empty(P_i) \wedge \neg empty(P_j) \Rightarrow head(P_i) \| head(P_j).$$

(A): Suppose $empty(Q_K)$. Then *Merge* returned via statement $R1$ and just before returning there existed i such that $empty(P_i)$. Let P_K be the empty one. Then all elements reside in one of $P_1, \ldots P_{K-1}$ or $Q_1, \ldots Q_{K-1}$. Thus after *FinishMerge* all elements are in $Q_1, \ldots Q_{K-1}$ which by lemma 4.6 are sorted. Thus the original input queues have been merged into $K-1$ output queues.

(B): Suppose $\neg empty(Q_K)$. Then *Merge* returned via $R2$ and just before returning there was no i such that $empty(P_i)$. Then by the invariant and since $ac = all$ we know $head(P_i)$ is concurrent with the head of every other P_j. Thus $\{head(P_i) \mid 1 \leq i \leq K\}$ is an antichain of size K.

(C): Follows from (A), (B), and Dilworth's theorem. ∎

Theorem 4.3 *There exists an antichain of size* K *in* $\cup_{1 \leq i \leq K} P_i$ *if and only if* FindAntiChain$((P_1, \ldots P_K))$ *returns an antichain of size* K*.*

Proof: Assume there exists an antichain which satisfies the left hand side. Then by Dilworth's result, $\cup_{1 \leq i \leq K} P_i$ cannot be merged into $K-1$ chains. Thus by lemma 4.7, *Merge* will return an antichain of size K.

Now assume there does not exist such an antichain. Then $\cup_{1 \leq i \leq K} P_i$ can be merged in $K-1$ chains and *Merge* will return with P_K empty on iteration $N - K + 1$ of the *while* loop, which will cause *FindAntiChain* to return the empty set. ∎

4.5.2 Complexity

In this section, we analyze the complexity based on the number of comparisons required by the algorithm (i.e., the number of times the heads of two queues are compared in the *Merge* function. We prove an upper bound and a lower bound. The lower bound is proven by defining an adversary which produces a set of input queues that forces the merge algorithm to use at least KMN comparisons, where M is the number of elements in the largest input queue.

An Upper Bound:

Theorem 4.4 *The maximum number of comparisons required by the above algorithm is* $KMN(K + log_{\rho} N)$*.*

Proof: We first calculate the complexity of merging K queues into $K-1$ queues. From the *Merge* algorithm it is clear that each element must be in ac before it passes to an output queue and it requires K comparisons to be admitted to ac. Thus, if the total number of elements to be merged is l, then l elements must pass through ac on their way to the output queue for a total of Kl comparisons.

Initially $l \leq KM$ but the queues grow for each successive call to *Merge*. At this point, our technique of rotating the list of queues to be merged is useful. Let $level.i$ denote the maximum number of merge operations that any element in P_i has participated in. The algorithm rotates $Qlist$ to ensure that in each iteration the K queues with the smallest level numbers will be merged. Initially, there are N queues at level 0. Each of the first N/K merge operations reduces K queues with level 0 to $K-1$ queues with level 1. This pattern continues until there are $2K$ queues left, at which time the maximum level will be $log_\rho N$ where ρ is the reducing factor and equals $K/(K-1)$. Merging the remaining $2K$ queues into $K-1$ adds K more levels. Thus the maximum level of any final output queue is $K + log_\rho N$. Thus, there are at most MN elements, each of which participates in at most $K + log_\rho N$ merge operations at a cost of K comparisons per element per merge. Therefore the maximum number of comparisons required by the above algorithm is $KMN(K + log_\rho N)$. ∎

Note that for the special case when $K = 2$, the complexity is $O(mn\ log\ N)$. This is shown to be optimal in [15]. Further, if $M = 1$ and $K = 2$, this reduces to the well-known merge sort algorithm with the complexity of $O(N\ log\ N)$ comparisons. Another noteworthy special case is when $K = N$. In this case, the complexity becomes $O(MN^2)$ which is also known to be optimal [15].

A Lower Bound: In this section, we provide a lower bound on the number of comparisons required by any algorithm to solve the above problem.

Proposition 4.1 *Let $(P,<)$ be any partially ordered finite set of size MN. We are given a decomposition of P into N sets $P_1, ...P_N$ such that P_i is a chain of size M. Any algorithm which determines if there exists an anti-chain of size K must make at least $\Omega(KMN)$ comparisons.*

Proof: We use an adversary argument. Let $P_i[s]$ denote the s^{th} element in the queue P_i. The adversary will give the algorithm P_i's with the following characteristic:

$$(\forall i, j, s :: P_i[s] < P_j[s+1])$$

Formally, on being asked to compare $P_i[s]$ and $P_j[t]$, $(s \neq t)$ the adversary uses:

if $(s < t)$ **then** return $P_i[s] < P_j[t]$
if $(t < s)$ **then** return $P_j[t] < P_i[s]$

Thus, the above problem reduces to M independent instances of the problem which checks if a poset of N elements has a subset of size K containing pairwise incomparable elements. If the algorithm does not completely solve one instance then the adversary chooses that instance to show a poset consistent with all its answers but different in the final outcome.

We now show that the number of comparisons to determine whether any poset of size N has an anti-chain of size K is at least $N(K-1)/2$. The adversary will give that poset to the algorithm which has either $K-1$ or K chains such that any pair of elements in different chains are incomparable. In other words, the poset is a simple union of either $K-1$ or K chains. The adversary keeps a table *numq* such that $numq[x]$ denotes the number of questions asked about the element x. The algorithm for the adversary is shown in Fig. 4.12.

If the algorithm does not ask $K-1$ questions about any element x, the adversary can produce a poset inconsistent with the answer of the algorithm. If the algorithm answered that no anti-chain of size K exists then the adversary can produce an anti-chain which includes one element from each of the $K-1$ chains and the element x. On the other hand, if the algorithm answered that an anti-chain exists, then the adversary could put x and all other elements for which $K-1$ questions have not been asked in $K-1$ chains.

Since each comparison involves two elements, we get that the algorithm must ask at least $N(K-1)/2$ questions for each level. Thus, overall any algorithm must make at least $MN(K-1)/2$ comparisons. ∎

It is easy to see that the lower bound is not tight. If we choose $M=1$ and $K=2$, we get the lower bound of $N/2$. However, the lower bound of $N \log N$ is well known for this case.

```
var numq[N]:integer initially 0;      number of questions asked about x
function  compare (x,y:elements)
      numq[x]++; numq[y]++;
      if (numq[x] = K - 1) then
            chain[x] := chain in which no element has been
                        compared with x so far;
      end_if
      if (numq[y] = K - 1) then
            chain[y] := chain in which no element has been
                        compared with y so far;
      end_if
      if (numq[x] < K - 1) or (numq[y] < K - 1) then
            return x||y;
      end_if
      if (chain[x] ≠ chain[y]) then
            return x||y;
      else
            if x inserted earlier than y then return (x < y);
            else return (y < x);
            end_if
      end_if
end_function
```

Figure 4.12: Algorithm for the Adversary

Chapter 5

RCL: Recursive Logic

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. RCL, and recursive logics in general, 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.

This chapter begins with the syntax and semantics of RCL. Following that are examples of RCL formulas and how they can be applied in the observation of distributed programs. Then, a distributed algorithm for online detection of RCL formulas is presented, proven correct and analyzed for complexity. Two versions of the algorithm are described. A high level functional algorithm is used to prove correctness, and a data flow style implementation is used for complexity analysis.

5.1 Syntax

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 in figure 5.1.

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.

$$\begin{array}{rcl} f & = & S \mid f \triangle f \\ S & = & g \mid g\langle f\rangle S \mid g\langle\langle f\rangle\rangle S \end{array}$$

Figure 5.1: RCL Logic Syntax

5.2 Definitions

Before describing the semantics of RCL, some of the underlying mathematics are explained. 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 $\overrightarrow{A}$. Figure 5.2 shows an example of a closed poset.

$$\overleftarrow{A} \quad \triangleq \quad \{x \mid (\exists y : y \in A : x \rightarrow y \vee x = y)\}$$

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

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. Using '(' or ')' instead of the square braces is similar to an open interval – the "endpoint" is left out of the resulting set.

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

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 (this is proven later). The correctness proof shows that, given a computation P and a formula f, the detection algorithm

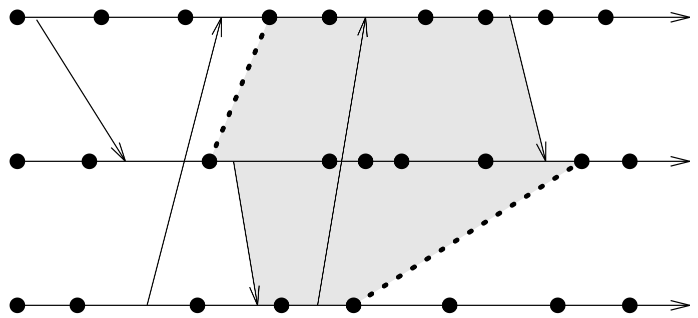

Figure 5.2: The shaded subposet is *closed*. All closed subposets are bounded by antichains as shown by the heavy dotted lines. The set would no longer be closed if any interior state (shaded but not on a boundary antichain) were removed.

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. Refer to figure 5.3 for examples. 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 B and they have no elements in common.

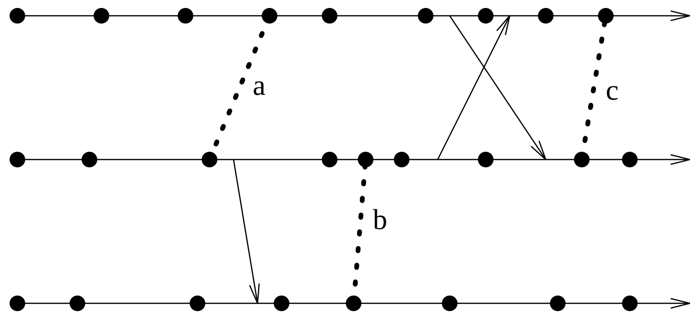

Figure 5.3: Example of the weak and strong precedes relations on cuts. Among the cuts a,b and c, the only pairs which are ordered by these relations are: $a \prec\!\!\prec c$, $a \prec b$, and $a \prec c$.

$$A \prec B \quad \triangleq \quad B \subseteq \overrightarrow{A} \wedge A \cap B = \emptyset$$

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:

$$A \prec\!\!\prec B \quad \triangleq \quad (\forall a, b : a \in A \wedge b \in B : a \rightarrow b)$$

5.3 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. This means that given any two antichains which satisfy the predicate, there exists an antichain which is "less than or equal to" both of these antichains.

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.

Although g can represent any global predicate which meets these requirements, the remainder of this chapter pretends that g denotes a weak conjunctive global predicate (CGP) since it is a well known example and satisfies the requirements. The only properties of g which are used in the proofs are those that result from the aforementioned requirements.

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:

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

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.

$$P \models f_1 \triangle f_2 \stackrel{\triangle}{=} (P \models f_1) \wedge (P \models f_2)$$

The next 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.

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

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

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

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 5.4 and 5.5 show similar examples.

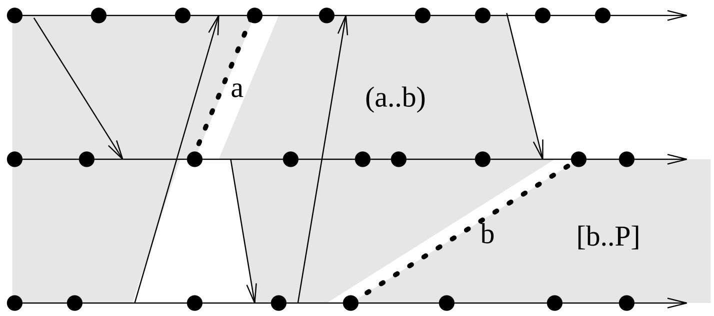

Figure 5.4: Example of a poset structure which could satisfy $g\langle f\rangle g$. The cuts a and b divide the poset into regions as indicated by the shading. Notice that each region is closed.

5.4 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.

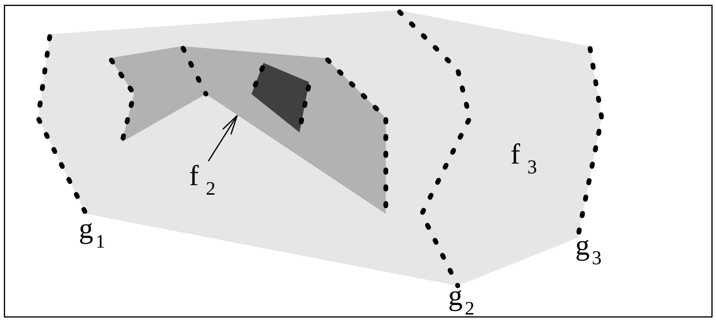

Figure 5.5: Example of a structure which might satisfy $g_1\langle f_2\rangle g_3\langle f_4\rangle g_5$. The CGPs, g_i, need not be full width antichains, which is why they are shown in a "free-form" manner. Each f_i is another RCL formula. Two levels of recursion are shown for f_2, where f_2 has the structure $g\langle f\rangle g\langle g\langle f\rangle g\rangle g$.

