AVOIDING STATE ENUMERATION IN DYNAMIC CHECKING OF DISTRIBUTED PROGRAMS

Eslam Al Maghayreh

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

April 2008

Your file Votre référence
ISBN: 978-0-494-37743-7
Our file Notre référence
ISBN: 978-0-494-37743-7

Abstract

Avoiding State Enumeration in Dynamic Checking of Distributed Programs

Eslam Al Maghayreh, Ph.D.
Concordia University, 2008

Distributed programs are particularly vulnerable to software faults. Bugs in these programs are usually very hard to detect without automatic verification. The idea of checking an expected property in a given distributed program run (also referred to as runtime verification) has recently been attracting a great deal of attention for analyzing execution traces to ensure the reliability and dependability of distributed programs.

Due to concurrency, the number of global states of a distributed program run tends to grow exponentially with respect to the number of program statements executed. As a result, checking the satisfaction of a property in a given distributed program run can incur significant overhead. This thesis introduces various ideas and exploits them to develop efficient dynamic property checking algorithms. These include the use of atoms, introducing and exploiting the notion of serialization and finally proposing a methodology that exploits the concept of atoms and partial order semantics to specify and to check properties of distributed programs.

The abstract specification of a distributed program can be mapped to the lower level implementation by labeling the code blocks that belong to the abstract functionalities of the program that are expected to be performed atomically. Each labeled code block is called an atom. Dynamically, an atom includes all the events that

result from executing the selected statements from the corresponding code block. An efficient on-the-fly atomicity error detection algorithm has been developed. It is shown that if a run of a distributed program is atomic then the required properties can be checked on a reduced lattice, referred to as the atomic state lattice, which is significantly smaller than the original state lattice.

Even with atomization, the number of global states can still grow exponentially in the number of atoms executed. However, when a number of processes has to maintain a property, we expect that each process will be, at some point in time, aware of the events of other processes that may affect the property. Consequently, it is not necessary to check the property in each state. Only synchronized states, where processes have already exchanged the information necessary to maintain the property, need to be considered. These states can be characterized by a synchronization predicate. Serialization of synchronized states is the minimal avenue for a set of processes to exchange the necessary information to maintain a property. Two efficient algorithms to check the satisfaction of a property in a distributed computation in cases where the synchronization predicate is conjunctive or disjunctive have been developed.

Finally, a methodology based on the concept of atoms and a partially ordered multi-set (POMSET) model to specify and to check distributed programs properties has been proposed. The POMSET model promotes the separation of two different concerns in specifying and checking properties, namely, the ordering requirements and the computational requirements. A methodology to specify and to efficiently check the two requirements has been introduced.

Acknowledgments

Praise be to Almighty God for giving me strength and patience, and for putting in my way people who have helped in accomplishing this work.

I would like to express my special thanks, deep gratitude and appreciation to my supervisors Dr. H. F. Li and Dr. D. Goswami for their valuable help throughout the course of this research.

I am very thankful to my parents for constantly supporting and encouraging me. I am grateful to them for all they have given me throughout my life.

I also thank my committee members, Dr. E. M. Aboulhamid, Dr. J. William Atwood, Dr. R. Jayakumar and Dr. A. Agarwal for all of their helpful comments and suggestions.

I would like to thank all of my brothers and sisters for their moral support and encouragement.

I would like to thank all of my friends for all kinds of help and support.

Finally, I am very thankful to everyone who has directly or indirectly helped me, in any way, to reach this Goal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Distributed programs are particularly hard to write and to reason about their correctness due to their asynchronous nature where processes do not share any memory or clock. Verification and validation of distributed programs and software fault-tolerance are important ways to ensure reliability of distributed programs. Detecting a fault in an execution of a distributed program is a fundamental problem that arises during the verification process. A programmer, upon observing a certain distributed computation, can check whether the observed computation satisfies some expected properties in order to detect bugs.

Traditional methods to detect bugs in concurrent programs include testing and formal methods such as model checking and theorem proving. Testing is an ad hoc technique that does not allow for formal specification and verification of the properties that the program needs to satisfy. Formal methods work on an abstract model of the program. Consequently, even if a program has been formally verified, we

still cannot be sure of the correctness of a particular implementation. However, for highly dependable systems, it is important to analyze the particular implementation [GMS06, OBFR96].

The idea of checking whether a distributed program run satisfies an expected property (runtime verification) has recently been attracting a lot of attention for analyzing distributed computations [GLM07, GMS06, SLS05, Gar02, CG98, OBFR96, CG95]. Runtime verification is more practical than other verification methods such as model checking and testing. Runtime verification verifies the implementation of the system directly rather than verifying a model of the system as done in model checking. It is also based on formal logics and provides formalism, which is lacked in testing [OBFR96].

The execution of a distributed program involves the concurrent advancement of multiple processes that work together to perform the necessary functions. Due to concurrency, the number of global states of an execution can grow exponentially with respect to the number of program statements executed. A set of n concurrent events forms 2^n states. As a result, deciding whether a distributed computation satisfies a global predicate (property) can incur significant overhead. It has been proved that the detection of a general property in a given distributed program computation is NP-complete [CG98].

1.2 Research Goals

From the previous section we can see that analyzing a distributed computation is essential to ensure reliability and dependability of such systems, and at the same time is difficult due to the state explosion problem.

Researchers have introduced several ideas to overcome this problem. In [CG98], the structure of the property to be checked in a given distributed program run is exploited to develop efficient property checking algorithms. In [MG01], the concept of computation slicing has been introduced as an abstraction technique to facilitate analyzing distributed computations. In [Lam90], Lamport has introduced an atomicity theorem that can be used to reduce the state space that needs to be considered for property checking. Chapter 2 contains a more comprehensive discussion of the above and other strategies used to tackle the state explosion problem.

This research continues the effort to tackle the state space explosion in global property checking. In this thesis, we investigate various ideas to reduce the cost of checking a property in an observed distributed computation. These include the use of atoms, introducing and exploiting the notion of serialization and finally proposing a methodology that exploits the concept of atoms and partial order semantics to specify and to check properties of distributed programs.

Design time knowledge can be used to abstract a distributed program into a set of atoms. Statically, each atom is mapped into a code block in the program. Dynamically, an atom is mapped to the events resulting from the execution of the corresponding code block. We have shown that the notion of atoms is effective in reducing the state space that needs to be considered in property checking.

When a number of processes have to maintain a shared property, we expect that each process should be, at some point in time, aware of the events of other processes that may affect the shared property; otherwise it will be hard for these processes to maintain the shared property. Based on this observation we have proposed a property checking procedure that involves three steps: identifying the set of states where the property needs to be checked, checking that these states form a sequence in which

each state is a proper subset of the next, and finally checking the property in each identified state. We refer to the requirement in the second step as the serialization requirement. This requirement guarantees that the number of states that need to be considered in property checking will be bounded by the number of atoms executed. Consequently, the cost of property checking will be significantly reduced.

A partial order model of a distributed computation represents concurrency explicitly as opposed to an interleaving model where concurrency is implicitly modeled by commuting the corresponding events in separate linear traces. Therefore a partial order model is a more realistic representation of concurrency. We have introduced a methodology that combines the benefits of atoms and partial order semantics to specify the properties of a distributed program in a way that can be efficiently checked in a given distributed computation. The following section describes the contributions of this thesis in more detail.

1.3 Contributions

There are different layers of abstraction relevant in considering the correctness of a distributed program. In the lower layer associated with the coding platform, the execution of each program statement is an atomic event. In the higher layer, each process of a distributed program executes atoms. A code block can be statically identified as an atom. At run time, an atom includes all the events that result from executing the statements in the corresponding code block. The result of the events in each atom should be as if they were executed in sequence without interleaving with any event of any other atom. The work presented in Chapters 3 and 4 adopts the use of design time knowledge to identify the set of atoms that can be executed by a given distributed program.

Given the set of atoms and a distributed program run, we have defined what an atomic run is, and developed an efficient on-the-fly algorithm to detect atomicity errors. We have proved that if the run is atomic then a reduced state lattice exists, referred to as the atomic state lattice. If the run is atomic then properties can be checked on the atomic state lattice, which is much smaller than the original state lattice and hence the cost of property checking can be significantly reduced [LMG07a, LMG07b].

A programmer may incorrectly or incompletely identify the set of atoms in a given distributed program. We have studied the effects of this problem on the conclusions about the atomicity of a run and the satisfaction of a property in the atomic state lattice. Namely, we have answered the following questions: assuming that the programmer has incompletely or incorrectly identified the atoms, will the conclusion about the atomicity of the run remain valid? Will the conclusion about the satisfaction of a property in a given run remain valid? [LMG07b].

Although the purpose of atomicity error checking is to ensure the atomicity of the run and hence the existence of the atomic state lattice where properties can be checked more efficiently, the presence of atomicity errors can also be an indication of the presence of errors in the program that a programmer needs to locate and fix.

Even with atomization, it is still difficult to check certain properties in large distributed programs. We have introduced and exploited the notion of serialization to efficiently check general properties in large distributed programs. Processes that have to maintain a shared property need to exchange the information necessary to maintain the property. This indicates that it is not necessary to check a property in every state. Only synchronized states, where processes have already exchanged the necessary information to maintain the property, need to be considered in property

checking. These states can be characterized by a synchronization predicate.

However, the number of such synchronized states can be exponential and hence it is still difficult to check general properties. To deal with this problem we have exploited the following idea: instead of directly concentrating on checking the satisfaction of a property in a given distributed computation, we will first try to answer the following question: is it reasonable to expect that the distributed program will be able to maintain the property under consideration? Earlier, we have mentioned that processes that need to maintain a shared property have to be aware of the events of each other that may affect the property, and hence if this requirement is satisfied then we can expect that the program will be capable of maintaining the expected property. This requirement can be guaranteed if the synchronized states are serialized; meaning that each synchronized state is a proper subset of the next. The serialization requirement guarantees that the number of synchronized states where a property needs to be checked will be bounded by the number of atoms executed.

Based on the above ideas, we have introduced a property checking procedure that involves three tasks. The first task is to identify the synchronization predicate that characterizes the states where the property under consideration needs to be checked. The second task is to check that all of the synchronized states satisfy the serialization requirement, and the final task is to check the property in each synchronized state. If the set of synchronized states is serialized and each state in it satisfies the property that we need to check, then we can conclude that the given distributed computation satisfies the expected property.

We have developed two algorithms to check the satisfaction of a general property in a given distributed computation in the cases where the synchronization predicate is conjunctive or disjunctive [LM07b]. It is important to note that the above tasks

need not be accomplished in sequence; they can overlap as it will be illustrated later in this thesis.

Finally, we have proposed a methodology that exploits the concept of atoms and partial order semantics to specify and to check properties of distributed programs efficiently. Atoms are used to simplify the analysis by compressing the events of a distributed computation into a much smaller number of atoms. We have shown that reordering of independent events in a process can be exploited to merge a sequence of well-formed atoms into a larger well-formed atom. As a result, the atomization will contain a smaller number of atoms leading to a reduction in the cost of monitoring/checking. Moreover, since not every atom is directly relevant to a required program property, we have introduced the notion of an atom slice. An atom slice contains the smallest number of atoms that need to be monitored and checked [LM07a].