5.4.1 Sequence of local predicates

Consider the sequence of local predicates as defined in [43]: $(p_1, p_2, \ldots p_n)$. Each local predicate p_i is a special case of CGP. Therefore, this sequence of local predicates is equivalent to the RCL formula $p_1\langle true\rangle p_2 \ldots \langle true\rangle p_n$.

5.4.2 Weak CGP

Weak CGP [19] are global state based predicates which meet the requirements of the form 'g'. Thus a weak CGP can be represented in RCL as the formula g.

5.4.3 ENF: Event Normal From

Chiou and Korfage [9] 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: $g_1\langle\langle true\rangle\rangle g_2\langle\langle true\rangle\rangle g_3 ... \langle\langle true\rangle\rangle g_n$.

5.4.4 Mutual Exclusion

Violation of mutual exclusion can be written as a CGP. Therefore, since CGP is a special case of RCL, it can also be written as an RCL formula.

5.4.5 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.

5.4.6 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 any antichain which contains exactly one state from each of process 1 and 2. Consider the example shown in figure 5.6. 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.

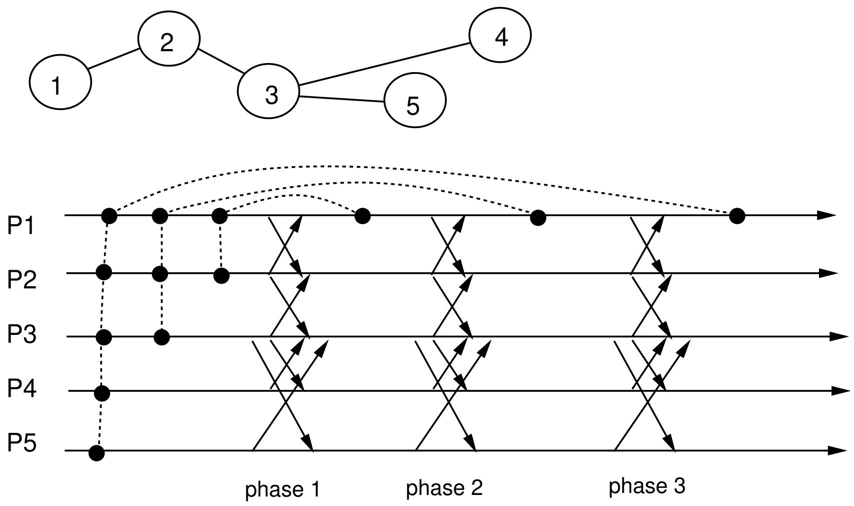

Figure 5.6: Distributed consensus example. See section 5.4.6.

5.4.7 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.

The global predicate g_X is true on any antichain which contains one local state from each process 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 5.7 shows an example.

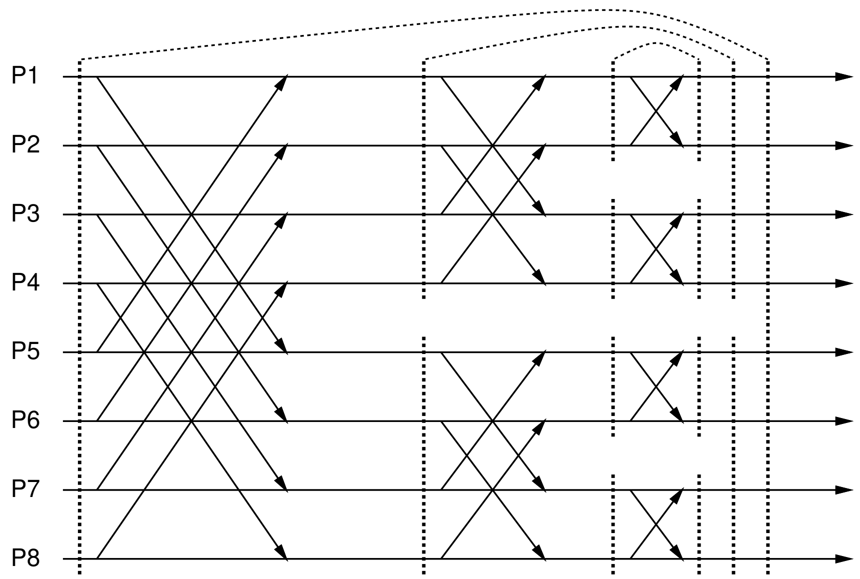

Figure 5.7: Butterfly synchronization example. See section 5.4.7.

5.4.8 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 in section 5.4.6, the formula is shown below. Figure 5.8 shows this example.

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

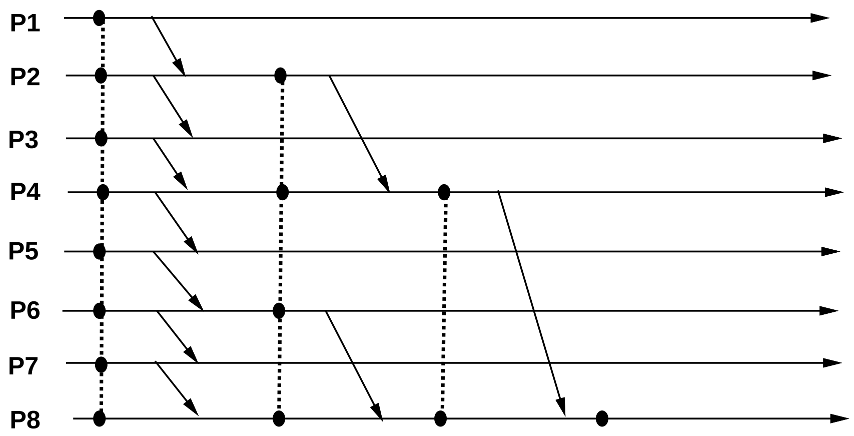

Figure 5.8: Distributed collection example. See section 5.4.8.

5.5 Algorithm

Two versions of the algorithm are presented: the data flow version and the functional version. Both algorithms make use of the subroutines $findCGP$, $advance$, and sup. In the data flow version, each node corresponds to an instantiation of one of these subroutines. The functional version parses a formula and calls the subroutines in an order which is consistent with the data flow version.

The functional version will used in the correctness proof, and the data flow version will be used in the complexity analysis. Although it is not proven, it is not difficult to see that the two algorithms are equivalent. The data flow version can be thought of as an implementation of the higher level functional version. The data flow representation is relatively easy to understand, and it neatly expresses any parallelism which can be exploited.

The next three sections describe each of the subroutines. Following that, the data flow and functional versions are presented.

5.5.1 Function *findCGP*

Function *findCGP* finds the first cut in a poset that satisfies a given weak conjunctive global predicate (CGP). An efficient, decentralized, token-based algorithm for detecting weak CGP appears in [16]. 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 [16]. 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 [16], 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.

5.5.2 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.

5.5.3 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.

5.5.4 Data Flow Version

Any RCL formula can be represented with a data flow graph with three types of nodes corresponding to the subroutines mentioned above: $findCGP$, *sup* and *advance*. The data that flows between the nodes are cuts which represent the current position in the poset which is being searched.

Figure 5.9a shows the data flow graph for the formula $g_{10}\langle g_{11} \triangle g_{12}\rangle g_{13}$. The algorithm finds the first cut c_{10} which satisfies g_{10}. Then, using c_{10} as a starting point, the algorithm looks for the first cut c_{11} which satisfies g_{11} and the first cut c_{12} which satisfies and g_{12}. The search for c_{11} and c_{12} can occur in parallel. Once c_{11} and c_{12} are determined, they are passed to the *sup* node which passes its result to a node which looks for g_{13}. Figure 5.9b shows an example which uses the $g\langle\langle f\rangle\rangle g$ constructions. It is similar to figure 5.9a except that it has an *advance* node in order to advance the output of the *sup* node before looking for the next CGP. Figure 5.9c shows a more complicated example which uses the previous examples as subformulas.

There are two levels of parallelism in the data flow version. One level is shown by the data flow graph itself. Any two nodes which are not common to any one path through the graph can be executed in parallel. For example, nodes g_7, g_8, and g_9 can execute in parallel, as can nodes g_2 and g_{13}. The other level

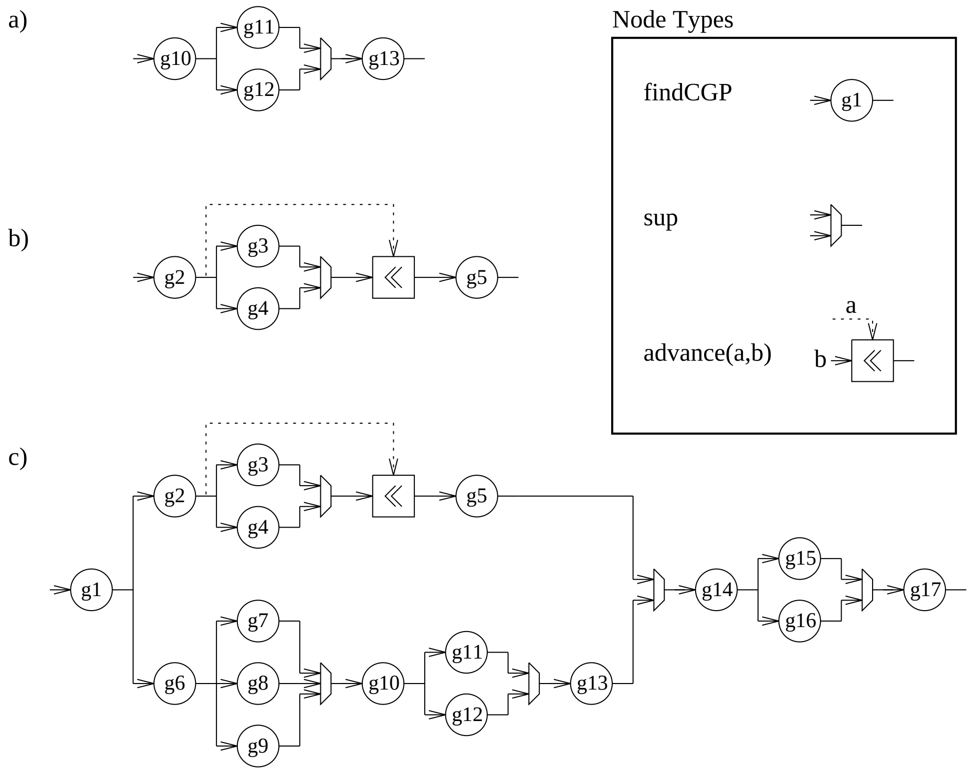

Figure 5.9: Some RCL formulas and their data flow graphs. Figure (a) shows the graph for $g_{10}\langle g_{11} \triangle g_{12}\rangle g_{13}$. Figure (b) shows the graph for $g_2\langle\!\langle g_3 \triangle g_4\rangle\!\rangle g_5$. Figure (c) shows the graph for $g_1 \ \langle \ g_2\langle\!\langle g_3 \triangle g_4\rangle\!\rangle g_5 \ \triangle \ g_6\langle g_7 \triangle g_8 \triangle g_9\rangle g_{10}\langle g_{11} \triangle g_{12}\rangle g_{13} \ \rangle \ \ g_{14} \ \ \langle g_{15} \triangle g_{16}\rangle \ \ g_{17}$

of parallelism is hidden inside the $findCG$P nodes. Recall that $findCGP$ is a procedural interface to a decentralized weak CGP detection algorithm. These coexisting levels of parallelism indicate that multiple invocations of $findCGP$ can be executing concurrently.

At the lowest level the algorithm appears to be fairly complex. For example, imagine all the token based weak CGP detection algorithms being spawned, executing concurrently, and the results being assembled (the *sup* nodes). However, this conceptual complexity is avoided by encapsulating the token inside $findCGP$ and by the use of the data flow abstraction.

This data flow version of the algorithm will be used in section 5.6 to determine the complexity of RCL detection. In the next section we present the functional version, which will be used to prove the algorithm.

5.5.5 Functional Version

The functional version of the algorithm parses a formula and calls subroutines in an order which is consistent with the data flow version.

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 $\top$. Figure 5.10 gives four definitions of $\Psi(a, f)$, one for each syntactic form of f.

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 figure 5.10. Only the subroutines $findCGP$ and $advance$ need access to the poset structure.

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

$a_1 = fc(a, g);$
$a_2 = fc(a_1, f);$
$a_3 = fc(a_2, S);$
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 $sup(fc(P, f_1), fc(P, f_2))$;

Figure 5.10: High level view of algorithm for detecting RCL formulas. The parameter a and the variables a_i are cuts.

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 monotonicity we show 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.

5.6 Algorithm Complexity

The complexity, which depends on the RCL formula upon which the algorithm is being run, is determined as a function of the following parameters:

B = The maximum branching of the data flow graph.

N = The number of processes in the system.

M = The number of messages generated by the application.

The complexity of detecting an RCL formula can be determined by inspecting its data flow graph. Every RCL data flow graph has an attribute which we call "maximum branching". It is an integer value grater or equal to 1 and is determined by counting the maximum number of nodes (or paths) which can execute concurrently. For example, the examples in the graphs shown in figure 5.9 have maximum branching of 2 and 5.

Along any given path in the flow graph, any single poset element (i.e., local state) is processed by only one node. This is evident from the following facts which apply to each node type:

- Each node only processes elements which occur after the first input cut (i.e., infimum of all its input cuts) and before its output cut.
- The output cut is greater or equal to all of its input cuts.

In the worst case, the most expensive node on a path will consume all the elements and the remaining nodes will have almost no work to do.

The cost of the $findCGP$ and $advance$ are $O(NM)$ and $O(N^2)$ respectively. The cost of sup depends on the number of cuts in its input. This value is at most equal to the maximum branching of the graph. Thus the cost of sup is $O(NB)$. Thus the worst case complexity of any single path is $O(max(NB, NN, NM) = O(N \cdot max(B, N, M))$.

Since there are B paths in the graph, the complexity of the entire algorithm is $O(B \cdot N \cdot max(B, N, M))$.

5.7 RCL Properties and Algorithm Proof

5.7.1 Monotonicity

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 5.1 proves that RCL is monotonic.

Lemma 5.1 *Monotonicity:* $P \subseteq R \wedge closed(R) \wedge P \models f \Rightarrow R \models f$

Proof: The proof is by structural induction. For convenience, define the following:

$$M(f) \triangleq P \subseteq R \wedge closed(R) \wedge P \models f \Rightarrow R \models f$$

Using structural induction, the obligation is to prove each of the following four parts. Part 1 is the base case.

Part 1: Prove: $P \subseteq R \wedge closed(R) \wedge P \models g \;\Rightarrow\; R \models g$

$P \subseteq R \wedge closed(R) \wedge P \models g$

$\Rightarrow$ { defn of antichains }

$\overline{P} \subseteq \overline{R} \wedge P \models g \wedge closed(R)$

$\Rightarrow$ { semantics }

$\overline{P} \subseteq \overline{R} \wedge g \cap \overline{P} \neq \emptyset \wedge closed(R)$

$\Rightarrow$ { set theory }

$g \cap \overline{R} \neq \emptyset \wedge closed(R)$

$\Rightarrow$ { semantics }

$R \models g$

Part 2: Given: $M(f_1)$ and $M(f_2)$.

Prove: $P \subseteq R \wedge closed(R) \wedge P \models (f_1 \triangle f_2) \;\Rightarrow\; R \models (f_1 \triangle f_2)$

$P \subseteq R \wedge closed(R) \wedge P \models (f_1 \triangle f_2)$

$\Rightarrow$ { semantics }

$P \subseteq R \wedge closed(R) \wedge P \models f_1 \wedge P \models f_2$

$\Rightarrow$ { $M(f_1)$ and $M(f_2)$ }

$R \models f_1 \wedge R \models f_2$

$\Rightarrow$ { semantics }

$R \models (f_1 \triangle f_2)$

Part 3: Given: $M(g)$ and $M(f)$ and $M(S)$

Prove: $P \subseteq R \wedge closed(R) \wedge P \models g\langle f\rangle S \;\Rightarrow\; R \models g\langle f\rangle S$

$P \subseteq R \wedge closed(R) \wedge P \models g\langle f\rangle S$

$\Rightarrow$ { semantics of $g\langle f\rangle S$ imply that cuts a and b exist }

$P \subseteq R \wedge closed(R) \wedge a, b \in \overline{P} \wedge a \prec b \wedge a \models g \wedge (a..b) \models f \wedge [b..P] \models S$

$\Rightarrow$ { Since $P \subseteq R$, we know $a, b \in \overline{R}$ and $[b..P] \subseteq [b..R]$.
And by $M(S)$, we know $[b..R] \models S$. }

$a, b \in \overline{R} \wedge a \prec b \wedge a \models g \wedge (a..b) \models f \wedge [b..R] \models S$

$\Rightarrow$ { semantics }

$R \models g\langle f\rangle S$

Part 4: Given: $M(g)$ and $M(f)$ and $M(S)$

Prove: $P \subseteq R \wedge closed(R) \wedge P \models g\langle\langle f \rangle\rangle S \;\Rightarrow\; R \models g\langle\langle f \rangle\rangle S$

The proof is similar to Part 3. ∎

5.7.2 Lattice

Given any poset P and formula f, the cutset of (P, f) forms a lattice. The proof is by structural induction and is divided into four parts, one for each syntactic structure. The base case relies on the assumption that cuts in g form a lattice (recall that g is a set of antichains). 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.

Parts 1 and 2 of the proof are fairly simple, part 3 is a little complicated, and part 4 is virtually identical to part 3. Figure 5.11 demonstrates graphically what is going on in part 3.

Lemma 5.2 *Lattice:* $a, b \in \Psi(P, f) \;\Rightarrow\; inf(a, b) \in \Psi(P, f) \wedge sup(a, b) \in \Psi(P, f)$

Proof: Due to the definition of $\Psi(P, f)$, it suffices to show that if $a, b \in \Psi(P, f)$, then the following claims are true:

1. $inf(a, b) \in \overline{P}$
2. $[P .. inf(a, b)] \models f$
3. $sup(a, b) \in \overline{P}$
4. $[P .. sup(a, b)] \models f$

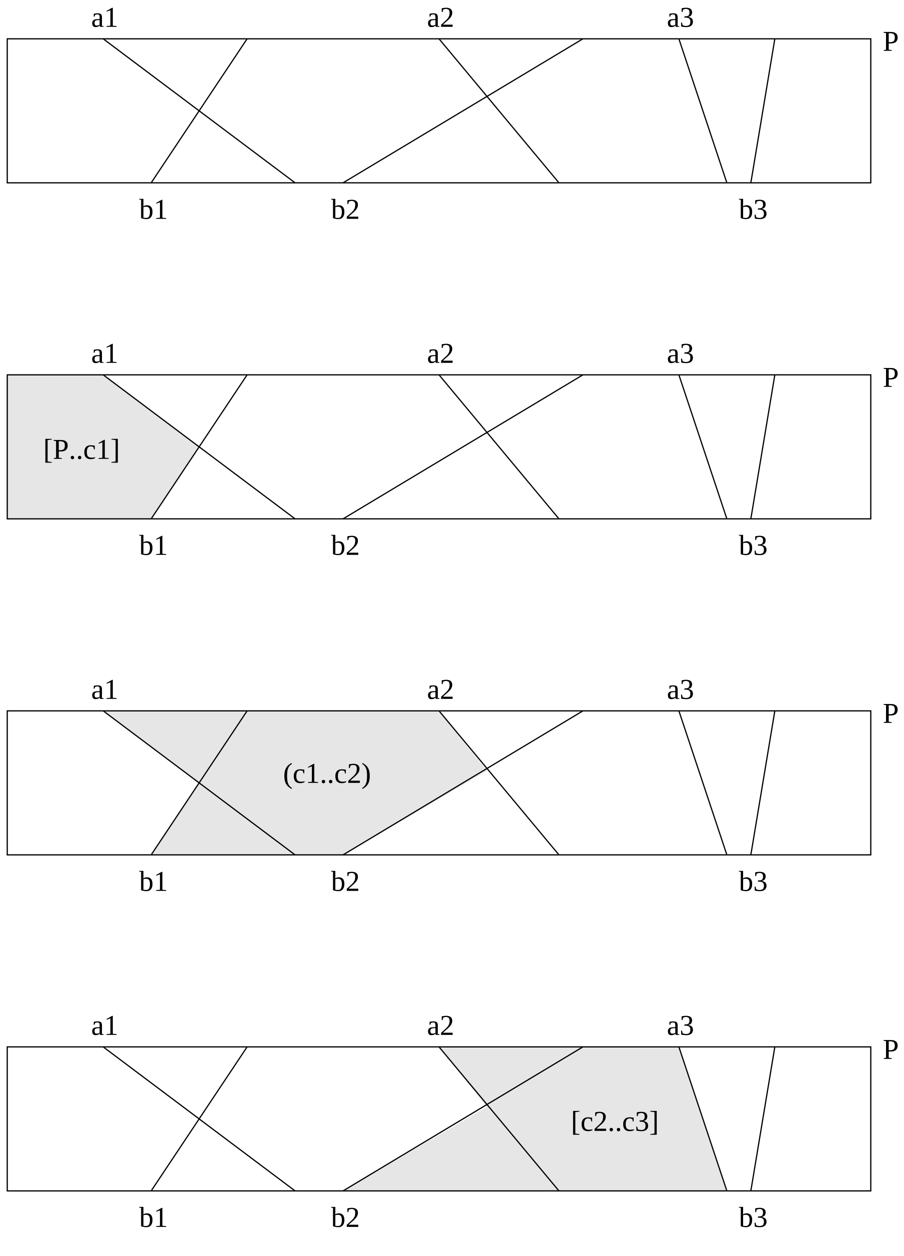

Figure 5.11: Part 3 of Lemma 5.2.

The set of cuts in any poset form a lattice, therefore claims 1 and 3 are true. Claim 4 follows from monotonicity. Claim 2 is proven below by induction on the structure of the logic. Part 1 is the base case. For convenience, define $L(P, f)$ to mean that the lemma holds specifically for P and f.

Part 1: Prove: $a, b \in \Psi(P, g) \;\Rightarrow\; inf(a, b) \in \Psi(P, g)$

By assumption of the RCL logic, the set of cuts which satisfy g form a lattice. Thus if $a, b \in \Psi(P, g)$, then $inf(a, b) \in \Psi(P, g)$. (See [21] for proof of this when g represents a CGP.)

Part 2: Given: $L(P, f_1) \wedge L(P, f_2)$

Prove: $a, b \in \Psi(P, f_1 \triangle f_2) \;\Rightarrow\; inf(a, b) \in \Psi(P, f_1 \triangle f_2)$

$a, b \in \Psi(P, f_1 \triangle f_2)$

$\Rightarrow$ { defn of $\Psi(\ )$, semantics of $\triangle$ }

$a, b \in \Psi(P, f_1) \wedge a, b \in \Psi(P, f_2)$

$\Rightarrow$ { induction: $L(P, f_1) \wedge L(P, f_2)$ }

$inf(a, b) \in \Psi(P, f_1) \wedge inf(a, b) \in \Psi(P, f_2)$

$\Rightarrow$ { defn of $\Psi(\ )$, semantics of $\triangle$ }

$inf(a, b) \in \Psi(P, f_1 \triangle f_2)$

Part 3: (See figure 5.11 for a graphical representation of this part.)

Given: $L(P, g) \wedge L(P, f) \wedge L(P, S)$

Prove: $a_3, b_3 \in \Psi(P, g\langle f\rangle S) \;\Rightarrow\; inf(a_3, b_3) \in \Psi(P, g\langle f\rangle S)$

$a_3, b_3 \in \Psi(P, g\langle f\rangle S)$

{ By definition of $\Psi(\ )$: }

$[P..a_3] \models g\langle f\rangle S \wedge [P..b_3] \models g\langle f\rangle S \wedge a_3, b_3 \in \overline{P}$

{ From the semantics, cuts a_1, a_2, b_1, b_2 exist such that: }

$a_1, a_2 \in \overline{[P..a_3]} \wedge a_1 \prec a_2 \wedge a_1 \models g \wedge (a_1..a_2) \models f \wedge [a_2..a_3] \models S$

$b_1, b_2 \in \overline{[P..b_3]} \wedge b_1 \prec b_2 \wedge b_1 \models g \wedge (b_1..b_2) \models f \wedge [b_2..b_3] \models S$

{ Let $c_1 = inf(a_1, b_1)$. Then by $L(P, g)$: }

$c_1 \models g \wedge c_1 \in \overline{P}$

{ Since $c_1 = inf(a_1, b_1)$: }

$(a_1..a_2) \subseteq (c_1..a_2) \wedge (b_1..b_2) \subseteq (c_1..b_2)$

{ Recall that $(a_1..a_2) \models f$ and $(b_1..b_2) \models f$. Thus by monotonicity: }

$(c_1..a_2) \models f \wedge (c_1..b_2) \models f$

{ Let $c_2 = inf(a_2, b_2)$. Then by $L((c_1..P], f)$: }

$(c_1..c_2) \models f \wedge c_2 \in \overline{P}$

{ By definition of c_1 and c_2: }

$c_1 \prec c_2$

{ Since $c_2 = inf(a_2, b_2)$: }

$[a_2..a_3] \subseteq [c_2..a_3] \wedge [b_2..b_3] \subseteq [c_2..b_3]$

{ Recall $[a_2..a_3] \models S$ and $[b_2..b_3] \models S$ Thus by monotonicity: }

$[c_2..a_3] \models S \wedge [c_2..b_3] \models S$

{ Let $c_3 = inf(a_3, b_3)$. Then by $L([c_2..P], S)$: }

$[c_2..c_3] \models S \wedge c_3 \in \overline{P}$

{ If you look above you will see the following statements: }

$c_1, c_2 \in \overline{P} \wedge c_1 \prec c_2 \wedge c_1 \models g \wedge (c_1..c_2) \models f \wedge [c_2..c_3] \models S$

{ Thus by semantics: }

$[P..c_3] \models g\langle f\rangle S$

{ This, combined with $c_3 \in \overline{P}$, indicates: }

$c_3 \in \Psi(P, g\langle f\rangle S)$

{ And, by definition of c_3: }

$inf(a_3, b_3) \in \Psi(P, g\langle f\rangle S)$

Part 4: Given: $L(P, g) \wedge L(P, f) \wedge L(P, S)$

Prove: $a, b \in \Psi(P, g\langle\langle f\rangle\rangle S) \;\Rightarrow\; inf(a, b) \in \Psi(P, g\langle\langle f\rangle\rangle S)$

Proof is similar to Part 3.