Partial order semantics promotes the separation of two different concerns in specifying and checking properties of a distributed computation. The first concern is the necessary ordering among atoms, and the second is the correctness of each atom. We have adapted a POMSET automaton model to specify the ordering requirements that a given distributed program must satisfy. Computational requirements are modeled as a general predicate that needs to be satisfied at the minimal prefix of each atom. A tool for checking distributed programs based on this methodology is currently being implemented by other members of our research group.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows: Chapter 2 reviews the background and discusses the related work in the literature. Chapter 3 presents the model of atomicity and explains in detail how atoms can be useful in reducing the state space that needs

to be considered in property checking. This chapter also presents an efficient on-the-fly atomicity error detection algorithm along with its proof of correctness. Chapter 4 discusses the effects of incomplete and/or incorrect knowledge of atoms on the conclusions made about the atomicity of a given distributed computation and the satisfaction of a property in that computation. Chapter 5 introduces the concepts of synchronized atoms and serialization and explains how it can be exploited to develop efficient property checking algorithms. Chapter 6 presents a hierarchy of atoms that can be used for the purpose of abstracting a distributed program computation. It also proposes a methodology that exploits the concept of atoms and a partially ordered multi-set (POMSET) model of a distributed computation to specify and efficiently check the satisfaction of a property in a given distributed program computation. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

2.1 Introduction

Distributed programs are particularly vulnerable to software faults. These programs often contain bugs, which are very difficult to detect without automatic verification. Verification of distributed programs and software fault-tolerance are important ways to ensure the reliability of distributed systems. Detecting a fault in an execution of a distributed system is a fundamental problem that arises during the verification process. A programmer upon observing a certain distributed computation for bugs can check whether the observed computation satisfies some expected property.

The idea of checking whether a distributed program run satisfies an expected property (also referred to as ***runtime verification***) has recently been attracting a lot of attention for analyzing execution traces [GLM07, GMS06, SLS05, Gar02]. Runtime verification is more practical than other verification methods such as model checking and testing. Runtime verification verifies directly on the implementation of the system rather than on the system model as done in formal verification methods

such as model checking. It is also based on formal logics and provides formalism, which is missing in testing.

The execution of a distributed program involves the concurrent advancement of multiple processes that work together to perform the necessary functions. Due to concurrency, the number of global states of an execution can grow exponentially with respect to the number of program statements executed. A set of n concurrent events forms 2^n states. As a result, deciding whether a distributed computation satisfies a global predicate (property) can incur significant overhead. In this chapter we will present the theoretical background necessary for the proper understanding of the following chapters and we will discuss several techniques introduced by different researchers to handle the state space explosion problem.

The rest of this chapter is organized as follows: Section 2 presents a model of a distributed computation. Section 3 presents the two main approaches used to verify distributed programs. Section 4 explores some of the logics used to formally specify the properties that a programmer may need to check. Section 5 briefly describes the difficulty of observing global properties of distributed programs. Sections 6 to 10 are dedicated to review a variety of strategies for ameliorating the state explosion problem that have been explored in the literature. These strategies include: exploiting the structure of the property, atomicity, program slicing, partial order reduction and exploiting partial order semantics.

2.2 A Distributed Computation Model

One of the important issues in reasoning about a distributed program is the model used for a distributed computation. In this section, we will present a distributed system model and a model for capturing the behavior of a distributed program.

2.2.1 A Distributed System Model

We assume a loosely-coupled message-passing distributed program without any shared memory or global clock. It consists of n processes denoted by $P_1, P_2, \ldots, P_n$ and a set of unidirectional channels. A channel connects two processes. The delay incurred in sending a message through a channel is arbitrary but finite. The state of a channel at any point is defined to be the sequence of messages sent along that channel but not received.

2.2.2 The Happened-before Model of a Distributed Computation

An ***event*** marks the execution of a statement. It can be an internal computational event or an external message event. Lamport [Lam78] has argued that, in a true distributed system, events can only be partially ordered. Events are related by either their execution order in a process or message send/receive relations across processes. The traditional happened-before relation ($\rightarrow$) between events applies to all events executed [Lam78].

A ***run*** of a distributed program is an event structure $\langle E, \rightarrow \rangle$ formed by the set of events executed (E) and the happened-before relation ($\rightarrow$) among these events.

A run of a distributed program can be viewed in terms of a two dimensional space time diagram. Space is represented in the vertical direction and time in the horizontal direction. Circles are used to depict events. Transmission of a message is represented by a directed edge linking the send event with the corresponding receive event. The space time diagram shown in Figure 1 depicts a distributed computation involving three processes.

In a space time diagram, the events within a single process are totally ordered. $a \rightarrow b$ if and only if there is a directed path from event a to event b. The happened before relation is only a partial order on the set of events. Thus two events a and b may not be related by the happened before relation, in this case we say that a and b are ***concurrent*** (denoted by $a \parallel b$). For example, in Figure 1, $a \parallel e$ because $\neg(a \rightarrow e)$ and $\neg(e \rightarrow a)$.

A ***consistent cut*** C of an event structure $\langle E, \rightarrow \rangle$ is a finite subset $C \subseteq E$ such that if $e \in C \ \wedge \ e' \rightarrow e$ then $e' \in C$.

For each consistent cut C, there is a corresponding ***global state*** of the program represented by the values of the program variables and channels states attained upon completion of the events in C. For simplicity of presentation, we do not distinguish between a consistent cut and the global state associated with it.

The set of global states of a given distributed computation endowed with set union and intersection operations form a distributive lattice, referred to as the ***state lattice*** [Mat89]. Based on this state lattice, one can check if the run satisfies the required properties in program testing and debugging.

Figure 2 shows the state lattice L corresponding to the event structure shown in Figure 1. Each state is identified by the most recent event executed in each process. For example, $\langle a, c, f \rangle$ is the state reached after executing event a in P_1, event c in P_2 and event f in P_3. Only a subset of the events in Figure 1 and a subset of the states in Figure 2 are labeled for future reference.

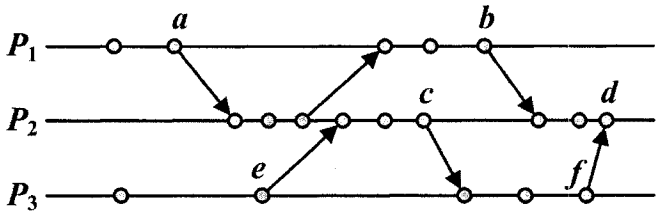

Figure 1: An example of an event structure that represents a run of a distributed program consisting of three processes.

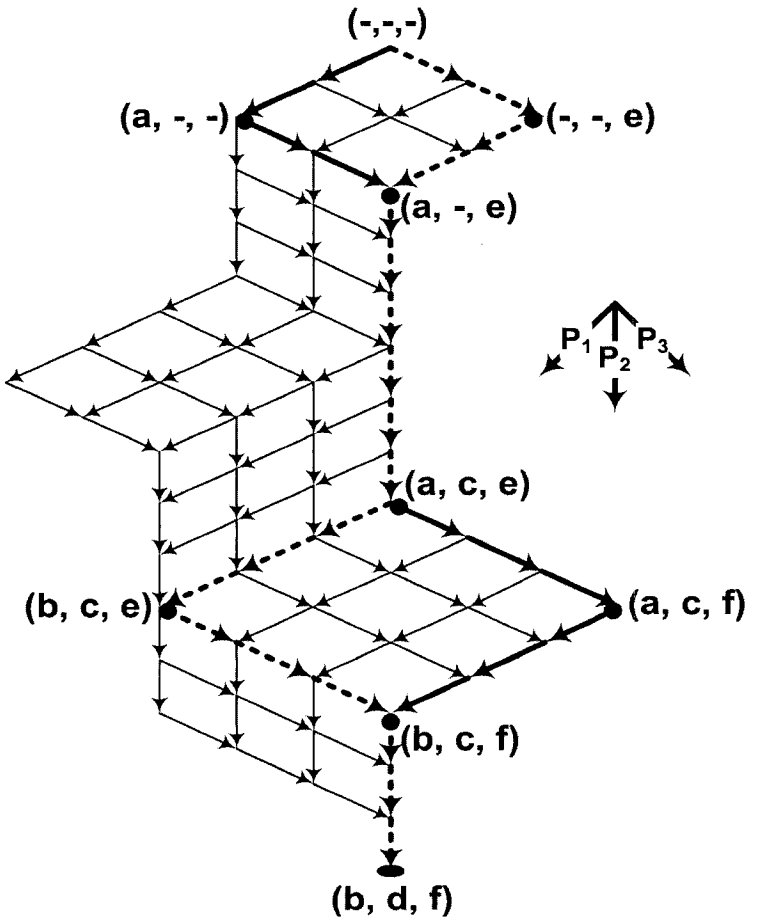

Figure 2: The state lattice (L) associated with the event structure shown in Figure 1. The bold and dotted paths represent the set of all atomic state paths.

2.3 Taxonomy of Approaches for Debugging Distributed Systems

2.3.1 Static Analysis

Static analysis approaches analyze the source code of a distributed program without the use of any runtime information, and assume that all of the execution paths of the program are possible [Lad01].

Some of the static analysis methods are based on traversing all the states that the program may enter. These methods have, in general, exponential time and space complexity.

Other static analysis methods use formal proof techniques. The main advantage of these techniques is that they are accurate. However, their main disadvantages are:

1. They are expensive.
2. Formal proof techniques cannot be fully automated.
3. The proof must be repeated for each change in the program.

2.3.2 Dynamic Analysis

Dynamic analysis techniques collect information during the execution of the distributed program and analyze it. These methods instrument the program to collect the required information during the execution [Lad01].

There are two main types of dynamic analysis:

1. On-the-fly: The program behavior is analyzed during its execution such that errors are detected and reported as they occur during the run.

2. Offline: These techniques collect a log of events that occur during a program's execution and post-process the log to detect errors. The main disadvantage of these techniques is that execution logs can be very large for parallel and distributed programs.

2.4 Specifying Properties of Distributed Systems

2.4.1 Propositional Logic and First Order Logic

Propositional and first order logics can be used to express properties of states [AP98]. Each formula represents a set of states that satisfy it. These formulas are considered static in the sense that they represent a collection of states, but not the dynamic evolution among them during program execution.

Modal logics [HC96] extend propositional and first order logics by allowing the description of the relation between different states during the execution. This is more suitable for specifying properties of distributed systems, where we are not only interested in the relation between the values at the beginning and at the end of the execution, but also other properties related to the sequence of states during the execution. In the following subsection we will give a brief introduction to temporal logic which is based on modal logics and has been widely used to specify properties of distributed programs.

2.4.2 Temporal Logic

Temporal logics are based on modal logics [Eme90, Pnu79]. Modal logics were introduced to study situations where the truth of a statement depends on its mode. For example, an assertion P may be false now, but given a mode future, future P

could be true. Pnueli suggested using temporal logics, in 1977, for formally specifying behavioral properties of systems [Pnu77, Pnu79]. This view has been confirmed by the extensive use of temporal logics in the last two decades.