∎

5.7.3 The Final Push

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 $g\langle\!\langle f\rangle\!\rangle S$.

Theorem 5.1 *$fc(c, g) = inf\ \Psi(P, g)$, 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 [21]. ■

Theorem 5.2 *$fc(c, f_1 \triangle f_2) = inf\ \Psi(P, f_1 \triangle f_2)$, 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 5.12 shows the structure of the posets mentioned in the proof.

Lemma 5.3 *$Z(a_1, a_2, b_1, b_2, f, P)$, 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:

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

$a_1 \preceq b_1$

$[b_1..b_2] \models f$

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

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 $a_2 \preceq b_2$. ■

Theorem 5.3 *$fc(c, g\langle f\rangle S) = inf\ \Psi(P, g\langle f\rangle S)$, where $c = inf\ \overline{P}$*

Proof: From the algorithm, it is clear that $fc(c, g\langle f\rangle S) = a_3$, where:
$a_1 = fc(c, g)$
$a_2 = fc(a_1, f)$

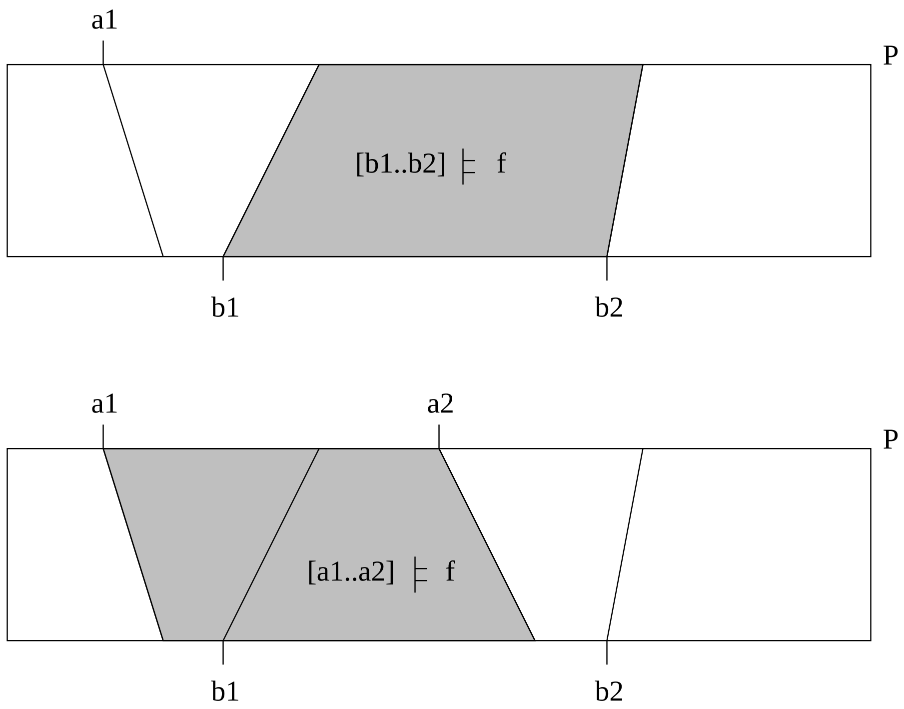

Figure 5.12: Structure of posets mentioned in lemma 5.3.

$a_3 = fc(a_2, S)$

By structural induction, we also know that:

$a_1 = \mathit{inf}\ \Psi([c..P], g)$
$a_2 = \mathit{inf}\ \Psi([a_1..P], f)$
$a_3 = \mathit{inf}\ \Psi([a_2..P], S)$

We must show that $a_3 = \mathit{inf}\ \Psi(P, g\langle f\rangle S)$. 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 $a_3 = \mathit{inf}\ \Psi(P, g\langle f\rangle S)$, 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: $\}$

$a_1 = \mathit{inf}\ \Psi(P, g) \wedge [\bot..b_1] \models g \wedge \bot \preceq \bot$

$\{$ The consequents are: $\}$

$a_1 \preceq b_1 \wedge [\bot..a_1] \models g$

$\left\{\begin{array}{l}\text{Now we do the same thing with } Z(a_1, a_2, b_1, b_2, f, [a_1..P]).\\ \text{The preconditions are:}\end{array}\right\}$

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

$\{$ The consequents are: $\}$

$a_2 \preceq b_2 \wedge (a_1..a_2) \models f$

$\left\{\begin{array}{l}\text{One more time with } Z(a_2, a_3, b_2, x, S, [a_2..P]).\\ \text{The preconditions are:}\end{array}\right\}$

$a_2 \preceq b_2 \wedge a_3 = \mathit{inf}\ \Psi([a_2..P], S) \wedge [b_2..x] \models S$

$\{$ 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: $\}$

$a_3 = \top$

∎

Theorem 5.4 $fc(c, g\langle\langle f \rangle\rangle S) = inf\ \Psi(P, g\langle\langle f \rangle\rangle S)$, *where* $c = inf\ \overline{P}$

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

Chapter 6

DAG Expressions

This chapter introduces a simple but powerful logic that can express general properties on control flows of a distributed program. A control flow is essentially a chain of local states. The states in the chain can reside at different processes.

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 chapter also presents a simple decentralized algorithm to detect the formulas expressed with this logic. The detection algorithm is superimposed on the distributed computation and 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 chapter begins by defining the syntax and semantics of the DAG expression logic. Example use of this logic are presented, and finally, the algorithm for detection of DAG expressions is given.

6.1 DAG Expression 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 6.2 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.)

6.1.1 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.

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

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

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

6.1.2 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 $s \in P$ if and only if formula f_j is true in s. This can be stated formally as follows:

$$s \models x_j \quad \triangleq \quad s \models f_j$$

$\Diamond_l$ 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*.

$$s \models \Diamond_l\, x_j \quad \triangleq \quad (\exists s' : s' \prec_{im} s : s' \models x_j)$$

$$s \models \Diamond_m \, x_j \quad \triangleq \quad (\exists s' : s' \rightsquigarrow s : s' \models x_j)$$

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

$$s \models p \quad \triangleq \quad p \in \lambda(s)$$

The remaining semantic definitions are straightforward.

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

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

6.1.3 Predefined predicates

It is useful to define predicates on local states whose truth values depend on the position of the local state s in P. These built-in predicates can be used to specify properties which take into account the structure of P. 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.

$$s \models \mathit{initial} \quad \triangleq \quad \neg(\exists s' :: s' \prec_{im} s)$$

$$s \models \mathit{receive} \quad \stackrel{\triangle}{=} \quad (\exists s' :: s' \leadsto s)$$

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:

$$s \models \mathit{external} \stackrel{\triangle}{=} (s \models \mathit{send}) \vee (s \models \mathit{receive})$$

6.2 Examples

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

6.2.1 Invariance and Existence in the Local Past

If x_1 is defined as:

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

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:

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

6.2.2 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:

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

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:

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

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

6.2.3 Interval Abstraction

It is sometimes useful to consider a distributed execution modeled as a poset of intervals [55] 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. Consider the following pattern specification:

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

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.

6.2.4 Regular Expressions

Please note that 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".

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. $(\forall k : 0 \leq k < n : \sigma^k \prec_{im} \sigma^{k+1} \vee \sigma^k \rightsquigarrow \sigma^{k+1})$
3. $\sigma^n = s$
4. $(\forall k : 0 \leq k \leq n : \alpha^k \in \lambda(\sigma^k))$

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 [44], atomic sequences of predicates [31] and regular patterns [14] are special cases of sequential properties that can be described by the DAG 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 DAG 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.

Let $\diamond\ x_k$ be a short form for $(\diamond_l\ x_k \vee \diamond_m\ x_k)$[1]. 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

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

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 6.1) implementing the regular expression $a + cb^*c$ (q_3 is the only final state).

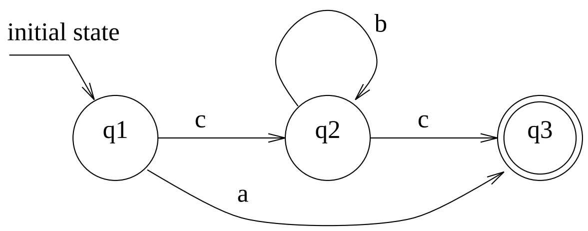

Figure 6.1: State machine implementing $a + cb^*c$

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:

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

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 $a + cb^*c$.

6.2.5 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. Consider an algorithm which uses multiple phases of this scatter and gather operation on a matrix D, and let D_0 denote the matrix D in phase 0.

D_0 is initially centralized at one process. It is then distributed among the processes so that submatrix D_0^i is located at process P_i. After each process performs some operation on its submatrix, the pieces are collected to form the centralized matrix D_1 which will be used to start the next phase.

We will use logic variables A_0, A_1, and A_0^i to specify this structure. A_0 is true if the predicate "D_0 is centralized" was true in some state in the global past.

$$A_0 \;\; := \;\; (D_0 \text{ is centralized}) \vee \Diamond\, A_0$$

A_0^i is true if the predicate "submatrix D_0^i is located at P_i and A_0 is true" was true in some state in the global past.

$$A_0^i \;\; := \;\; ((D_0^i \text{ is at } P_i) \wedge A_0) \vee \Diamond\, A_0^i$$

And finally, A_1 is true when the predicate "D_1 is centralized and A_0^i is true for all i" was true in some state in the global past.

$$A_1 \;\; := \;\; ((D_1 \text{ is centralized}) \wedge \forall i : A_0^i) \vee \Diamond\, A_1$$

6.3 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.

6.3.1 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 6.2.4:

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

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) **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.

6.3.2 Correctness Proof

The proof consists in showing the following equivalence, where s is any state in P. (Recall that the term "state" always refers to a local state).

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

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 P. 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:

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

■

Chapter 7

Program Verification

This chapter presents a method for specifying and proving distributed programs. The method is based on the partial order of local states generated during execution (causal domain) and avoids the notions of time and simultaneity (time domain). The causal domain refers to a world based on Lamport's causally precedes relation, whereas the time domain refers to a world based on physical time. The importance of the causal domain was first emphasized by Lamport in [34] where he defined the causally precedes relation and showed that execution of a distributed program can be modeled by a set of local states partially ordered by this relation.

Program properties are often more simply stated in the causal domain than in the time domain, which makes them easier to understand and manipulate. This simplicity and elegance is gained because the causal domain inherently accounts for different execution schedules. For example, consider the concept of "global state", which has different meanings in each domain. In the causal domain, a global state is a set of local states which are all concurrent with each other. In the time domain, a global state is a set of local states which occur simultaneously. A global state in the time domain is also a global state in the causal domain. However, the converse is not true. This results from the fact that the causal definition takes into account variable process execution speeds.

Suppose we are debugging a distributed program and we want to halt the program if mutual exclusion is violated by the current run or another otherwise identical run with a different execution schedule. In the causal domain, this breakpoint would be specified as follows: "there exists a global state in which two or more processes have access to the critical section". The time domain equivalent is more convoluted: "there exists a set of local states in which two or more processes have access to the critical section and which could coexist in this run or another otherwise identical run with different process execution speeds."

Consider another example which highlights the hidden dangers of the time domain. The statement "there exists a global state in which more than two processes have access to the critical section" cannot be verified in the time domain except in special cases. This is because it is impossible to determine if a given global state occurs in the time domain without access to perfectly synchronized local clocks, which do not exist in the real world. In the causal domain however, it is possible to determine if a given global state occurs [7, 19, 20, 55].

Despite the advantages of the causal domain, it is common in the research community to use the time domain to specify and prove properties of distributed programs [35, 46]. A classical example is distributed mutual exclusion in which the absence of violation of mutual exclusion is specified as $\Box\neg(CS_1 \wedge CS_2)$. That is, there is no time at which CS_1 and CS_2 are both true.

In our method, programs are specified by documenting the relationship between states which are adjacent to each other in the partial order. Properties are stated using mathematical relations which are derived from the partial order (e.g., the causally precedes relation).

This chapter also introduces a proof technique which uses induction on the complement of the causally precedes relation and is useful in the causal domain. Examples of using the causal domain to specify and prove distributed programs are given for several distributed algorithms. The method presented in this chapter for specifying verifying a program in the causal domain is outlined below:

1. The program is specified by documenting how the process state is altered between adjacent states. Any poset that satisfies the rules is a valid run of the program.
2. The desired properties of the program are specified using $\rightarrow$, $\not\rightarrow$ and $\|$. It is important to note that the concept of global time and therefore the global state is completely avoided in this approach. Any variable, x, has meaning only in the context of a local state (i.e. $s.x$).
3. The desired properties are proven using properties derived from the program text. This is done using induction on $\rightarrow$ and $\not\rightarrow$.

The first task undertaken in this chapter is to develop a theoretical basis for the proof technique of induction on $\not\rightarrow$. Following that, examples are given which demonstrate the use of the causal domain to specify and prove distributed programs. The examples used are: the vector clock algorithm, the direct dependency clock algorithm, and the mutual exclusion algorithm.

7.1 Model

For our purposes, some restrictions on the poset of local states are required. They restrict how the program is represented, but not the programs which can be modeled (dummy states can be inserted where necessary to satisfy the restrictions).

Initial and final states of a process can be identified by the structure of the poset as shown by the following predicates:

$$\begin{aligned} Init(s) &\triangleq \neg(\exists u :: u \prec_{im} s) \\ Final(s) &\triangleq \neg(\exists u :: s \prec_{im} u) \end{aligned}$$

The restrictions on the poset are listed below. The first restriction ensures that no state causally precedes an initial state. The second restriction ensures that a final state does not causally precede any state. The third restriction means that at most one message is sent or received in between consecutive states in a process.

1. $Init(s) \Rightarrow \neg(\exists u :: u \leadsto s)$
2. $Final(s) \Rightarrow \neg(\exists u :: s \leadsto u)$
3. $s \prec_{im} t \Rightarrow \ \mid \{u \mid s \leadsto u \vee u \leadsto t\} \mid \leq 1$

For every pair of consecutive states, $s \prec_{im} t$, exactly one event occurs between s and t. There are three types of events denoted by int, snd, and rcv. Which event occurs between two states can be determined from the deposet structure as shown in the following definitions.

$$
\begin{aligned}
(s, snd(u), t) &\triangleq s \prec_{im} t \wedge s \leadsto u \\
(s, rcv(u), t) &\triangleq s \prec_{im} t \wedge u \leadsto t \\
(s, snd, t) &\triangleq (\exists u :: (s, snd(u), t)) \\
(s, rcv, t) &\triangleq (\exists u :: (s, rcv(u), t)) \\
(s, int, t) &\triangleq s \prec_{im} t \wedge \neg(s, snd, t) \wedge \neg(s, rcv, t)
\end{aligned}
$$

The above relations model the events that occur between consecutive local states: (s, snd, t) models a message send, (s, rcv, t) models a message receive, and (s, int, t) models an internal event.

7.2 Proofs by Induction on $\xrightarrow{k}$ and $\overset{k}{\not\rightarrow}$

The causally-precedes relation, $\rightarrow$, and its complement, $\not\rightarrow$, is quite useful in designing, analyzing and debugging asynchronous distributed programs. In this section, we define variants of these relations so that properties based on them can be proven by induction. While the technique for using induction on $\rightarrow$ is almost obvious, it is not obvious how to perform induction on $\not\rightarrow$. In this section we formalize both techniques.

A chain in $(S, \rightarrow)$ is a sequence of states $c_0, c_1, \ldots c_n$ such that $c_i \prec_{im} c_{i+1}$ or $c_i \leadsto c_{i+1}$. For any chain $c = c_0, c_1, \ldots c_n$, we define $first(c) = c_0$, $last(c) = c_n$, and $length(c) = n$. The inductive proof technique can be used on $(S, \rightarrow)$ because every decreasing chain in $(S, \rightarrow)$ is finite. That is, for any state t and chain c such that $last(c) = t$, c has finite length. For any pair of states s, t we define the maximum length function, $ml(s,t)$ as follows:

$$ml(s,t) \quad = \quad \begin{cases} (max\, c : first(c) = s \wedge last(c) = t : length(c)) & \text{if } s \rightarrow t \vee s = t \\ -1 & \text{otherwise} \end{cases}$$

The max expression is well defined since there are a finite number of states which causally precede t. This implies that ml has a well defined value for every pair of states s and t.

If $s \rightarrow t$, then $ml(s,t)$ equals the length of the longest chain between s and t. If $s = t$, then $ml(s,t) = 0$. We use $ml(Init, t)$ to denote $(max\, u : Init(u) : ml(u,t))$. Thus $ml(Init, t)$ is length of the longest chain from some initial state to t. The following statement is true by definition of ml and is used in some of our proofs. We refer to it as the *chain lemma.*

$$ml(s,t) > 0 \quad \Leftrightarrow \quad (\exists u :: ml(s,u) = ml(s,t) - 1 \wedge ml(u,t) = 1)$$

The relations which enable straightforward induction on $\rightarrow$ and $\not\rightarrow$ are $\rightarrow_k$ and $\not\rightarrow_k$ and are defined as follows:

$$\begin{array}{lccc} \text{for } k > 0: & s \rightarrow_k t & \triangleq & ml(s,t) = k \\ \text{for } k \geq 0: & s \not\rightarrow_k t & \triangleq & s \not\rightarrow t \wedge ml(Init, t) = k \end{array}$$

Thus $s \rightarrow_k t$ if and only if $s \rightarrow t$ and the longest chain from s to t has length k, and $s \not\rightarrow_k t$ if and only if $s \not\rightarrow t$ and the longest chain from some initial state to t has length k. Figure 7.1 shows examples of these relations.

Lemmas 7.1, 7.2 and 7.3 justify using induction on $\rightarrow$. For example, suppose we wish to prove the claim $s \rightarrow t \Rightarrow P(s,t)$ where $P(s,t)$ is some predicate on the local variables in s and t. From lemma 7.1 we note that it is sufficient to prove $s \rightarrow_k t \Rightarrow P(s,t)$ for all $k > 0$. The proof can proceed by induction on k, using lemma 7.2 for the base case and lemma 7.3 for the induction case. The base case, $s \rightarrow_1 t$, implies that either states s and t are consecutive states in a process, or a message was sent in s and received in t. Generally the base case can be easily proven from the program text since it involves only one state transition or one message.

Lemma 7.1 $s \rightarrow t \Leftrightarrow (\exists k : k > 0 : s \rightarrow_k t)$

Proof:

$$
\begin{array}{ll}
 & s \rightarrow t \\
\Leftrightarrow & \{ \text{ by defn of a chain, and since } \rightarrow \text{ is the transitive closure of } \leadsto \cup \prec_{im} \} \\
 & (\exists c :: first(c) = s \wedge last(c) = t) \\
\Leftrightarrow & \{ \text{ defn of } ml \ \} \\
 & (\exists k : k > 0 : ml(s,t) = k) \\
\Leftrightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & (\exists k : k > 0 : s \rightarrow_k t)
\end{array}
$$

∎

Lemma 7.2 $s \rightarrow_1 t \Rightarrow s \prec_{im} t \vee s \leadsto t$

Proof:

$$
\begin{array}{ll}
 & s \rightarrow_1 t \\
\Rightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & ml(s,t) = 1 \\
\Rightarrow & \{ \text{ defn of } ml \ \} \\
 & (\exists c :: first(c) = s \wedge last(c) = t \wedge len(c) = 1) \\
\Rightarrow & \{ \text{ defn of a chain } \} \\
 & s \prec_{im} t \vee s \leadsto t
\end{array}
$$

∎

The converse of lemma 7.2 does not hold in the case where messages are received out of order. For example, in figure 7.1, $s1 \leadsto t4$ holds but not $s1 \to_1 t4$ does not. The reason is that there is a chain of length four from $s1$ to $t4$, thus $s1 \to_4 t4$.

Lemma 7.3 $s \to_k t \wedge (k > 1) \Rightarrow (\exists u :: s \to_{k-1} u \wedge u \to_1 t)$

Proof:

$$
\begin{array}{ll}
 & s \to_k t \wedge (k > 1) \\
\Rightarrow & \{ \text{ defn of } \to_k \} \\
 & ml(s,t) = k \wedge k > 1 \\
\Rightarrow & \{ \text{ chain lemma } \} \\
 & (\exists u :: ml(s,u) = k - 1 \wedge ml(u,t) = 1) \\
\Rightarrow & \{ \text{ defn of } \to_k \} \\
 & (\exists u :: s \to_{k-1} u \wedge u \to_1 t)
\end{array}
$$

■

Lemmas 7.4, 7.5 and 7.6 are used in inductive proofs for properties stated with the $\not\to_k$ relation. The method is similar to the one described above for $\to_k$. Suppose $s \not\to t \Rightarrow R(s,t)$. Lemma 7.4 tells us that it is sufficient to prove $s \not\to_k t \Rightarrow R(s,t)$ for all $k \geq 0$, which can be proven by induction on k. To prove the base case, $s \not\to_0 t$, we need to show that $R(s,t)$ holds when t is an initial state. The inductive case $(k > 0)$ uses lemma 7.6.

Lemma 7.4 $s \not\to t \Leftrightarrow (\exists k : k \geq 0 : s \not\to_k t)$

Proof:

$$\begin{array}{ll} & s \not\rightarrow t \\ \Leftrightarrow & \{ \text{ by defn of } ml(Init, t) \ \} \\ & s \not\rightarrow t \wedge ml(Init, t) \geq 0 \\ \Leftrightarrow & \{ \text{ defn of } \rightarrow_k \ \} \\ & (\exists k : k \geq 0 : s \not\rightarrow_k t) \end{array}$$

■

Lemma 7.5 $s \not\rightarrow_0 t \Leftrightarrow Init(t)$

Proof:

$$\begin{array}{ll} & s \not\rightarrow_0 t \\ \Leftrightarrow & \{ \text{ defn of } \not\rightarrow_0 \ \} \\ & ml(Init, t) = 0 \wedge s \not\rightarrow t \\ \Leftrightarrow & \{ \text{ defn of } ml(Init, t) \ \} \\ & \neg(\exists u :: u \rightarrow t) \wedge s \not\rightarrow t \\ \Leftrightarrow & \{ \text{ left conjunct implies right conjunct } \} \\ & \neg(\exists u :: u \rightarrow t) \\ \Leftrightarrow & \{ \text{ defn of } Init(t) \ \} \\ & Init(t) \end{array}$$

■

It is not immediately obvious that lemma 7.6 will always apply in the inductive case, but consider the following. The assumption in the inductive case is $k > 0 \wedge s \not\rightarrow_k t$. This implies that t is not an initial state (see lemma 7.5), which in turn implies that there exists some state u such that $u \rightarrow t$. Thus the left hand side of lemma 7.6 will always be true if $k > 0$.

Lemma 7.6 $k > 0 \wedge s \not\rightarrow_k t \wedge u \rightarrow t \Rightarrow (\exists j : 0 \leq j < k : s \not\rightarrow_j u)$

Proof:

$$
\begin{array}{ll}
 & k > 0 \wedge s \not\to_k t \wedge u \to t \\
\Rightarrow & \{ \text{ otherwise } s \to t \ \} \\
 & k > 0 \wedge s \not\to_k t \wedge u \to t \wedge s \not\to u \\
\Rightarrow & \{ \text{ defn of } \not\to_k \ \} \\
 & k > 0 \wedge s \not\to u \wedge ml(Init, t) = k \wedge u \to t \\
\Rightarrow & \{ \text{ otherwise } ml(Init, t) > k \ \} \\
 & k > 0 \wedge s \not\to u \wedge ml(Init, u) < k \\
\Rightarrow & \{ \text{ defn of } \not\to_j \ \} \\
 & (\exists j : 0 \leq j < k : s \not\to_j u)
\end{array}
$$

■

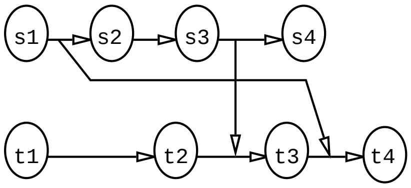

Figure 7.1: Example relations: $s3 \to_1 t3$, $t1 \to_2 t3$, $s1 \to_4 t4$, $s2 \not\to_0 t1$, $s4 \not\to_3 t3$.

7.3 Vector Clock

In this section we present a vector clock algorithm, state the desired properties of the algorithm and prove that it satisfies the properties. Vector clocks are widely used in applications such as debugging, concurrency control in databases, recovery in fault tolerant systems, and ordered broadcast algorithms. We use a variant of the traditional algorithm [13, 41] that uses less state space and is more difficult to prove as explained later.

Vectors of integers can be partially ordered by an appropriately defined comparison relation $<$. They are useful for characterizing the relationship between local states since local states are partially ordered by $\to$. The comparison relation is defined for vectors u and v of length N as follows:

$$u < v \triangleq (\forall k : 1 \leq k \leq N : u[k] \leq v[k]) \wedge (\exists j : 1 \leq j \leq N : u[j] < v[j])$$

The traditional vector clock algorithm assigns a vector $s.v$ to every local state s such that $s.v < t.v$ if and only if $s \rightarrow t$. We use a slightly different version in which this condition holds when s and t are on different processes. We use this version because it is harder to prove (as discussed later) and also because it is practical. It conserves state space since the vector components are incremented less frequently, and in general, one is interested in causal relationships between states on different processes. The version we use maintains the following property:

$$(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$$

7.3.1 Algorithm

Let there be N processes uniquely identified by an integer value between 1 and N inclusive. Recall that for any state s, $s.p$ indicates the identity of the process to which it belongs. It is not required that message communication be ordered or reliable. The algorithm is described by the initial conditions and the actions taken for each event type.

For any initial state s:

$(\forall i : i \neq s.p : s.v[i] = 0) \wedge (s.v[s.p] = 1)$

Rule for a send event (s, snd, t):

$t.v := s.v;$

$t.v[t.p] + +;$

Rule for a receive event $(s, rcv(u), t)$:

for $i := 1$ to N

$t.v[i] := max(s.v[i], u.v[i]);$

Rule for an internal event (s, int, t):

$t.v := s.v;$

The version presented above is harder to prove than the traditional algorithm because of the message receive action. In the traditional algorithm, when a message is received in state s, the local clock, $s.p$, is incremented. This ensures that $(s, rcv(u), t)$ implies $s.v < t.v$ and $u.v < t.v$. The action taken in this version, $t.v := max(s.v, u.v)$, does not imply $s.v < t.v$ nor does it imply $u.v < t.v$. This makes this version more difficult to prove.