Temporal logic formulas consist of the usual atomic propositional logic formulas (non-temporal part), plus temporal operators. The non-temporal part specifies the basic properties of states. The temporal operators specify temporal properties. Non-temporal properties can be checked on each state based on the values of the variables in that state, whereas verifying temporal properties requires the investigation of state paths.

We will use the term atomic properties to refer to the basic properties of states (non-temporal part). Given a transition system $T = (S, I, R)$ where S is a set of states, $I \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a transition relation, we define AP to be the set of atomic properties of the system. We assume that we have a procedure $L : S \times AP \rightarrow \{true, false\}$ such that given a property $p \in AP$ and a state $s \in S$, procedure L can decide if $s \models p$, i.e., $s \models p \; iff \; L(s,p) = true$.

Various temporal logics have been defined in the literature. LTL, CTL and CTL^* are the most commonly used temporal logics. All these logics use four temporal operators $\mathbf{G}, \mathbf{F}, \mathbf{U}, \mathbf{X}$. The meaning of these operators can be described as follows:

$\mathbf{G}p$: Globally p, i.e., assertion p holds in every state.

$\mathbf{F}p$: Future p, i.e., assertion p will hold in a future state.

$p\mathbf{U}q$: p Until q, i.e., assertion p will hold until q holds.

$\mathbf{X}p$: Next p, i.e., assertion p will hold in the next state.

In the following subsections we will give some details about LTL, CTL and CTL^*.

2.4.3 Linear Temporal Logic (LTL)

Linear time temporal logic (LTL) is used to specify properties of interleaving sequences (i.e., paths). LTL uses the four basic temporal operators **G, F, U, X**. The syntax of LTL is defined by the following grammar where terminals $\neg, \wedge,$ and $\vee$ are logical connectives; terminals **X**, and **U** are temporal operators; terminal ap denotes an atomic property, ($ap \in AP$); and nonterminal f denotes an LTL formula:

$$f ::= ap | f \wedge f | f \vee f | \neg f | \mathbf{X} f | f \mathbf{U} f$$

We have the following equivalences for LTL operators,

$\mathbf{F}p \equiv true\mathbf{U}p$

$\mathbf{G}p \equiv \neg(true\mathbf{U}\neg p)$

Based on the above equivalences, any LTL property can be expressed using only symbols from the set $\{\neg, \vee, \mathbf{X}, \mathbf{U}\} \cup AP$.

More details about LTL can be found in [Pnu79].

2.4.4 Computation Tree Logic (CTL)

Computation tree logic CTL is a branching time logic; meaning that the computation starting from a state is viewed as a tree where each branch corresponds to a path [Eme90]. The truth of a temporal formula in CTL is defined on states. A temporal formula is true for a state if all the paths or some of the paths originating from that state satisfy a condition.

CTL temporal operators are pairs of symbols. They talk about what can happen

from the current state. The first member of the pair is one of the path quantifiers **A** or **E**. Path quantifier **A** means "for all paths", whereas **E** means "there exists a path". The second member of the pair is one of the basic temporal operators $\mathbf{G}, \mathbf{F}, \mathbf{U}, \mathbf{X}$.

The syntax of CTL is defined by the following grammar, where terminals $\neg, \wedge,$ and $\vee$ are logical connectives; terminals $\mathbf{EX}, \mathbf{EU}$ and $\mathbf{AU}$ are temporal operators; terminal ap denotes an atomic property, $(ap \in AP)$; and non-terminal f denotes a CTL formula:

$$f ::= ap \mid f \wedge f \mid f \vee f \mid \neg f \mid \mathbf{EX} f \mid f\ \mathbf{EU}\ f \mid f\ \mathbf{AU}\ f$$

We have the following equivalences for CTL operators,

$\mathbf{EF}p \equiv true\ \mathbf{EU}\ p$

$\mathbf{EG}p \equiv \neg(true\ \mathbf{AU} \neg p)$

$\mathbf{AF}p \equiv true\ \mathbf{AU}\ p$

$\mathbf{AG}p \equiv \neg(true\ \mathbf{EU} \neg p)$

$\mathbf{AX}p \equiv \neg\mathbf{EX}\neg p$

The semantics of CTL is defined in Table 1. Based on the rules given in this table, we can decide if a state in a transition system satisfies a CTL formula. The semantics of the temporal operators $\mathbf{EF}, \mathbf{AF}, \mathbf{EG}$ and $\mathbf{AG}$ follow from the equivalences given above and the rules given in Table 1.

Table 1: CTL Semantics.

$s \models p$	iff	$L(s,p) = true$ where $p \in AP$
$s \models \neg p$	iff	not $s \models p$
$s \models p \wedge q$	iff	$s \models p$ and $s \models q$
$s \models p \vee q$	iff	$s \models p$ or $s \models q$
$s_0 \models EX\ p$	iff	there exists a path $(s_0, s_1, s_2, ...)$, such that $s_1 \models p$.
$s_0 \models AX\ p$	iff	for all paths $(s_0, s_1, s_2, ...)$, $s_1 \models p$
$s_0 \models p\ EU\ q$	iff	there exists a path $(s_0, s_1, s_2, ...)$, such that there exists an i, $s_i \models q$ and for all $j < i, s_j \models p$.
$s_0 \models p\ AU\ q$	iff	for all paths $(s_0, s_1, s_2, ...)$, there exists an i, $s_i \models q$ and for all $j < i, s_j \models p$.

2.4.5 CTL^*

CTL^* combines LTL and CTL into a single framework. There are two types of formulas in CTL^*: path formulas and state formulas. A path formula is true or false for paths (all LTL formulas are path formulas). A state formula is true or false for states (all CTL formulas are state formulas). The truth set of a state formula is a set of states, whereas the truth set of a path formula is a set of paths. All the formulas in AP are state formulas. Every state formula can be considered as a path formula with the interpretation that the truth of the formula depends only on the first state of the path.

The formulas $\mathbf{G}p, \mathbf{F}p, p\mathbf{U}q$, and $\mathbf{X}p$ are all path formulas. The path quantifiers **A** and **E** can be used to generate state formulas from path formulas. Given a path formula p, $\mathbf{A}p$ and $\mathbf{E}p$ are two state formulas. $\mathbf{A}p$ is true in a state, if and only if, for all paths originating from that state, p is true. $\mathbf{E}p$ is true on a state, if and only if, there exists a path originating from that state where p is true. For example, given an atomic property p, $\mathbf{F}p$ will be true for a path if there exists a state on the path

that satisfies p. More details about CTL^* can be found in [BBF+01].

2.5 Problems in Checking Global Properties

Many of the problems in distributed programs can be reduced to the problem of observing a distributed computation to check whether it satisfies a given property or not. Termination detection and deadlock detection are some examples [Gar02].

The difficulties of observing a distributed computation to check the satisfaction of a given property are due to the following characteristics of a distributed program [Gar02]:

1. The lack of a common clock, which implies that the events in a given distributed computation can only be partially ordered.

2. The lack of shared memory, which indicates that the evaluation of a global property will incur message overhead to collect the necessary information to evaluate it.

3. The existence of multiple processes that are running concurrently, which implies that the number of global states that needs to be considered in property checking will be exponential in the number of processes.

In general, the problem of detecting a global property in a given distributed program computation is NP-complete [CG98]. The rest of this chapter will be mainly dedicated to explore some of the work presented in the literature to efficiently check the satisfaction of a property in a given distributed computation.

2.6 Property Checking Approaches

The first approach to check the satisfaction of a global property in a distributed computation is based on the global snapshot algorithm by Chandy and Lamport [CL85, Bou87, SK86]. Their approach requires repeated computation of consistent global snapshots until a snapshot is found in which the desired property (predicate) is satisfied. This approach works only for stable properties, i.e., properties that do not turn false once they become true. The Chandy and Lamport approach may fail to detect a non-stable property because it may turn true only between two successive snapshots.

The second approach to check the satisfaction of a global property in a given distributed computation was proposed by Cooper and Marzullo [CM91]. Their approach is based on the construction of the state lattice [CM91, JMN95] and can be used for the detection of $possibly : \phi$ and $definitely : \phi$. The predicate $possibly : \phi$ is true if ϕ is true for any global state in the lattice. The predicate $definitely : \phi$ is true if, for all paths from the initial global state to the final global state, ϕ is true in some global state along that path. Though this approach can be used to detect both stable and unstable properties, the detection may be prohibitively expensive. This approach requires exploring $O(m^n)$ global states in the worst case, where n is the number of processes and m is the number of relevant local states in each process.

The third approach avoids the construction of the state lattice by exploiting the structure of the property to identify a subset of the global states such that if the property is true, it must be true in one of the states from this subset. This approach can be used to develop more efficient algorithms, but it is less general than the second approach. For example, [GW94, GW96] present algorithms of complexity $O(n^2m)$ to detect $possibly : \phi$ and $definitely : \phi$ when ϕ is a conjunction of local properties

(a local property is a property of a single process). [CG95, TG93] present efficient algorithms that use this approach to detect $\Sigma x_i < C$ where the x_i are variables on different processes and C is constant.

In [SS95], a hybrid of the second and third approaches has been proposed to reduce the size of the lattice that must be explored during detection.

It has been shown that the detection of a general property in a given distributed computation is intractable [CG98]. Consequently, the class of the property must be restricted to allow for efficient detection. In the following subsection we will give a brief description of the classes of properties (predicates) that can be checked in a run efficiently.

2.6.1 Property Classes

In this section we will describe four classes of properties for which efficient detection algorithms have been developed.

2.6.1.1 Stable Properties

A stable property remains true once it becomes true. Termination detection is an example of stable property detection. Stability depends on the system; some properties are stable in some systems but not stable in others. The global snapshot algorithm by Chandy and Lamport [CL85] can be used to check the satisfaction of a stable property in a given distributed computation.

The value of an unstable property may change from time to time. There is very little value in using a snapshot algorithm to detect a non-stable property—the property may have held even if it is not detected. Many researchers have described algorithms to detect stable and unstable properties [GW96, GW94, SM94, GW92, HW88].

2.6.1.2 Observer Independent Properties

Charron-Bost, *et al.*, [CBDGF95] present a class of properties that they call observer independent. This class includes all properties such that $(Possibly : \phi) \equiv (Definitely : \phi)$. A disjunction of local properties is an observer independent property. Any stable property is also observer independent, the proof of this fact can be found in [CBDGF95].

The name "observer independent" originates from the notion of a set of observers where each observes a different sequential execution of the distributed program. Each observer can determine if ϕ becomes true in any of the global states observed by him. If property ϕ is observer independent, then all observers will agree on whether ϕ ever became true.

Observer independent properties can be detected efficiently because we need to traverse only a single path of the state lattice to detect an observer independent property ϕ. If ϕ is true for any global state in the path, then $Possibly : \phi$ is clearly true. On the other hand, if ϕ is false along that path, we know that $Definitely : \phi$ is also false. Since $(Possibly : \phi) \equiv (Definitely : \phi)$ for observer independent properties, we conclude that $Possibly : \phi$ is also false for the lattice.

2.6.1.3 Linear Properties

A linear property is based on the notion of a "forbidden" state.

Let s be a local state, X and Y be two global states. X^s denotes the cut formed by advancing X to the successor of s. A local state is a forbidden state with respect to a property ϕ if its inclusion in any global state X implies that ϕ is not satisfied at X. This can be defined formally as follows [CG98]:

$$forb_\phi(s, X) \equiv \forall Y : X \subseteq Y : \phi(Y) \Rightarrow X^s \subseteq Y$$

This means that if ϕ is false in X and s is the forbidden state, then ϕ will remain false until a successor to s is reached.

Conjunctive properties are linear. An efficient algorithm to detect conjunctive properties can be found in [Gar02]. Some properties are linear in some systems but not in others. For example, the property $x_1 + x_2 \leq c$, where x_1 and x_2 are two variables belonging to different processes and c is a constant, is linear in systems where x_1 or x_2 is monotonically increasing.

2.6.1.4 Regular Properties

Let L be the state lattice of a distributed computation. A property ϕ is regular if and only if for any two global states G and H,

$$\phi(G) \wedge \phi(H) \Rightarrow \phi(G \cap H) \wedge \phi(G \cup H).$$

That is, a global property is a regular property if the set of global states satisfying the predicate forms a sublattice of the lattice of global states. A regular predicate is also linear and hence easy to detect.

The property $\phi =$ "There is no outstanding message in the channel" is an example of a regular property. This property holds on a global state X if for each send event in X the corresponding receive event is also in X. Suppose ϕ holds on X and Y $(\phi(X) \wedge \phi(Y))$, it is easy to show that it holds in $(X \cup Y)$. To show that ϕ holds in $(X \cap Y)$, consider a send event $s \in (X \cap Y)$. Let r be the corresponding receive

event. $\phi(X)$ implies that $r \in X$ and $\phi(Y)$ implies that $r \in Y$. Thus $r \in (X \cap Y)$. Consequently, $\phi(X \cap Y)$ and hence ϕ is a regular predicate [Gar02].

2.6.2 Temporal Predicate (Property) Detection

A number of algorithms have been developed in the literature to detect predicates for distributed computations under the following temporal operators: *possibly*(**EF**) [CM91, GW94, GM01], *definitely* (**AF**) [CM91, GW96, GM01], *controllable*(**EG**) [TG98, GM01], and *invariant*(**AG**) [CM91, GW94, GM01].

Sen and Garg [SG02] presented solutions to the predicate detection of linear and observer-independent predicates under the **EG** and **AG** operators of CTL. For linear predicates they developed polynomial-time predicate detection algorithms that exploit the structure of finite distributive lattices. For observer-independent predicates they proved that predicate detection is NP-complete under the **EG** operator and *co-*NP-complete under the **AG** operator. They also presented polynomial-time algorithms for a CTL operator called ***until*** (***U***).

Sen and Garg generalized an effective abstraction technique called computation slicing [SG03]. They presented polynomial time algorithms to compute slices with respect to temporal logic predicates from a "regular" subset of CTL, which contains temporal operators **EF**, **EG**, and **AG**. Furthermore, they showed that these slices contain precisely those global states of the original computation that satisfy the predicate. Using temporal predicate slices, they gave an efficient (polynomial in the number of processes) predicate detection algorithm for a subset of CTL that they call regular CTL. A brief description of the slicing concept will be presented in Section 2.8.

2.6.3 Linked Predicates and Atomic Sequences

Predicates can be combined to describe a sequence of events. For example, a user may want to halt a program and examine its state when a specified sequence of events is observed during the execution of the program. Such predicates are called Linked Predicates. In [GW94, GW96], the problem of detecting linked predicates has been discussed.

In [HW88, HPR93], the authors formally defined atomic sequences of predicates and proposed a distributed algorithm to detect their occurrences. These global predicates are defined for message-passing distributed programs only. They describe global properties by causal composition of local predicates augmented with atomicity constraints. Atomicity constraints specify prohibited properties that must remain false during the detection of the sequence. A sequence of local predicates satisfied by a sequence of local states is a valid solution only if it is free of occurrence of forbidden events; for this reason these sequences are called atomic sequences.

2.6.4 Channel Properties

The work on predicate detection presented earlier considers properties specified as Boolean formula of local properties. However, many properties of distributed systems use the state of channels. For example, the property, "there is no token in the system" uses the state of the channels. This property is equivalent to "no process has the token" and "no channel has the token". The state of the channel is defined as the set difference of the send events and the receive events on that channel. A channel predicate is a Boolean function of the state of the channel, which will include all the messages in transit.

In [GCKM97], Garg *et al.* introduced the concept of linear channel predicates and

presented efficient centralized and distributed algorithms to detect any conjunction of local and linear channel predicates. The time complexity of their detection algorithm is $O(mn^2)$, the space complexity is $O(mn^2)$ and the message complexity is $O(mn)$, where n is the number of processes and m is the maximum number of messages sent/received by any application process.

In [MI92], Manabe and Imase proposed a method for detecting predicates, including channel predicates, using a replay approach. Their approach is restricted to channel predicates that can be evaluated by a single process and requires two identical runs.

2.7 Atomicity

In the previous section we have described some of the property classes for which efficient checking algorithms can be found. However, if the property that we want to check does not belong to any of these classes (general property), the checking will incur significant overhead due to the exponential number of states that needs to be considered.

The notion of atomicity can be exploited to reduce the number of states where a given property needs to be checked. In this section we will describe some of the related work in the literature that exploits the notion of atomicity for this purpose.

2.7.1 Lamport's Atomicity Theorem

Lamport has developed a theorem in atomicity to simplify verification of distributed systems [Lam90]. Lamport adopted the common approach of formally defining an execution of a distributed algorithm to be a sequence of atomic actions. At the

lowest level of abstraction each event result from executing any statement in the distributed program can be considered an atomic action. Reducing the number of atomic actions makes reasoning about a concurrent program easier because there are fewer interleavings to consider. This is the main goal of Lamport's theorem.

According to this theorem, a sequence of statements in a distributed program can be grouped together and be treated as an atom under some stated conditions. Informally, an atom may receive information from other processes, followed by at most one externally visible event (for instance, altering a variable relevant to some global property), before sending information to other processes. This theorem allows a distributed program to be abstracted into a reduced distributed program with more general and possibly larger atoms. As a result, the cost of program verification can also be reduced.

Formally, consider a distributed algorithm A in which each process executes a sequence of non-atomic operations. Each operation removes a set of messages from the process's input buffers, performs some computation, and sends a set of messages to other processes. According to this theorem, the reduced version A' of algorithm A is one in which an entire operation is a single atomic action and message transmission is instantaneous; this means that a message appears in the receiver's input buffer when the message is sent. In this case, algorithm A' will not have any computation state in which a process is in the middle of an operation or a message is in transit. Hence, algorithm A' is simpler than algorithm A and it is easier to reason about A' than about A. In his work, Lamport defined six conditions and proved that if these conditions are satisfied, then A satisfies a correctness property ϕ if and only if A' satisfies ϕ.

2.7.2 Atomicity in Multithreaded Programs

Ensuring the correctness of multithreaded programs is hard due to the potential for unexpected and nondeterministic interactions between threads. This motivated the development of numerous static and dynamic analysis techniques for detecting race conditions, a situation where two threads simultaneously access the same data variable, and at least one of the accesses is a write [SAWS05, NM92, Net91]. However, the absence of race conditions does not guarantee the absence of errors due to unexpected thread interactions. Consequently, there has to be a property to ensure the correctness of a program at a higher level of granularity, namely the atomicity of code blocks [FF04]. If a code block is atomic, we can safely reason about the program's behavior at a higher level of granularity, where the atomic block is executed in one step, even though the scheduler is free to interleave threads at instruction level granularity.

An atomicity violation occurs if the effect of a code block depends on the execution order of concurrent threads that use a shared data structure. The atomicity of a code block can be violated even if the data structure is protected through synchronization and individual accesses do not create a data race [PG04].

The following are two examples of methods that are data race free but not atomic [WS06a, WS06b].

Example 1:[WS06b]

Consider a graphics package with a Rectangle class that offers synchronized methods $setX(int\ x)$ and $setY(int\ y)$ and an unsynchronized method $setXY(int\ x, int\ y)$ whose body is simply $\{setX(x); setY(y);\}$.

Assume that there are two threads that share two variables x and y and that run concurrently. Thread 1 calls $setXY(100, 100)$ and thread 2 calls $setXY(200, 200)$. The execution of the two threads can interleave in the following way:

```
public class Vector extends ... implements ... {
   public Vector(Collection c) {
         // c is v1, elementCount is the field of v2.
1        elementCount = c.size();
2        elementData = new Object[(int)Math.min((elementCount*110L)/100,Integer.MAX_VALUE)];
3        c.toArray(elementData);
   }
   public synchronized int size() { return elementCount; }
   public synchronized Object[ ] toArray(Object a[ ]) {
      if (a.length < elementCount) // i.e. v2.length < v1.elementCount
         // this branch will be taken if v1.add is executed.
         a = (Object[ ])java.lang.reflect.Array.newInstance(
                     a.getClass().getComponentType(), elementCount);
      System.arraycopy(elementData, 0, a, 0, elementCount);
      if (a.length > elementCount)
         a[elementCount] = null;
      return a;
 }
  public synchronized void removeAllElements() { ... }
  public synchronized boolean add(Object o) { ... }
}


thread_1                          thread_2
Vector v2 = new Vector(v1);         v1.removeAllElements();
```

Figure 3: Part of the implementation of Vector in Sun JDK 1.4.2.

$thread1.setX(100)$; $thread2.setX(200)$; $thread2.setY(200)$; $thread1.setY(100)$;

The result is $x = 200, y = 100$. This is different from the result of any serial execution of the two calls. As a result, we can conclude that the concurrent invocations of $setXY$ are not atomic, even though this class is data race free [WS06b].

Example 2:

The following example (taken from [WS06a]) shows that the constructor of *java.util.Vector* in Sun JDK 1.4.2 violates atomicity.

Figure 3 shows part of the implementation of Vector in Sun JDK 1.4.2. Consider the following execution of the program at the bottom of the figure: thread 1 calls the constructor for vector to construct a new vector v_2 from another vector v_1 with k elements. Thread 1 yields execution to thread 2 immediately after statement **1** in the Vector constructor before the constructor completes. Thread 2 removes all elements of v_1, and then thread 1 resumes execution at statement **2**. The result of

this execution is that v_2 has k elements, all of which are null because the elementData array of v_2 is allocated according to the previous size of v_1 [WS06a].

2.7.2.1 Static Detection of Atomicity Violations

C. von Praun and T. Gross have developed a static analysis technique that infers atomicity constraints and identifies potential violations [PG04]. Their technique is based on an abstract model of threads and data. A symbolic execution is used to track object locking and access and to provide information that can be exploited to determine potential violations of atomicity.