We use the following properties of the algorithm in our proof. Their validity is clear from the algorithm text. Our proof is derived strictly from these properties; the algorithm itself is not used. Therefore the proof is valid for any algorithm which satisfies these properties. For example, in the send rule of the algorithm, $t.v[t.p]$ could be increased by any positive amount and our proof would still be valid.

Init rule:	$Init(s) \Rightarrow (\forall i : i \neq s.p : s.v[i] = 0) \wedge (s.v[s.p] = 1)$
Snd rule:	$(s, snd, t) \Rightarrow (\forall i : i \neq t.p : t.v[i] = s.v[i]) \wedge t.v[t.p] > s.v[t.p]$
Rcv rule:	$(s, rcv(u), t) \Rightarrow (\forall i :: t.v[i] = max(s.v[i], u.v[i]))$
Int rule:	$(s, int, t) \Rightarrow t.v = s.v$

7.3.2 Proof

In this section we prove the property stated earlier: $(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$. This is accomplished by proving the following claims:

$$s.p \neq t.p \wedge s \rightarrow t \Rightarrow s.v < t.v \tag{7.1}$$

$$s.p \neq t.p \wedge s.v < t.v \Rightarrow s \rightarrow t \tag{7.2}$$

Lemma 7.7 states that if there is a chain of events from s to t then $s.v \leq t.v$. In the traditional algorithm, proof of the property $s \rightarrow t \Rightarrow s.v < t.v$ (which does not hold here) is essentially the same as this proof. This is because, in the traditional algorithm, local clocks are incremented for every event type. Note also that the proof of lemma 7.7 does not use the initial conditions. Thus the lemma holds independent of the initial values of the vectors.

Lemma 7.7 $s \rightarrow t \Rightarrow s.v \leq t.v$

Proof: It is sufficient to show that for all $k > 0$: $s \rightarrow_k t \Rightarrow s.v \leq t.v$. We use induction on k.

$Base\ (k = 1)$:

$$\begin{array}{ll}
 & s \rightarrow_1 t \\
\Rightarrow & \{ \text{ lemma 7.2 } \} \\
 & s \prec_{im} t \vee s \rightsquigarrow t \\
\Rightarrow & \{ \text{ expand } s \prec_{im} t \text{ and } s \rightsquigarrow t \ \} \\
 & (s, int, t) \vee (s, snd, t) \vee (\exists u :: (s, rcv(u), t)) \vee (\exists u :: (u, rcv(s), t)) \\
\Rightarrow & \{ \text{ Snd, Rcv, and Int rules } \} \\
 & (s.v = t.v) \vee (s.v < t.v) \vee (s.v \leq t.v) \vee (s.v \leq t.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & s.v \leq t.v
\end{array}$$

Induction: $(k > 1)$

$$
\begin{array}{ll}
 & s \rightarrow_k t \wedge (k > 1) \\
\Rightarrow & \{ \text{ lemma 7.3 } \} \\
 & (\exists u :: s \rightarrow_{k-1} u \wedge u \rightarrow_1 t) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
 & (\exists u :: s.v \leq u.v \wedge u.v \leq t.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & s.v \leq t.v
\end{array}
$$

■

Lemma 7.8 states that if two states s and t are on different processes, and s does not causally precede t, then $t.v[s.p] < s.v[s.p]$. Our formal proof of this lemma is nontrivial. This proof is by induction on k in the $\not\rightarrow_k$ relation.

Lemma 7.8 $(\forall s, t : s.p \neq t.p : s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p])$

Proof: It is sufficient to show that for all $k \geq 0$: $s \not\rightarrow_k t \wedge s.p \neq t.p \Rightarrow t.v[s.p] < s.v[s.p]$. We use induction on k.

Base $(k = 0)$:

$$
\begin{array}{ll}
 & s \not\rightarrow_0 t \wedge s.p \neq t.p \\
\Rightarrow & \{ \text{ lemma 7.5 } \} \\
 & Init(t) \wedge s.p \neq t.p \\
\Rightarrow & \{ \text{ let } u \text{ be initial state in } s.p \ \} \\
 & Init(t) \wedge s.p \neq t.p \wedge (\exists u : Init(u) \wedge u.p = s.p : u = s \vee u \rightarrow s) \\
\Rightarrow & \{ \text{ lemma 7.7 } \} \\
 & Init(t) \wedge s.p \neq t.p \wedge (\exists u : Init(u) \wedge u.p = s.p : u.v = s.v \vee u.v \leq s.v) \\
\Rightarrow & \{ \text{ Init rule } \} \\
 & t.v[s.p] = 0 \wedge (\exists u : u.v[s.p] = 1 : u.v = s.v \vee u.v \leq s.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Induction: $(k > 0)$

$$
\begin{array}{ll}
 & s \not\rightarrow_k t \wedge s.p \neq t.p \wedge k > 0 \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec_{im} t,\ u \text{ exists since } \neg Init(t) \ \} \\
 & s \not\rightarrow_k t \wedge s.p \neq t.p \wedge u.p = t.p \wedge u \prec_{im} t \\
\Rightarrow & \{ \text{ lemma 7.6 } \} \\
 & s \not\rightarrow_j u \wedge 0 \leq j < k \wedge u.p \neq s.p \wedge u \prec_{im} t \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & u.v[s.p] < s.v[s.p] \wedge u \prec_{im} t \\
\Rightarrow & \{ \text{ expand } u \prec_{im} t \ \} \\
 & u.v[s.p] < s.v[s.p] \wedge ((u, int, t) \vee (u, snd, t) \vee (u, rcv(w), t))
\end{array}
$$

Consider each disjunct separately:

Case 1: (u, int, t)

$$
\begin{array}{ll}
 & u.v[s.p] < s.v[s.p] \wedge (u, int, t) \\
\Rightarrow & \{ \text{ Int rule } \} \\
 & u.v[s.p] < s.v[s.p] \wedge t.v = u.v \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 2: (u, snd, t)

$$
\begin{array}{ll}
 & u.v[s.p] < s.v[s.p] \wedge (u, snd, t) \\
\Rightarrow & \{ \text{ Snd rule, } s.p \neq t.p \ \} \\
 & u.v[s.p] < s.v[s.p] \wedge t.v[s.p] = u.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 3: $(u, rcv(w), t)$

$$
\begin{array}{ll}
 & u.v[s.p] < s.v[s.p] \wedge (u, rcv(w), t) \\
\Rightarrow & \{ \text{ Rcv rule } \} \\
 & u.v[s.p] < s.v[s.p] \wedge (u, rcv(w), t) \wedge (t.v[s.p] = u.v[s.p] \vee t.v[s.p] = w.v[s.p]) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p] \vee ((u, rcv(w), t) \wedge t.v[s.p] = w.v[s.p])
\end{array}
$$

For case 3, it suffices to prove the following two cases.

Case 3A: $w.p = s.p$

$$
\begin{array}{ll}
 & t.v[s.p] = w.v[s.p] \wedge (u, rcv(w), t) \wedge w.p = s.p \\
\Rightarrow & \{ \text{ let } x \text{ satisfy } w \prec_{im} x,\ x \text{ exists since } w \rightsquigarrow t \text{ implies } \neg Final(w) \ \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \\
\Rightarrow & \{ \text{ otherwise } s \rightarrow t \ \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \ \wedge \ w \rightarrow s \\
\Rightarrow & \{ \text{ since } w \prec_{im} x \ \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \ \wedge \ (x = s \vee x \rightarrow s) \\
\Rightarrow & \{ \text{ Snd rule } \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < x.v[s.p] \ \wedge \ (x = s \vee x \rightarrow s) \\
\Rightarrow & \{ \text{ lemma 7.7 } \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < x.v[s.p] \ \wedge \ x.v \leq s.v \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

Case 3B: $w.p \neq s.p$

$$
\begin{array}{ll}
 & t.v[s.p] = w.v[s.p] \wedge (u, rcv(w), t) \wedge w.p \neq s.p \\
\Rightarrow & \{ \text{ use } s \not\rightarrow_k t,\ k > 0, \text{ and lemma 7.6 } \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ w.p \neq s.p \ \wedge \ s \not\rightarrow_j w \ \wedge \ 0 \leq j < k \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < s.v[s.p] \\
\Rightarrow & \{ \text{ simplify } \} \\
 & t.v[s.p] < s.v[s.p]
\end{array}
$$

■

Lemma 7.9 is a refinement of lemma 7.7 for the case when $s.p \neq t.p$. Note that the result of lemma 7.8 is used in this proof, indicating that perhaps it is necessary to prove equation 7.1 in order to prove equation 7.2. This is interesting because intuition tells us that equation 7.2 should be easier to prove than equation 7.1.

Lemma 7.9 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Rightarrow s.v < t.v)$

Proof: It is sufficient to show that for all $k > 0$: $s \rightarrow_k t \wedge s.p \neq t.p \Rightarrow t.v <$

$s.v$. We use induction on k.

Base $(k = 1)$:

$$
\begin{array}{ll}
& s \rightarrow_1 t \wedge s.p \neq t.p \\
\Rightarrow & \{ \text{ defn of } \rightarrow_1 \text{ and lemma 7.2 } \} \\
& s \rightsquigarrow t \wedge s.p \neq t.p \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec_{im} t \ \} \\
& s.p \neq u.p \wedge (u, rcv(s), t) \\
\Rightarrow & \{ \text{ otherwise } t \rightarrow s \text{ (since there is only one event between } u \text{ and } t) \ \} \\
& u \not\rightarrow s \wedge s.p \neq u.p \wedge (u, rcv(s), t) \\
\Rightarrow & \{ \text{ lemma 7.8 and rcv rule } \} \\
& s.v[s.p] < u.v[s.p] \wedge (\forall i :: t.v[i] = max(u.v[i], s.v[i])) \\
\Rightarrow & \\
& s.v < t.v
\end{array}
$$

Induction $(k > 0)$:

$$
\begin{array}{ll}
& s \rightarrow_k t \wedge k > 0 \wedge s.p \neq t.p \\
\Rightarrow & \{ \text{ lemma 7.3 } \} \\
& (\exists u :: s \rightarrow_{k-1} u \wedge u \rightarrow_1 t \wedge s.p \neq t.p) \\
\Rightarrow & \{ \ u.p \text{ can not have two values } \} \\
& (\exists u :: s \rightarrow_{k-1} u \wedge u \rightarrow_1 t \wedge (u.p \neq t.p \vee u.p \neq s.p)) \\
\Rightarrow & \\
& (\exists u :: (s \rightarrow_{k-1} u \wedge u \rightarrow_1 t \wedge u.p \neq t.p) \vee (s \rightarrow_{k-1} u \wedge u \rightarrow_1 t \wedge u.p \neq s.p)) \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
& (\exists u :: (s \rightarrow_{k-1} u \wedge u.v < t.v) \vee (s.v < u.v \wedge u \rightarrow_1 t)) \\
\Rightarrow & \{ \text{ lemma 7.7 } \} \\
& (\exists u :: (s.v \leq u.v \wedge u.v < t.v) \vee (s.v < u.v \wedge u.v \leq t.v)) \\
\Rightarrow & \\
& s.v < t.v
\end{array}
$$

■

Theorem 7.1 states the property which we set out to prove at the beginning of this section.

Theorem 7.1 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Leftrightarrow s.v < t.v)$

Proof: Immediate from Lemmas 7.8 and 7.9. ∎

7.4 Direct Dependency Clock

Lamport's algorithm uses acknowledgments to implement a local clock which is equivalent to direct dependency clocks. In this section we define a direct dependency clock (DDClock) so that we may use it at an abstract level in the mutual exclusion algorithm.

Direct dependency clocks are a weaker version of vector clocks [41]. They require smaller message tags to implement (one integer vs. the number of processes, N), but they provide a weaker form of causality information. For many applications such as this one, the weaker version of the clock suffices.

7.4.1 Algorithm

The algorithm for maintaining a DDClock is described by the initial conditions and the actions taken for each event type.

For any initial state s:

$$(\forall i : i \neq s.p : s.v[i] = 0) \wedge (s.v[s.p] = 1)$$

Rule for a send event or an internal event (i.e., $(s \prec_{im} t) \wedge \neg(\exists u :: u \rightsquigarrow t)$):

$$t.v[t.p] = s.v[t.p] + 1$$

Rule for a receive event (i.e., $s \prec_{im} t \wedge u \rightsquigarrow t$):

$$t.v[t.p] = max(s.v[t.p], u.v[u.p]) + 1$$

$$t.v[u.p] = max(u.v[u.p], s.v[u.p])$$

7.4.2 Proof

It is easy to see from the implementation of DDClock that property $DD1$, shown below, holds:

$$s \prec_{im} t \vee s \rightsquigarrow t \Rightarrow \; s.v[s.p] < t.v[t.p] \wedge s.v[s.p] \leq t.v[s.p] \qquad (DD1)$$

Property $DD2$ follows from $DD1$ since there must be a chain between s and t and each link in the chain must satisfy $DD1$.

$$s \rightarrow t \Rightarrow \; s.v[s.p] < t.v[t.p] \wedge s.v[s.p] \leq t.v[s.p] \qquad (DD2)$$

Theorem 7.2 states the main DDClock property. It uses the relation $\mapsto$ which is a subset of $\rightarrow$. Given states s and t, $s \mapsto t$ is true if and only if $s \neq t$ and there is a path (ie, chain) from s to t which includes at most one message.

$$s \mapsto t \equiv s \preceq t \vee (\exists u, v : s \preceq u \wedge u \rightsquigarrow v \wedge v \preceq t)$$

Theorem 7.2 $\forall s, t : s \neq t : (s \mapsto t) \Leftrightarrow (s.v[s.p] \leq t.v[s.p])$

Proof:

First prove $s \neq t \wedge s \mapsto t \Rightarrow s.v[s.p] \leq t.v[s.p]$:

$$\begin{array}{ll} & s \mapsto t \\ \Rightarrow & \{ \text{ defn of } \mapsto \} \\ & s \rightarrow t \\ \Rightarrow & \{ \text{ property } \mathit{DD2} \} \\ & s.v[s.p] \leq t.v[s.p] \end{array}$$

Then prove $s \neq t \wedge \neg(s \mapsto t) \Rightarrow s.v[s.p] > t.v[s.p]$ by induction on the states at process $t.p$:

Base case: $Init(t)$

$Init(t) \Rightarrow t.v[s.p] = 0$. Also, $s.v[s.p] \geq 1$ for all s.