Their algorithm is neither sound, i.e., there can be underreporting, nor complete, i.e., there can be overreporting. However, although the algorithm does not guarantee to find all violations of atomicity, experimental results show that this method is efficient and effective in determining several synchronization problems in a set of application programs and the Java library.

2.7.2.2 Dynamic Detection of Atomicity Violations

Flanagan and Freund have developed a dynamic atomicity checker tool for multi-threaded programs called Atomizer [FF04]. Their detection strategy is based on Lipton's reduction theorem [Lip75]. Wang and Stoller [WS06a, WS06b] also developed a dynamic reduction based atomicity violation detection algorithm, which classifies events based on commutativity and applies Lipton's reduction theorem.

In addition to the reduction based algorithms, Wang and Stoller [WS06a, WS06b] developed a block based atomicity violation detection algorithm. This algorithm decomposes the problem of checking atomicity of a set of transactions into a set of smaller problems, each of which requires checking atomicity of two blocks. This

algorithm is more complicated and more expensive but significantly more precise than the reduction based algorithm.

One way to combine the two algorithms is to first run the reduction based algorithm. If it indicates possible violations of atomicity, then run the block-based algorithm, which is more expensive and more precise. If it also indicates possible atomicity violations, then report the possible violations to the user.

2.8 Program Slicing

The concept of slicing arose from research on dataflow analysis and static program analysis [Wei84]. The main goal of introducing this concept was to facilitate the debugging and understanding of sequential imperative programs. A slice of a program P with respect to a criterion C (usually a program point) is a set of statements of P, which are relevant to the computations performed in C. It has been extended in various ways to deal with more complex program constructs, e.g., arrays, pointers, and concurrency. Moreover, the concept of slicing has been extended to more modern formalisms, including Z specifications [WY04], and hierarchical state machines [HW97].

In general, the basic idea behind program slicing is to remove details of the program that are not relevant to the analysis in hand. For example, if we wish to verify some property, slicing can be applied to eliminate parts of the program that do not affect that property.

Computation slicing was introduced in [LRG04, SG03, GM01, MG01] as an abstraction technique for analyzing distributed computations (finite execution traces). A computation slice, defined with respect to a global property, is the computation

with the least number of global states that contains all global states of the original computation for which the property evaluates to true. This is in contrast to traditional slicing techniques, which either work at the program level or do slicing with respect to variables. Computation slicing can be used to eliminate the irrelevant global states of the original computation, and keep only the states that are relevant for our purpose. In [MG01], Mittal and Garg proved that a slice exists for all global predicates. However, it is, in general, NP-complete to compute the slice. They developed efficient algorithms to compute slices for special classes of predicates.

2.9 Partial Order Reduction

Partial order reduction [God96, Pel96, Val91] is a technique to reduce the size of the state space to be searched by a model checking algorithm by addressing a specific reason behind the state space explosion, namely the existence of many potentially equivalent execution traces. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. These techniques assume an interleaving representation of partial orders instead of working with direct representations of partial orders.

Partial order reduction techniques analyze systems that are modeled as state transition graphs. Let S be the set of system states. A transition is identified with a particular action that the system can execute and is given by a relation $\alpha \subseteq S \times S$, which defines the pairs of states between which the action can be executed. A state transition graph is a tuple $M = (S, S_0, T, L)$, where $S_0 \subseteq S$ is a set of initial states, T is a set of transitions $\alpha \subseteq S \times S$, and $L : S \rightarrow 2^{AP}$ is a labeling function that assigns to each state a subset of the set AP of atomic propositions. A transition is enabled in a state s if there exists a state s' such that $(s, s') \in T$.

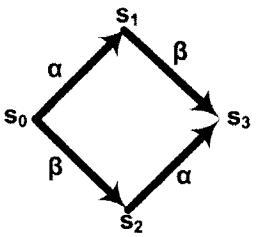

Figure 4: Execution of independent transitions.

Partial order reduction is performed by modifying the DFS (Depth First Search) used to construct the state graph. The search starts with the initial state and proceeds recursively. For each state s, it selects only a subset (called $ample(s)$) of the enabled transitions $enabled(s)$ rather than the full set of enabled transitions as in the full state space construction. The DFS explores only successors generated by these transitions.

Partial order reduction exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders, as depicted in Figure 4. The commutativity condition proposes a potential reduction to the state graph, for it does not matter whether α is executed before β or vise versa in order to reach the state s_3 from s_0. Consequently, only one of the above two paths, needs to be considered in model checking. However, the commutativity property might not be enough to conduct this reduction because of the following reasons: ($\boldsymbol{i}$) the checked property might be sensitive to the choice between the states s_1 and s_2, not only the states s_0 and s_3 ($\boldsymbol{ii}$) the states s_1 and s_2 may have other successors in addition to s_3, which may not be explored if either is eliminated. Consequently, in order to be able to ignore one of the two paths in the above figure they must satisfy an additional condition, namely, they must be stuttering equivalent.

Two paths are stuttering equivalent when they can be decomposed into infinitely many blocks, such that the states in the k^{th} block of one are labeled the same as

the states in the k^{th} block of the other. The ample sets will be used by the DFS algorithm to construct the reduced state graph so that for every path not considered by the DFS algorithm there is a stuttering equivalent path that is considered.

The procedure that calculates, at each state s, a suitable set $ample(s)$ of transitions to be explored is the key to apply partial order reduction efficiently. On one hand, the ample set should be significantly smaller than the set of all enabled transitions in order to effectively reduce the size of the state space. On the other hand, the ample set must include in the reduced state graph at least one equivalent execution sequence for each execution of the original model in order to preserve the correctness of the verification result. Finally, the overhead for computing an ample set must be reasonably small such that the verification time is not increased compared to full state space search [God96, Pel96, Val91].

2.10 Partial Order Semantics versus Interleaving Semantics

The interleaving model (the representation of concurrency by non-determinism) has traditionally been a popular model for describing the possible behaviors of a concurrent program. This model has been successful, both because it is generally easy to formalize and because many interesting properties of a concurrent program can be expressed and analyzed using interleaved sequences [Bes90].

The state-explosion problem is due, among other causes, to the use of the interleaving model of concurrency, or, more precisely, to the exploration of all possible interleavings of concurrent events. For example, the execution of n concurrent events is analyzed by exploring all $n!$ interleavings of these events [God96].

Interleaving semantics ignore real asynchronous behaviors that actually exist: $a||b$ where a and b are atomic is represented by the same transition system as the non-deterministic choice a then b or b then a. This fact creates problems due to the fact that interleaving builds a huge number of uninteresting states, and hence the verification cost will be significantly increased [Gou00].

It has been recognized that concurrency is not the same thing as non-determinism. This observation has stimulated a fairly large body of work on "partial-order models" of concurrency [Maz87, Pra86, Lam78].

To deal with the state explosion problem, partial order reduction, described in the previous section, and methods that exploit partial order semantics have been applied. These methods proved to be successful with systems with a high degree of asynchronous parallelism.

The intuition behind partial-order methods is that concurrent executions are really partial orders and that using the set of all interleavings corresponding to such a partial order is an inefficient way of analyzing concurrent executions. Instead, concurrent events should be left unordered since the order of their occurrence is irrelevant. Hence the name "partial order methods". However, partial order reduction methods keep an interleaving representation of partial orders instead of working with direct representations of partial orders. Partial order reduction methods attempt to limit the expansion of each partial-order computation to just one of its interleavings, instead of all of them [God96, Pel96].

Another problem with the interleaving model is that it imposes on the process a global clock, by which all events are timed, but there is no such clock in distributed systems. Different observers may see different temporal orders for the events. As a

result, it is not reasonable to say that they occur in some interleaved order. Non-interleaving models do not project events into a linear timescale. $a||b$ means that there is no information about the order relation between a and b [Gup94].

In the partial order model, causal dependency between events is considered. However, in the interleaving model, only the temporal behavior of the events of a run is observable [MR97].

Researchers supporting the interleaving model rely on the following two points to justify their choice [MR97]

1. Specifications of concurrent systems ignore causal behavior and refer only to the temporal behavior.

2. Interleaving semantics are much simpler than partial order semantics. Many model checking algorithms have been developed based on the interleaving semantics, but much less work has been done starting from partial order semantics because it seemed technically hard to formalize.

The idea of avoiding the cost of modeling concurrency by interleaving in finite-state verification has been discussed in [PL91b, McM92, Esp93]. In [PL91b], a method based on a POMSET grammar description of the system is introduced. Also, in [McM92, Esp93], a verification method that works by unfolding a Petri net description of a concurrent system into a finite acyclic structure has been proposed.

Chapter 3

Using Atoms to Simplify Checking of Distributed Programs

3.1 Introduction

There are different layers of abstraction relevant in considering the correctness of a distributed program. In the lower layer associated with the coding platform, the execution of each statement of a program is an atomic event. In the higher layer, each process of a distributed program executes atomic actions, or atoms in short. Each atom can be a single statement or a sequence of statements. For example, a code block in a program can be identified as an atom. Safety and progress properties of a distributed program can be defined on the basis of this atomicity. These properties are often expressed in temporal logic [Pnu77, Pnu79].

The execution of a distributed program may include millions of events, each corresponding to the execution of a program statement. Due to concurrency, the number of global states of an execution can grow exponentially with respect to the number

of program statements executed. To reduce the cost of monitoring and checking a distributed program run, atomization is an effective abstraction technique that can be used.

The work reported in this chapter considers message-passing distributed programs in which a code block can be identified as an atom based on design time knowledge, similar to the shared memory counter-part. Without requiring Lamport's conditions on atomicity, we wish to check if a run of the program is atomic, i.e., its behavior is equivalent to its behavior when the execution of each atom is instantaneous, without being interfered by events from other atoms. It is our premise that global properties of a distributed program are not asserted over all states, but over states corresponding to the completion of atoms. Hence, a run must be atomic before such properties should be checked.

3.2 Atomicity Model

In this chapter we advocate the use of design-time knowledge in identifying larger pieces of code, called atoms. An atom can be viewed both statically in code space and dynamically in the event space of a run. Statically, an atom is mapped to a code block in the program and implements some operation that should have atomic effect to the global properties of the program. Such global properties are interpreted as the correctness requirements of the program. When viewed dynamically in a run, an atom contains those events corresponding to the execution of the selected statements from the code block.

Figure 5 shows the atomic abstraction of the run shown in Figure 1, the events belong to six atoms, A, B, C, D, E and F. Rectangles are used to depict atoms. For example, the events of process P_1 belong to two atoms, A and B.

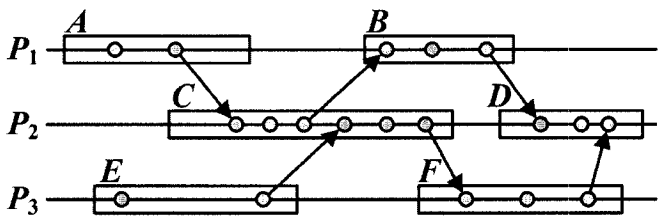

Figure 5: An example of an event structure with atoms, each rectangle represents an atom.

With atoms as the basis in viewing a run, checking the satisfaction of a property in a given distributed program run (runtime verification) involves two steps. First, the atomicity of the run needs to be checked. If the run is not atomic, then there is interference among the atoms associated with the program and debugging should proceed with analyzing the source of this error. This error could arise due to inappropriate synchronization among the processes. Second, if the run is atomic, then the application specific properties of the program can be checked.

Definition 1 *Let $C = \langle E, \rightarrow \rangle$ be a run of a distributed program and L be the corresponding state lattice.*

i. *If there is a path in L in which the events of each atom appear consecutively without interleaving with events of other atoms, then C is an* ***atomic run.***

ii. *The corresponding path in L is called an* ***atomic state path.***

For example, the state lattice in Figure 2 contains a path, shown in dotted lines, in which the atoms are executed in the order $(E; A; C; B; F; D)$. This path traces a sequence of states traversed such that the events of each atom are executed without interleaving with the events of other atoms. From Definition 1, this path is an ***atomic state path.***

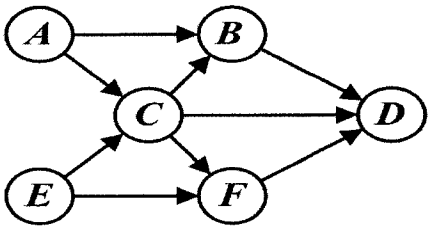

Figure 6: The WG graph associated with the event structure shown in Figure 5.

Definition 2 *Atom A is **weakly ordered** before atom B (denoted by, $A \xrightarrow{w} B$), if*

i. *B is the atom that immediately follows A in the same process, or*

ii. *There exists a message send event in A and the corresponding message receive event is in B.*

For example, in Figure 5, $A \xrightarrow{w} B$ because B is the atom that immediately follows A in process P_1, and $A \xrightarrow{w} C$ because a message was sent in A and received in C.

It is important to observe that this weak-order relation between two atoms does not require all the events in one atom to happen before all the events in the other atom. Trivially, one can deduce that the weak-order relation among atoms is not transitive. For example, in Figure 5, $E \xrightarrow{w} C$ and $C \xrightarrow{w} B$ but not $E \xrightarrow{w} B$.

Definition 3 *The **Weak-order Graph** (WG) of a run is the directed graph $\langle S, E \rangle$, where S is the set of atoms in the run and E is the set of edges corresponding to the weak-order relation $\xrightarrow{w}$ among these atoms.*

Figure 6 shows the WG corresponding to the run shown in Figure 5. This weak-order graph is the basis for understanding the necessary and sufficient conditions for an atomic run and for constructing an efficient algorithm to detect atomicity errors.

A weak-order graph WG can be cyclic or acyclic. It is well-known that the reachability relation among nodes of an acyclic graph is a partial order relation: reachability

is a transitive, reflexive and asymmetric relation in an acyclic graph. For any two atoms A and B, we say that $A \xrightarrow{r} B$ if there is a path from atom A to atom B in WG. An acyclic WG induces the partial order $\langle S, \xrightarrow{r} \rangle$ where $\xrightarrow{r}$ is the reachability relation among atoms in S.

Definition 4 *For an acyclic WG that induces the partial order $\langle S, \xrightarrow{r} \rangle$, the set of* ***linearizations*** *of $\langle S, \xrightarrow{r} \rangle$ is the set of total orders of all atoms in S that is consistent with $\xrightarrow{r}$, i.e., atom A appears before B in a total order only if $\neg(B \xrightarrow{r} A)$.*

There are four possible linearizations of $\langle \{A, B, C, D, E, F\}, \xrightarrow{r} \rangle$ induced by WG shown in Figure 6. These linearizations are $\langle A; E; C; F; B; D \rangle$, $\langle A; E; C; B; F; D \rangle$, $\langle E; A; C; F; B; D \rangle$, and $\langle E; A; C; B; F; D \rangle$ respectively. Each linearization corresponds to an atomic state path in the state lattice shown in Figure 2. For example, the linearization $\langle E; A; C; B; F; D \rangle$ corresponds to the atomic state path shown in dotted lines. The set of all atomic state paths corresponding to the linearizations of $\langle \{A, B, C, D, E, F\}, \xrightarrow{r} \rangle$ is shown in bold, dotted and solid, lines in Figure 2.

Definition 5 *We refer to the state lattice induced by the partial order $\langle S, \xrightarrow{r} \rangle$ as the* ***atomic state lattice (L^*)****. The nodes in the atomic state lattice represent the global states reached by the execution of the atoms.*

Figure 7 shows the atomic state lattice (L^*) associated with the event structure shown in Figure 5. Each node in this lattice represents a state that is identified by the most recent atom executed in each process. For example, $\langle A, C, F \rangle$ is the state reached after executing atom A in P_1, atom C in P_2 and atom F in P_3. In comparison, L^* contains only **8** states and is much smaller than L, which contains **59** states, excluding the initial state.

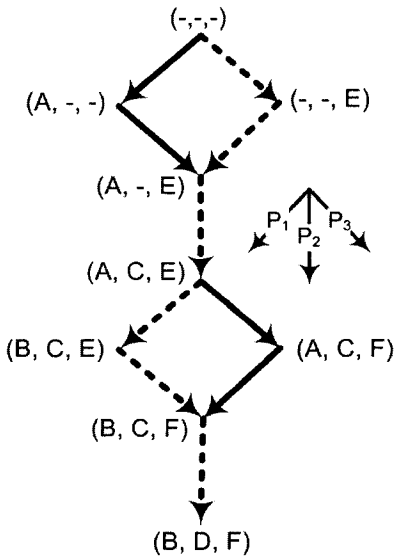

Figure 7: The atomic state lattice (L^*) associated with the event structure shown in Figure 5. The dotted path corresponds to the dotted path in the state lattice (L) shown in Figure 2.

If a run is not atomic, then we infer that some coding error may exist, since every state path in the run contains some atom whose events must be interleaved with some events from other atoms, and hence the program state does not change as if each atom has occurred "instantaneously" without the necessity of something else happening. Figure 8 shows a distributed program run that is not atomic and has a cyclic WG graph.

The absence of atomicity errors indicates that global properties can be checked on the atomic state lattice rather than the much more complex state lattice corresponding to all of the events in a run.

The following lemma establishes an interesting property of an atomic run.

Lemma 1 *A run is atomic iff the corresponding WG is acyclic.*

Proof. (Cyclic $WG \Rightarrow$ the run is not atomic): Suppose otherwise, from the definition of an atomic run, there is an atomic state path in the state lattice. In this path, the

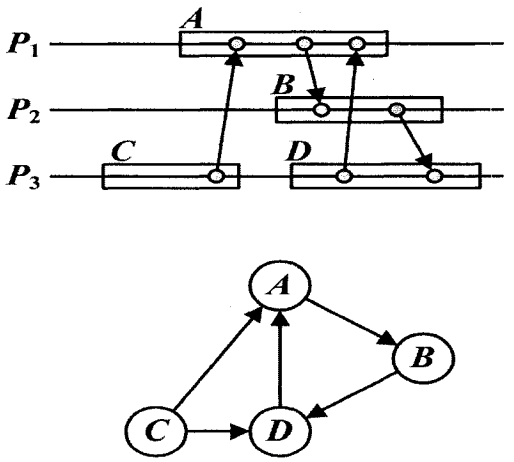

Figure 8: A non-atomic run and the corresponding cyclic WG. Atoms A, B and D form a cycle in WG ($A \xrightarrow{w} B \xrightarrow{w} D \xrightarrow{w} A$).

atoms in a cycle of WG must appear in some serial order, say $\langle A; \ldots; B \rangle$. However, since these atoms form a cycle in WG, we know that $A \xrightarrow{w} \ldots \xrightarrow{w} B \xrightarrow{w} A$. From the definition of the weak-order relation, $B \xrightarrow{w} A$ implies that some event in B happened before some event in A. Consequently, the above atomic state path $\langle A; \ldots; B \rangle$ can not exist in the state lattice.

(Acyclic $WG \Rightarrow$the run is atomic): Since WG is acyclic, there is at least one linearization of $\langle S, \xrightarrow{r} \rangle$. Select an arbitrary linearization. Since a later atom cannot reach an earlier atom in the linearization, the events associated with the atoms can occur according to the ordering of these atoms. Hence, there is a corresponding atomic state path in the state lattice. Consequently, the run is atomic. ■

For each path in the atomic state lattice L^*, there is a corresponding atomic state path in the original state lattice L such that the ordering of atoms is the same in both paths. Conversely, for each atomic state path of the state lattice L, there is a corresponding path in the atomic state lattice L^*. This follows immediately from the fact that for each linearization of the atoms in WG, there is a corresponding path in

both the atomic lattice of $\langle S, \xrightarrow{r} \rangle$ and the lattice of $\langle E, \rightarrow \rangle$. For example, the dotted path in the atomic state lattice shown in Figure 7 corresponds to the dotted path of the original state lattice shown in Figure 2. There are only **6** states traversed in this path in L^*, compared with **19** states in the corresponding atomic state path in L. Notice that for every state in the dotted path of L^* there is a corresponding state in the dotted path of L. For example, the state $\langle B, C, F \rangle$ in the atomic state lattice L^* corresponds to the state $\langle b, c, f \rangle$ in L.

3.3 A Probe Based Atomicity Error Detection Algorithm

In this section, we will describe and prove an on-the-fly distributed algorithm for detecting atomicity errors. We consider a weak-order graph $WG = \langle S, E \rangle$ where S is the set of atoms and E is the set of edges corresponding to the weak-order relation among these atoms. During a run, this graph is constructed distributively as the run unfolds. The associated kernel of each process keeps track of all the necessary information about all of its atoms, and their inbound and outbound edges. For space efficiency, the graph is periodically pruned (distributively) to remove processed information.

For each process P_i, there is an associated vector clock T_i, which is initialized as the zero vector. Let A_i be an atom of process P_i. The global timestamp associated with atom A_i, denoted as $T(A_i)$, is an n-tuple $\langle t_1, t_2, \ldots, t_i, \ldots, t_n \rangle$, where t_i is the local timestamp of atom A_i. Both T_i and $T(A_i)$ are updated as follows:

1. At the start of the execution of an atom A_i, the current value of T_i is updated to $\langle t_1, t_2, \ldots, t_i + 1, \ldots, t_n \rangle$. As a result, the local timestamp of atom A_i is

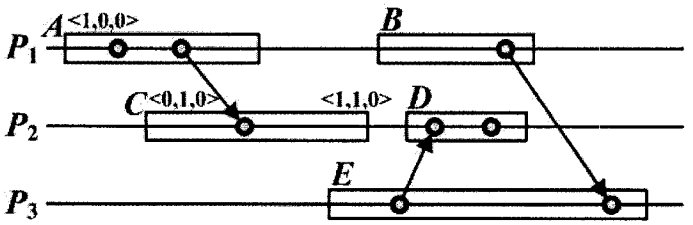

Figure 9: An example illustrating the vector clock initialization and updating policy.

established as $t_i + 1$.