Thus, $s.v[s.p] > t.v[s.p]$.

Induction case: Assume theorem holds for $\bar{t}$, where $\bar{t} = t.prev$.

Case 1: $(\bar{t}, rcv(s'), t) \wedge s'.p = s.p$

From the algorithm, $t.v[s.p] = s'.v[s.p]$. If $\neg(s \mapsto t)$, then $s' \rightarrow s$, which implies $s'.v[s.p] < s.v[s.p]$ (since s' is send state and the local clock is incremented). Thus, $s.v[s.p] > t.v[s.p]$.

Case 2: Negation of case 1.

In this case, $t.v[s.p] = \bar{t}.v[s.p]$. Also, $s \neq t \wedge \neg(s \mapsto t)$ is equivalent to $s \neq \bar{t} \wedge \neg(s \mapsto \bar{t})$. Thus, by substitution into the theorem,

$s \neq t \wedge \neg(s \mapsto t) \Rightarrow s.v[s.p] > t.v[s.p]$. ∎

7.5 Mutual Exclusion

Let a system consist of a fixed number of processes and a shared resource called the critical section. It is required that no more than one process use the critical section at any time. The algorithm to coordinate access to the critical section must satisfy the following properties which are stated informally in the time domain. They will be given in the causality domain when formalized.

Safety: Two processes should not have permission to use the critical section simultaneously.

Liveness: Every request for the critical section is eventually granted.

Fairness: Different requests must be granted in the order they are made.

Now the problem is formalized in the causality domain. To start, Lamport's algorithm assumes that all channels are FIFO, which can be stated as follows:

$$s \prec t \wedge s \rightsquigarrow u \wedge t \rightsquigarrow v \Rightarrow \neg(v \prec u)$$

For any state s, we define $req(s)$ to be true if and only if the process $P_{s.p}$ has requested the critical section and has not yet released it, and $cs(s)$ to be true if and only if the process $P_{s.p}$ has permission to enter the critical section in the state s. Note that $req(s)$ and $cs(s)$ are predicates, not program variables. They are a function of process state and will be defined formally in the algorithm.

Both req and cs are false in an initial state. Now suppose $t \prec u \prec v$ and that a request for the critical section was made in t, access was granted in u, and it was released in v. Then $req(s)$ is true for all states s such that $t \preceq s \preceq v$, and $cs(s)$ is true for $u \preceq s \preceq v$. It is assumed that a process which is granted access to the critical section eventually releases it:

$$cs(s) \Rightarrow (\exists t : s \prec t : \neg req(t)) \qquad \textbf{(Cooperation)}$$

The task is to develop a distributed algorithm to ensure the required safety, liveness and fairness properties. The safety and liveness properties can be stated formally in the causality domain as follows:

$$s \| t \Rightarrow \neg(cs(s) \wedge cs(t)) \qquad \textbf{(Safety)}$$

$$req(s) \Rightarrow (\exists t :: s \prec t \wedge cs(t)) \qquad \textbf{(Liveness)}$$

Before presenting the causality based fairness property, a little ground work is needed. First, let

$$next\_cs(s) = min\{t | s \prec t \wedge cs(t)\}$$

Informally, $next\_cs(s)$ is the first local state after s in which the process $P_{s.p}$ has access to the critical section. Also, define a boolean function $req\_start(s)$ as follows:

$$req\_start(s) = req(s) \wedge \neg req(s.prev)$$

Thus, $req\_start(s)$ is true if and only if $P_{s.p}$ made a request for the critical section in state s. Then, the fairness property can be stated as:

$$(req\_start(s) \wedge req\_start(t) \wedge s \to t) \Rightarrow next\_cs(s) \to next\_cs(t) \quad \textbf{(Fairness)}$$

Note that $next\_cs(s)$ and $next\_cs(t)$ exist due to liveness. Furthermore, $next\_cs(s)$ and $next\_cs(t)$ are not concurrent due to safety. Therefore,

$$next\_cs(s) \to next\_cs(t)$$

is equivalent to

$$\neg(next\_cs(t) \to next\_cs(s))$$

7.5.1 Algorithm

An informal description of Lamport's mutual exclusion algorithm [34] is given, followed by a formal description in the causality domain. In the informal description each process maintains a logical clock (used for timestamps) and a queue (stores requests for the critical section).

- To request the critical section, a process sends a timestamped message to all other processes and adds a timestamped request to the queue.
- On receiving a request message, the request and its timestamp is stored in the queue and an acknowledgment is returned.
- To release the critical section, a process sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if and only if: 1) it has a request in the queue with timestamp t, and 2) t is less than all other requests in the queue, and 3) it has received a message from every other process with timestamp greater than t (the request acknowledgments ensure this).

The formal description is shown below. The DDClock is maintained as described in section 7.4.2. In local state s, variable $s.v$ refers to the value of the DDClock. Thus, $s.v$ is the timestamp of state s. The array $s.q$ functions as the queue (an array suffices since processes submit one request at a time).

- Local variables in each state s:

  $s.q[1..n]$: integer, each element initially ∞

  $s.v$: DDClock

- To request the critical section in t where $s \prec_{im} t$:

  $t.q[t.p] = s.v[t.p]$

  for all $j : j \neq t.p$: send "request" to P_j

- On receiving "request" in state t which was sent from state u (i.e., $u \leadsto t$):

  $t.q[u.p] = u.q[u.p]$

  send ack to $u.p$

- To release the critical section in state t:

  $t.q[t.p] = \infty$

  for all $j \neq t.p$, send "release" to P_j

- On receiving "release" in state t which was sent from state u:

  $t.q[u.p] = \infty$

State s has received a request from P_i if $s.q[i] \neq \infty$, in which case the timestamp of the request is the value of $s.q[i]$. State s has permission to access the critical section when there is a request from $P_{s.p}$ with timestamp less than all other requests and $P_{s.p}$ has received a message from every other process with a timestamp greater than the timestamp of its own request. We use the predicates $req(s)$ and $cs(s)$ to denote that a request has been made and access has been granted. In the definitions shown below, the $<$ relation on tuples is a lexicographic ordering. In the rest of the paper, we write $s.q[s.p] < s.v[j]$ instead of $(s.q[s.p], s.p) < (s.v[j], j])$ for notational simplicity.

$$req(s) \equiv s.q[s.p] \neq \infty$$

$$cs(s) \equiv (\forall j : j \neq s.p : \ (s.q[s.p], s.p) < (s.v[j], j) \ \wedge \ (s.q[s.p], s.p) < (s.q[j], j) \ )$$

7.5.2 Proof

We define the predicate

$$msg(s,t) \equiv (\exists u, t' : u \rightsquigarrow t' \wedge u \prec s \wedge t \prec t')$$

That is, there exists a message which was sent by $P_{s.p}$ before s and received by $P_{t.p}$ after t.

The mutual exclusion algorithm satisfies the property stated in lemma 7.10. Intuitively, the lemma states that if $s \not\rightarrow t$ and no message sent before s arrives after t, then $t.q[s.p] = s.q[s.p]$.

Lemma 7.10 *Assume FIFO.* $\forall s, t : s.p \neq t.p : s \not\rightarrow t \wedge \neg msg(s,t) \Rightarrow t.q[s.p] = s.q[s.p]$.

Proof: We will use induction on $\not\rightarrow_k$.

Base Case: $(k = 0) \equiv Init(t)$

If $Init(s)$, then the result follows from the initial assignment. Otherwise, let u be the initial state in the process $s.p$. From the program text, any change in $s.q[s.p]$ is followed by a message send to all processes. Given this and $\neg msg(s,t)$ it follows that $s.q[s.p] = u.q[s.p]$. (This argument can be formalized using induction on the number of states that precede s in the process $s.p$.) From initial assignment, it again follows that $t.q[s.p] = s.q[s.p]$.

Induction case: $s \not\rightarrow_j t.prev, j < k$

Let $u = t.prev$. Let $event(u) \neq receive$, then $\neg msg(s,t)$ implies $\neg msg(s,u)$. Using the induction hypothesis, we get that $u.q[s.p] = s.q[s.p]$ and by using program text, we conclude that $t.q[s.p] = u.q[s.]$. Now let $event(u) = receive(w)$. If $w.p \neq s.p$, the previous case applies.

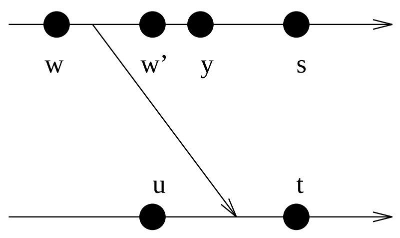

Figure 7.2: Proof for the induction case for lemma 7.10

So, let $w.p = s.p$. Since $s \not\rightarrow t$, it follows that $w \prec s$. Let $w' = w.next$. From program, $w'.q[s.p] = t.q[s.p]$. We now claim that $s.q[s.p] = w'.q[s.p]$ from which the result follows. If not, there exists y such that $w' \preceq y \preceq s$ and $y.prev.q[s.p] \neq y.q[s.p]$. From the program, there exists a message from y to the process $t.p$ received in the state z (assuming reliable messages). The condition $t \prec z$ violates $\neg msg(s,t)$. The condition $t = z$ is not possible since exactly one message is received before t which is w. And $z \prec t$ violates FIFO since $w \prec y$ but $z \prec t$. ∎

The following lemma is crucial in proving the safety property. The remaining theorems prove that the algorithm satisfies the required properties: safety, liveness, and fairness.

Lemma 7.11 *Assume message channels are FIFO:* $s.p \neq t.p \wedge s \not\rightarrow t \wedge s.q[s.p] < t.v[s.p] \;\Rightarrow\; t.q[s.p] = s.q[s.p]$

Proof: Using lemma 7.10, we note that it is sufficient to show that the stated the assumptions and the antecedent imply $\neg msg(s,t)$. Let $u \prec s$ be such that

$uvsp = sqsp$. That is, the request was made in state u. Let the message sent at u be received by process $t.p$ in state w. From the event descriptions, it follows that there is neither a request message nor a release message sent after u and before s. (1)

Since $uvup < tvup$, from the property of dependency clocks, we get that $u \mapsto t$. Therefore there exists a message sent at or after u and received before t. From FIFO, it follows that $w \preceq t$. From FIFO, and the fact that the message sent at u is received before t, it follows that $\neg msg(u, t)$. (2)

From (1) and (2), it follows that $\neg msg(s, t)$. ■

Theorem 7.3 (Safety) $s.p \neq t.p \wedge s \| t \;\; \Rightarrow \;\; \neg(cs(s) \wedge cs(t))$.

Proof: We will show that $(s \| t) \wedge cs(s) \wedge cs(t)$ implies false.

Case 1: $t.v[s.p] < s.q[s.p] \wedge s.v[t.p] < t.q[t.p]$

We get the following cycle.

$$
\begin{array}{ll}
 & s.q[s.p] \\
< & \{ \; cs(s) \wedge s.p \neq t.p \; \} \\
 & s.v[t.p] \\
< & \{ \text{ this case } \} \\
 & t.q[t.p] \\
< & \{ \; cs(t) \wedge s.p \neq t.p \; \} \\
 & t.v[s.p] \\
< & \{ \text{ this case } \} \\
 & s.q[s.p].
\end{array}
$$

Case 2: $s.q[s.p] < t.v[s.p] \wedge t.q[t.p] < s.v[t.p]$

We get the following cycle.

$$
\begin{array}{ll}
& s.q[s.p] \\
< & \{\ cs(s) \wedge s.p \neq t.p\ \} \\
& s.q[t.p] \\
= & \{\ t.q[t.p] < s.v[t.p],\ t \not\to s, \text{ Lemma 7.11 }\} \\
& t.q[t.p] \\
< & \{\ cs(t) \wedge s.p \neq t.p\ \} \\
& t.q[s.p] \\
= & \{\ s.q[s.p] < t.v[s.p],\ s \not\to t, \text{ Lemma 7.11 }\} \\
& s.q[s.p].
\end{array}
$$

Case 3: $s.q[s.p] < t.v[s.p] \wedge s.v[t.p] < t.q[t.p]$

We get the following cycle.

$$
\begin{array}{ll}
& s.q[s.p] \\
< & \{\ cs(s) \wedge s.p \neq t.p\ \} \\
& s.v[t.p] \\
< & \{\ \text{this case}\ \} \\
& t.q[t.p] \\
< & \{\ cs(t) \wedge s.p \neq t.p\ \} \\
& t.q[s.p] \\
= & \{\ s.q[s.p] < t.v[s.p],\ s \not\to t, \text{ Lemma 7.11 }\} \\
& s.q[s.p].
\end{array}
$$

Case 4: Similar to case 3. ∎

Theorem 7.4 (Liveness) $req(s) \ \Rightarrow\ \exists t : s \prec t \wedge cs(t)$

Proof: $req(s)$ is equivalent to $s.q[s.p] \neq \infty$. $s.q[s.p] \neq \infty$ implies that there exists $s_1 \in P_{s.p}$ such that $s_1.v[s.p] = s.q[s.p] \wedge event(s_1) = request$.