2. At the end of the execution of an atom A_i, the global timestamp $T(A_i)$ is assigned the current value of T_i.
3. Each outgoing (application) message from process P_i is tagged with the current value of T_i.
4. When an application message from process P_j is received in P_i tagged with T_j, update T_i to $max(T_i, T_j)$. The max function is applied to each pair of corresponding elements in the vector timestamps.

Consider the example shown in Figure 9, the vector clock T_i associated with each process P_i is initialized to zero. At the start of the execution of atom A, the current value of T_1 is updated to $\langle 1, 0, 0 \rangle$, and at the start of the execution of atom C the current value of T_2 is updated to $\langle 0, 1, 0 \rangle$. When P_1 sends a message to P_2, the message will be tagged with the current T_1. When P_2 receives the message sent by P_1 it will update T_2 to $max(T_1, T_2)$, and hence T_2 will become $\langle 1, 1, 0 \rangle$. At the end of the execution of atom C, the global timestamp $T(C_2)$ is assigned the current value of T_2 which is $\langle 1, 1, 0 \rangle$.

Let us denote the j^{th} element of the global timestamp of A_i by $t_j(A_i)$. From the previous discussion, we know that $t_i(A_i)$ is the local timestamp of A_i.

Definition 6 *Let A_i and B_j be two atoms in P_i and P_j respectively. Atom A_i knows atom B_j if and only if $t_j(A_i) \geq t_j(B_j)$. A_i knows B_j is analogous to saying that B_j is known to A_i.*

The following property follows trivially from the vector timestamp protocol described above.

Lemma 2 *If A_i knows B_j then there is a path from B_j to A_i in the WG graph.*

Proof. From the vector timestamp, if A_i is not reachable from B_j, then $t_j(A_i) < t_j(B_j)$. Hence A_i does not know B_j and the claim holds. ■

It should be noted that the converse of Lemma 2 is not true, due to the non-transitivity of $\xrightarrow{w}$.

Definition 7 *For any (node) atom A_i in the WG graph, we define*

i. *The set of causal predecessors of A_i, $C_p(A_i) = \{B_j | A_i \text{ knows } B_j\}$.*

ii. *The set of causal successors of A_i, $C_s(A_i) = \{C_k | C_k \text{ knows } A_i\}$.*

iii. *The set of causally unconnected atoms of A_i,*
$C_u(A_i) = \{D_l | D_l \notin C_p(A_i) \wedge D_l \notin C_s(A_i)\}$.

iv. *The set of latest causal predecessors of A_i, $C_{pl}(A_i) = \{E_m | E_m$ is an atom with the largest local timestamp of atoms in P_m that is known to $A_i\}$.*

The following property also follows directly from the previous definitions.

Lemma 3 $B_j \in C_{pl}(A_i) \Leftrightarrow t_j(A_i) = t_j(B_j)$.

Algorithm 1 On-the-fly atomicity error detection algorithm

Goal: For a given atom A_i in the WG graph, to determine whether there is a cycle involving A_i.

Initial step:
for each atom $B_j \in C_{pl}(A_i)$ **do**
  inform B_j about $T(A_i)$;
end for
This step is achieved by broadcasting a probe message containing $T(A_i)$ to all such B_j's. (From Lemma 3, A_i knows $C_{pl}(A_i)$).

for each atom B_j that receives a probe **do**
  if $B_j \in C_s(A_i)$ **then**
    report a cycle; // Reporting step
  else
    for each atom C_j (in P_j) with local timestamp $t_j(C_j)$ such that $t_j(A_i) \leq t_j(C_j) \leq t_j(B_j)$ **do**
      if C_j has received an (application) message from atom D_k such that $D_k \notin C_p(A_i)$ and C_j has not propagated a probe to D_k before **then**
        C_j forwards a probe to D_k; //Diffusion step
      end if
    end for
  end if
end for

Proof. This immediately follows from the definition of $C_{pl}(A_i)$ such that $t_j(B_j)$ is the largest local timestamp of process P_j known to A_i. ■

The abstract algorithm for atomicity error detection through dynamic cycle detection in the WG graph (Algorithm 1) is discussed in the following.

It is assumed that the detection kernel periodically (or when instructed) checks for cycles in the WG graph, and it starts with an atom A_i. We will highlight the algorithm with the following example.

In Figure 10, a partial run involving seven atoms is shown. For simplicity, only the relevant interactions among these seven atoms are shown, without showing other computational events and the interactions among these atoms and other atoms.

From Definition 7, $C_p(A_1) = \{D_3, B_2, C_2, A_1\}$; these atoms are causally known to A_1. $C_s(A_1) = \{E_3, G_5\}$; these atoms causally know A_1. This causal knowledge is maintained by the vector timestamp. On the other hand, $C_u(A_1) = \{F_4\}$; F_4 and A_1 do not causally know each other. Also from Definition 7, $C_{pl}(A_1) = \{A_1, C_2, D_3\}$; specifically C_2 is the most recent atom in process P_2 known to A_1.

The detection algorithm starts with a broadcast of probe messages from A_1 to C_2 and D_3. D_3 forwards the probe to F_4 in the diffusion step. In turn, F_4 forwards the probe to G_5, whereupon G_5 reports a cycle: G_5 knows A_1 and has received a probe emanating from A_1. This reporting corresponds to the sequence of relations among these atoms: $A_1 \xrightarrow{r} G_5 \xrightarrow{w} F_4 \xrightarrow{w} D_3 \xrightarrow{r} A_1$. Notice that the probe does not actually traverse a cycle in WG. In terms of efficiency, a probe never traverses nodes in $C_p(A_i) - C_{pl}(A_i)$ (such as B_2) nor is it forwarded within $C_s(A_i)$.

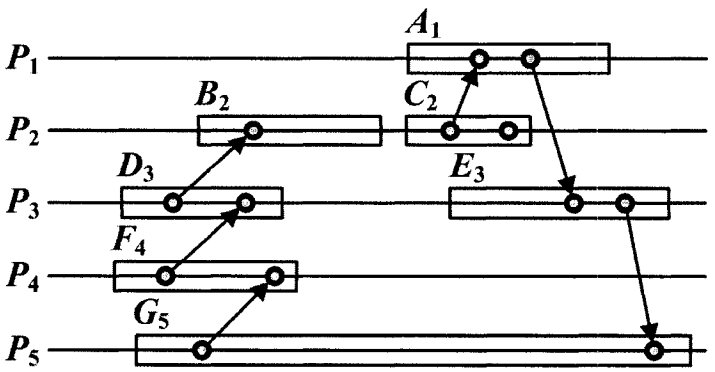

Figure 10: An example to demonstrate the algorithm developed to detect atomicity errors.

Lemma 4 *If atom B_j receives a probe then there is a path from B_j to A_i in the WG graph.*

Proof. Every time a probe is sent from an atom, say C_k, to another atom B_j (by the initial step or the diffusion step), there is a path from B_j to C_k. Since all probes

originate from A_i, the receipt of a probe by B_j implies that there is a path from B_j to A_i. ■

Lemma 5 *If atom B_j receives a probe and $B_j \in C_s(A_i)$, then there is a cycle containing A_i and B_j in the WG graph.*

Proof. From Lemma 4, if atom B_j receives a probe, then there is a path from B_j to A_i. From Lemma 2 and Definition 7, $B_j \in C_s(A_i)$ implies that there is a path from A_i to B_j. Consequently, when both conditions hold, there is a cycle containing both A_i and B_j. ■

The following lemma deals with an important property of a WG graph. For simplicity of presentation, the subscript of an atom that identifies the process is omitted.

Lemma 6 *Every path $A \xrightarrow{w} B' \xrightarrow{w} \ldots \xrightarrow{w} C' \xrightarrow{w} A$ in WG can be rewritten into a target sequence of relations involving both $\xrightarrow{w}$ and $\xrightarrow{r}$ of the form $A \xrightarrow{r} B \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$, such that:*

i. *$B \in C_s(A)$.*

ii. *Each intermediate atom (between B and C) in the target sequence $\in C_u(A)$.*

iii. *$C \in C_{pl}(A)$.*

Proof. The following steps show the construction of the target sequence $A \xrightarrow{r} B \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$ with the required properties.

Let C be the earliest node in the sub-path $B' \xrightarrow{w} \ldots \xrightarrow{w} C'$ of the given path such that $C \xrightarrow{r} A$ and $C \in C_p(A)$. Such a node must exist since C' is such a node if none other exists (since $C' \xrightarrow{w} A$). Remove those nodes after C in the sub-path $C \xrightarrow{w}$

$\ldots \xrightarrow{w} C'$ from the original path to get the sequence $A \xrightarrow{w} B' \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$. We assert that $C \in C_{pl}(A)$. If not, then the node immediately preceding C, say X, in the sequence must be in $C_p(A)$, since X must be known to the node following C in the same process of C, and hence must be known to A. This contradicts the fact that C is the earliest node in the sub-path $B' \xrightarrow{w} \ldots \xrightarrow{w} C'$ such that C is known to A.

Apply a similar elimination of nodes in the sub-path $B' \xrightarrow{w} \ldots \xrightarrow{w} C$. Let B be the latest node in the sub-path such that $A \xrightarrow{r} B$ and $B \in C_s(A)$. This node must exist since B' is such a node if none other exists. Remove those nodes after B' in the sub-path to get target sequence $A \xrightarrow{r} B \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$. ■

In the following lemma, we show that the algorithm will propagate probe messages along the sequence of relations $A \xrightarrow{r} B \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$.

Lemma 7 *Referring to sequence* $A \xrightarrow{r} B \xrightarrow{w} \ldots \xrightarrow{w} C \xrightarrow{r} A$ *in Lemma 6, the algorithm guarantees that atom* B *will ultimately receive the probe broadcast from* A *and report a cycle.*

Proof. From the initial step of the algorithm, A broadcasts a probe to C. In the diffusion step, C forwards the probe to those atoms not in $C_p(A)$ (i.e., $C_u(A) \cup C_s(A)$) but have edges connected to C. The latter continue to forward the probe to its connected nodes within $C_u(A)$ until an atom in $C_s(A)$ is reached. As a result, the cycle in Lemma 6 is guaranteed to be traversed by these probes. ■

Theorem 8 *Algorithm* **1** *reports a cycle if and only if there is a cycle involving atom* A.

Proof. Follows directly from Lemmas 5 and 7. ■

3.3.1 Complexity Analysis

The message complexity of the algorithm can be easily analyzed. A probe is forwarded from atom X to atom Y only if there is a message sent from Y to X. Moreover, a probe is not forwarded from an atom unless the former is in $C_{pl}(A) \cup C_u(A)$. Hence the message complexity is bounded by the number of edges in the sub-graph of WG whose nodes are in the set $C_{pl}(A) \cup C_u(A)$.

3.4 Checking Program Properties

After establishing that a run is atomic, one can proceed to check application-specific properties associated with the run. In this section, we establish the relationship between checking properties on the atomic state lattice L^* and checking the corresponding properties on L, and demonstrate the cost-effectiveness in using atoms.