We show existence of the required t with the following two claims:

Claim 1: $\exists t_1 : \forall j \neq s.p : t_1.v[j] > s.q[s.p] \wedge s.q[s.p] = t_1.q[s.p]$

Claim 2: $\exists t_2 : \forall j \neq s.p : t_2.q[j] > s.q[s.p] \wedge s.q[s.p] = t_2.q[s.p]$

By choosing $t = max(t_1, t_2)$ and verifying that $cs(t)$ holds we get the desired result.

Claim 1 is true because the message sent at s_1 will eventually be acknowledged. It is enough to note that $\forall j : j \neq s.p : \exists w_j \in P_j : s_1 \leadsto w_j$. From the program, we get that on receiving request, the message is acknowledged. Thus, $\forall j : j \neq s.p : \exists u_j \in P_i : w_j \leadsto u_j$. By defining $t_1 = max\ j : j \neq s.p : u_j$, and observing that for any j, $w_j.v[j] > s.q[s.p]$, we get the claim 1.

To show claim 2, we use induction on the number of requests smaller than $s.q[s.p]$ in $t_1.q$. We define

$$nreq(u) = | \ \{k \mid u.q[k] < u.q[u.p]\} \ |.$$

If $nreq(t_1) = 0$, then $s.q[s.p]$ is minimum at $t_1.q$ and therefore $cs(t_1)$ holds. Assume for induction that the claim holds for $nreq(t_1) = k, (k \geq 1)$. Now, let $nreq(t_1) = k + 1$. Consider the process with the smallest request, that is assume that $t_1.q[j]$ is minimum for some j. Let u be the state in P_j such that $u.v[j] = t_1.q[j]$. We claim that $nreq(u) = 0$. If not, let m be such that $u.q[m] < u.q[u.p]$. This implies that $u.q[m] < u.q[u.p] < s.q[s.p]$. Since

$s.q[s.p] < t_1.v[m]$, from FIFO it follows that $t_1.q[m] = u.q[m]$. However, $u.q[u.p]$ is the smallest request message; a contradiction.

Therefore, we know that process $u.p$ will enter critical section and thus eventually set its $q[u.p]$ to ∞. This will reduce the number of requests at $t_1.p$ by 1. ■

Theorem 7.5 (Fairness) $(req\_start(s) \wedge req\_start(t) \wedge s \rightarrow t) \;\Rightarrow\; (next\_cs(s) \rightarrow next\_cs(t))$

Proof: Let $s' = next\_cs(s)$ be state in which critical section is acquired, and let s'' be state which it is released. Let $t' = next\_cs(t)$.
Let r be the state in $P_{t.p}$ which received the request message sent from s.
We know the following facts:

1. $r \preceq t$, due to FIFO channels.
2. $t.v[t.p] = t.q[t.p]$, due to request event at t.
3. $s.v[s.p] < t.v[t.p]$, since $s \rightarrow t$ $(DD2)$.
4. $s.q[s.p] = s.v[s.p]$, due to request event at s.
5. $r.q[s.p] = s.q[s.p]$, due to receiving request at r.
6. $r.q[s.p] < t.q[t.p]$, from $2, 3, 4, 5$.
7. $t.q[t.p] = t'.q[t.p]$, by defn of t'.
8. $t'.q[t.p] \leq t'.q[s.p]$, since $cs(t')$.
9. $r.q[s.p] < t'.q[t.p] \leq t'.q[s.p]$, from $6, 7, 8$.

This means that $q[s.p]$ must be increased between r and t'. That can only happen when $P_{t.p}$ receives the release message sent from s''. Thus $s'' \rightarrow t'$. And since $s' \rightarrow s''$, we conclude $s' \rightarrow t'$. ■

Chapter 8

Conclusions and Future Research

The software development process can be divided into seven phases: requirements, specification, design, development, testing, acceptance and maintenance. Program observation and verification are important parts of the specification, design, development and testing phases.

Verification is used in the specification and design phases to prove that the design meets the specification. In the development phase, observation is used by the programmer in order to track down bugs and clarify understanding of the program's behavior. In the testing phase, observation can be used to ensure that the program behaves as expected for a given set of inputs.

The fundamental difference between observation and verification is that observation applies to individual executions of a program and verification applies to all executions. Observation is relatively simple for sequential systems, but quite difficult for distributed systems due to the lack of global information. On the other hand, verification is an inherently difficult problem for all software systems, sequential or distributed.

This dissertation described three techniques for observation of distributed systems and one techniques for verification of distributed systems. The three observation techniques are global functions, recursive logic (RCL), and DAG expressions.

A global function has the form $x_1 \oplus x_2 \oplus \ldots x_n$ where x_i is a variable local to process i. An execution of a distributed program is monitored to determine if the value of the function exceeds some constant K. The $\oplus$ operator and the domain of the variables x_i must meet certain requirements which are described in the text. Two special cases of this problem were considered. The first case (referred to as "two variables") imposed the restriction that there are only two variables ($n = 2$). Notice that there can be more than two processes, but only two can contribute local variables to the global function. The second case (referred to as "bivalent variables") restricts the domain of x_i to be a set with two elements, one of which must be the identity element of $\oplus$. In both cases an algorithm was given for observing a computation in order to determine the maximum value of the global function over all global states in the computation.

There are several open problems relating to global functions. A general problem is: "What classes of global functions can be efficiently monitored on distributed programs?". Another more specific problem is: "Can we efficiently detect more general cases the two special cases considered in this dissertation?". For example, can these results be extended to consider N integer variables at N processes.

RCL is a recursive logic which is used to write predicates about the structure of a computation. 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. A distributed algorithm for online detection of RCL formulas is presented, proven correct and analyzed for complexity.

Future work with regard to RCL involves extending the logic so that any partial order of global conjunctive predicates can be detected. This does not seem to be a difficult problem since the dataflow description of the algorithm would seem to work with little or no modification.

DAG expressions are a simple but powerful method for observing properties of distributed computations related to the flow of information in the computation. DAG expressions are an extension of regular expressions to partial orders. A decentralized algorithm for observing a computation to determine of a DAG expression is satisfied was described and proven to be correct. The algorithm does not alter the causal structure of the computation (so it does not inhibit or initiate sending or receiving of messages). The algorithm complexity does not depend on the number of processes, but it does depend on the expression which is being detected.

The verification part of the dissertation presented a system for specifying and proving distributed programs that is unique in many respects. A program is a specification of the relationship between local states which are adjacent to each other in the poset. A program property is a property of posets which holds for any poset generated by that program. Poset properties are often succinct and elegant because they inherently account for varying processor execution speeds. The system utilizes a new proof technique, induction on the complement of the causally precedes relation, which is useful in proving poset properties. The verification techniques were demonstrated on two well known algorithms (vector clocks and mutual exclusion) and one lesser known algorithm (direct dependency clocks).

In the bigger picture, one area of interesting and future research involves the testing phase of the software project lifecycle. The work in this dissertation largely treated specification and testing as two independent tasks. However, it seems a lot could be gained from a closer coupling of these phases. For example, there could be mechanisms for automatically testing a program against its specification.

Programming languages usually cater to the computer (easy to compile) or to the user (power of expression). An interesting area of research is to determine what benefits can be gained by designing a language that caters to the entire software lifecycle.

BIBLIOGRAPHY

[1] M. Ahuja. Flush primitives for asynchronous distributed systems. *Information Processing Letters*, 34:5–12, 1990.

[2] M. D. Atkinson. The complexity of orders. In I. Rival, editor, *Algorithms and Orders*, volume 255 of *NATO ASI Series, Mathematical and Physical Sciences*, pages 195–230. 1987.

[3] 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.

[4] F. Baiardi, N. De Francesco, and G. Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[5] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems and Software*, 4(3):255–264, December 1983.

[6] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[7] 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.

[8] C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem in distributed systems. Technical Report TR-PDS-1995-005, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1995.

[9] 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.

[10] Sarah E. Chodrow and Mohamed G. Gouda. Implementation of the sentry system. Technical Report CS-TR-94-01, University of Texas at Austin, Dept. of Computer Sciences, May 1994.

[11] 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.

[12] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–166, 1950.

[13] 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.

[14] 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.

[15] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[16] 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.

[17] V. K. Garg and A. I. Tomlinson. Causality versus time: How to specify and verify distributed programs. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, October 1994.

[18] 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.

[19] 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.

[20] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, Submitted.

[21] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. Technical Report TR-PDS-1994-003, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994. available via ftp or WWW at maple.ece.utexas.edu.

[22] V. K. Garg and J. Ghosh. Repeated computation of global functions in a distributed environment. *IEEE Transactions on Parallel and Distributed Systems*, 5, 1994.

[23] V. K. Garg, A. I. Tomlinson, E. Fromentin, and M. Raynal. An efficient decentralized algorithm for detecting properties of distributed computations. Technical Report TR-PDS-1994-004, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994. available via ftp or WWW at maple.ece.utexas.edu.

[24] W. Wyat Gibbs. Software's chronic crisis. *Scientific American*, pages 86–95, September 1994.

[25] 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.

[26] Paul K. Harter, Jr., Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498–506, Denver CO, May 1985.

[27] J. M. Helary, N. Plouzeau, and M. Raynal. Computing particular snapshots in distributed systems. In *Proc. of the 9-th annual International Phoenix Conference on Computers and Communications*, pages 116–123, Scottsdale, AR, March 1990.

[28] 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.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, October 1969.

[30] 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).

[31] 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. (Reprinted in SIGPLAN Notices, Dec. 1993).

[32] Syed Irfan Hyder. *A Unified Approach to Concurrent Debugging.* PhD thesis, University of Texas at Austin, Dept. of Computer Sciences, Austin, TX, December 1994.

[33] C. Jard, T. Jeron, G. V. Jourdan, and J. X. Rampon. A general approach to trace-checking in distributed computing systems. In *Proc. of the International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

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

[35] Leslie Lamport and Nancy Lynch. Distributed Computing: Models and Methods. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18. Elsevier Science Publishers B. V., 1990.

[36] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[37] H. F. Li, T. Radhakrishnan, and K. Venkatesh. Global state detection in non-fifo networks. In *Proc. of the* 7^{th} *International Conference on Distributed Computing Systems*, pages 364–370, 1987.

[38] Yoshifumi Manabe and Makoto Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

[39] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proc. of the 5th Workshop on Distributed Algorithms*, pages 254–272, Delphi, Greece, October 1991.

[40] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434.

[41] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[42] 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.

[43] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 135–144. ACM SIGPLAN, June 1988.

[44] 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.

[45] M. Naimi. Global stability detections in the asynchronous distributed computation. In *Proc. of the IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 87–92, Hong Kong, September 1988.

[46] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[47] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual IEEE-ACM Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[48] M. Raynal. About logical clocks for distributed systems. *ACM Operating Systems Review*, 26(1):41–48, 1992.

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

[50] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[51] A. I. Tomlinson and V. K. Garg. An algorithm for minimally latent global virtual time. In *7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, May 1993.

[52] A. I. Tomlinson and V. K. Garg. Using induction to prove properties of distributed programs. In *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 478–485, Dallas, TX, December 1993. IEEE.

[53] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed systems. In N. Balakrishnan, T. Radhakrishnan, D. Sampath, and S. Sundaram, editors, *Proc. of the Intl. Conf. on Computer Systems and Education*, pages 257–272, New Delhi, June 1994. Tata McGraw-Hill.

[54] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. Technical Report TR-PDS-1994-006, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994. available via ftp or WWW at maple.ece.utexas.edu.

[55] 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).

[56] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science - Volume B: Formal Models and Semantics*. Elsevier, Amsterdam, 1990.

[57] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *Proc. of the Thirtieth Annual Allerton Conference on Communication, Control and Computing*, October 1992.

VITA

Alexander Ivor Tomlinson was born in Columbus, Ohio, on April 6, 1964, the son of Ruth Beatty and Harry Reading Tomlinson. After graduating from Lawrence High School in Lawrenceville, New Jersey (1982) he entered Carnegie Mellon University in Pittsburgh, Pennsylvania. He received a Bachelor of Science in Electrical and Computer Engineering from Carnegie Mellon University in 1986 and began work with Hewlett Packard Company in Roseville, California. He worked with Digital Telephone Systems (a subsidiary of Harris Corporation) in Novato, California for six months before enrolling in the graduate program at The University of Texas at Austin in 1990. He received a MS in Computer Engineering at UT Austin in August 1992 and a PhD in 1995.

Permanent address: Alex may be contacted via email at tomlinson@acm.org

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.