From Section 3.2, we know that for every state in the atomic state lattice L^* there is a corresponding state in L. For example, the state $\langle A, C, F \rangle$ of L^* in Figure 7 corresponds to the state $\langle a, c, f \rangle$ of L in Figure 2. L^* contains many fewer states than L; in the example, there are 59 states in L but only 8 states in L^*.

In the following, we show that checking some required property φ on L^* is equivalent to checking some transformed property φ' on L. Interestingly, not only that L^* is smaller than L, but also the property φ is simpler than φ' and does not require program augmentation using auxiliary program variables in order to perform the checking.

We first prove a simple result before moving on to more general temporal properties. Suppose φ is an invariant (denoted as $\mathbf{AG}\varphi$ in CTL), expressed as a global predicate defined on the set of distributed variables, and *async* is a global predicate

that is satisfied by every state in which at least one of the processes is not at the end of an atom. For example, in the global state lattice L in Figure 2, all the unlabeled states (altogether 51 of them) satisfy *async* since in each of these states, at least one of the three processes is not at the end of an atom.

Definition 8 *An **atomic state** is a global state in which all processes are at the end of an atom.*

In Figure 2, all the states that are labeled are atomic states, and all the states that are not labeled are non-atomic states. Obviously, every state in an atomic state lattice, such as the one shown in Figure 7, is an atomic state, as stated below.

Lemma 9 *Every atomic state in L is a state in L^*.*

Proof. This follows from the definition of an atomic state. Since L^* contains states reached upon completion of atoms, every atomic state in L must also be a state in L^*. ■

Lemma 10 *$\boldsymbol{AG}\varphi$ holds in L^* iff $\boldsymbol{AG}[async \vee \varphi]$ holds in L.*

Proof. ($\Rightarrow$): If φ holds in L^* then φ holds in all atomic states. Since *async* holds in non-atomic states and from Lemma 9, all atomic states in L are also in L^*, $(async \vee \varphi)$ holds in L.
($\Leftarrow$): If $(async \vee \varphi)$ holds in L, then φ must hold in all atomic states. Hence, φ holds in L^*. ■

In interpreting Lemma 10, a global invariant, such as a general predicate $\varphi \equiv x + y + z = k$, can be asserted as a safety property of a program. We can check the invariant in L^* or L. For checking in L, the invariant becomes $(async \vee \varphi)$.

In instrumentation, the program code of process P_i needs to be augmented with an $async_i$ flag, which is reset if process P_i is not at the end of an atom. Globally, $async = async_1 \vee async_2 \vee \ldots \vee async_n$. In comparison with checking in L^*, checking in L involves checking a more complex predicate $(async \vee \varphi)$ over all states in L, which is also larger than L^*.

Invariants are not enough to represent all required properties. In a distributed program, a common safety property can arise in which whenever a state that satisfies φ is reached, then φ must continue to hold in all next states unless another predicate ψ is satisfied in the latter. This can be expressed as $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$. We will use this property to demonstrate the advantage of checking in L^* against that in L. Through this case study, we show the equivalence of checking properties in L^* and L, and the advantage of using a simpler temporal predicate in L^*.

We start by assuming that using atoms to abstract a program run, the required program properties can be represented by temporal predicates that need to be checked at synchronized states where each process has just finished executing an atom. If we want to check the above global property $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ in the original state lattice L, then the following predicate transformations are applied.

$\varphi' = [(\neg async \wedge \varphi) \vee held\_\varphi]$

$\psi' = (\neg async \wedge \psi)$

Here $held\_\varphi$ is an auxiliary predicate of the program. When the program enters an atomic state, the predicate $held\_\varphi$ is modified to become true or false, depending on whether φ is satisfied at that state or not. The predicate $held\_\varphi$ is unchanged at a non-atomic state. For example, in Figure 2, suppose φ is satisfied in $\langle b, c, e\rangle$ and is not satisfied in $\langle b, c, f\rangle$. The predicate $held\_\varphi$ is set to true at $\langle b, c, e\rangle$ and will remain true in the states that lie between $\langle b, c, e\rangle$ and $\langle b, c, f\rangle$ in the lattice. It will be reset

to false at $\langle b, c, f \rangle$ and will remain false until the state $\langle b, d, f \rangle$ is reached.

This transformation is required because the predicates φ and ψ should not be checked at non-atomic states in the state lattice L. However, the satisfaction of φ in an atomic state leads to a requirement to continue to check for φ and ψ in the next atomic state in an atomic path in L. So *held_*φ is a necessary condition that must be maintained in a sub-path between these two atomic states. This leads to the more complex transformed formula $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$ to be checked in L.

We need to show that checking the temporal predicate $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ in the atomic state lattice L^* is equivalent to checking the more complex formula $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$ on the usually much larger state lattice L. Hence, there is potentially significant saving by using atomicity abstraction in terms of ***(i)*** the size of the state lattice used, and ***(ii)*** simplicity of the formula and its instrumentation.

Lemma 11 *$\boldsymbol{AG}[\varphi \Rightarrow \boldsymbol{AX}(\varphi \vee \psi)]$ holds in L^* iff $\boldsymbol{AG}[\varphi' \Rightarrow \boldsymbol{AX}(\varphi' \vee \psi')]$ holds in L.*

Proof. ($\Rightarrow$): Suppose $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$ does not hold in L. We want to show that $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold in L^*. The assumption asserts that there is a state s in L in which φ' holds while $(\varphi' \vee \psi')$ does not hold in one of its next states, say s'. There are four combinations of s and s':

i. *Both s and s' are atomic states*: Since s is an atomic state, $\neg async$ is true at s. $\varphi' = [(\neg async \wedge \varphi) \vee held\_\varphi]$ holds in s implies φ (and *held_*φ) holds in s. Similarly, $(\varphi' \vee \psi')$ does not hold in s' implies $(\varphi \vee \psi)$ does not hold in s'. Hence, φ holds in s but $(\varphi \vee \psi)$ does not hold in s'. From Lemma 9, s and s' are also states in L^*. Consequently, $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold in L^* as well.

ii. *Both s and s' are non-atomic states*: This case cannot arise as is shown in the

following. Since s is a non-atomic state, *async* is true at s. If φ' holds in s then *held_*φ must hold in s. Since s' is still a non-atomic state, from the definition of *held_*φ, then it must continue to hold in s'. Consequently, this cannot be a case where $(\varphi' \vee \psi')$ does not hold in s'.

iii. *s is an atomic state and s' is a non-atomic state*: This case also cannot arise as is shown in the following. Since φ holds in s and s is an atomic state, *held_*φ will be set and will remain true in s'. Consequently, this also cannot be a case where $(\varphi' \vee \psi')$ does not hold in s'.

iv. *s is a non-atomic state and s' is an atomic state*: This is the non-trivial case. As in case (ii), since φ' holds in non-atomic state s, *held_*φ must hold in s. So there must exist a predecessor atomic state s'' (that can reach s in the lattice L) such that φ holds in s'' (to set *held_*φ). Now we have a situation similar to that in (i) where s'' and s' are both atomic states. φ' holds in s'' implies that φ hold in s'', and $(\varphi' \vee \psi')$ does not hold in s' implies that $(\varphi \vee \psi)$ does not hold in s'. Consequently, there is a state s'' in L^* in which φ holds but $(\varphi \vee \psi)$ does not hold in its next state s'. Hence, $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold in L^* as well.

($\Leftarrow$): Suppose $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold in L^*. So there exists a state s and one of its next states, say s', in L^* such that φ holds in s but $(\varphi \vee \psi)$ does not hold in s'. From Lemma 9, s and s' are also states in L. More specifically, there is a path from state s through non-atomic states $s_1, \ldots, s_k$ in L to reach state s'. Since non-atomic states preserve *held_*φ, we have arrived at a situation where φ' holds in s_k but $(\varphi' \vee \psi')$ does not hold in s'. Hence, $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$ does not hold in L as well. This completes the proof. ■

An important observation is that $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ is a simpler predicate than $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$ in augmenting the program to be checked. In $\mathbf{AG}[\varphi' \Rightarrow \mathbf{AX}(\varphi' \vee \psi')]$, one has to introduce auxiliary variables (not in the original program), which are used to support the setting and resetting of the auxiliary flag $held\_\varphi$ in all atomic states.

In comparison, for checking the property $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$, the instrumentation simply requires recording/detecting a local state at the end of an atom, and later checking $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ on all atomic states in L^*. The checking involves a smaller lattice L^*, as there is no checking required in the non-atomic states. A basic premise is assumed: a distributed program consists of many parts that evolve asynchronously through message passing. Until synchrony is reached by the completion of atoms, there is no real reason to require specific program properties. Of course, errors can occur in the form of improper synchronization among atoms. These errors can lead to atomicity failure (Cyclic WG). When a run is non-atomic, the atomic state lattice is non-existent and this indicates one possible form of program error. If the run is atomic, safety properties such as $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ can be checked on the atomic state lattice L^*.

3.5 Serialization

Section 3.3 discussed run-time monitoring for atomicity errors in a run of a distributed program. Though ensuring atomicity of a run is a must for the existence of an atomic state lattice and is a necessary condition for the correctness of the run, it is not a sufficient condition for correctness. When atoms share a common property, e.g., sharing resources or affecting a shared property or invariant, they must be serialized in a run.

As an example, consider a distributed constraint optimization program that arises in an e-market application where three purchase agents are delegated to perform a coordinated purchases of three different categories of items, say x, y and z. A common budget constraint is applied to their purchases. While optimizing the global return from these purchases, the total budget is also a constraint. The system is designed with the requirement that the purchases are decided at atoms X, Y and Z respectively (corresponding to items x, y and z). For ensuring the global invariant asserted by the common budget, these atoms must be serialized in a run. Error in serialization will imply that at least two purchases are not ordered, leading to a potential violation of the global budget constraint.

In the following discussion, we call such a set of atoms that affect a common property or invariant a conflict set of atoms. Each atom belonging to the conflict set is called a *conflict atom.* All other atoms are called *trivial atoms.*

Definition 9 *The conflict atoms are* ***properly serialized*** *if they lie on a single path in the WG graph. A serialization error occurs if this condition is violated.*

In the following section we will present an on-the-fly serialization violation detection algorithm along with its proof of correctness.

3.6 Serialization Violation Detection Algorithm

Though there can be several conflict sets in a run, for simplicity of discussion we assume the existence of a single conflict set. The following discussion can easily be generalized to cover multiple sets. The detection algorithm dynamically monitors if conflict atoms lie on a single path during the run. To achieve this, each conflict atom is dynamically assigned a sequence number that identifies its position in such a path.

Existence of more than one path would imply that two or more conflict atoms may have an identical sequence number. The serialization error is detected under three scenarios discussed in the proof of Theorem 13.

In the detailed algorithm discussed below (Algorithm 2), each atom A is labeled with a 2-tuple, $label(A) = \langle owner, seq\_no \rangle$. In examining an acyclic WG graph, each atom can be labeled with an integer, referred to as its *seq_no*, that identifies the maximum number of conflict atoms lying on a same path leading to A. For example, consider the labeled WG graph in Figure 11. The conflict atoms are the shaded atoms $\{B, C, E, G, X, Y\}$. The *seq_no* of atom C is 3, indicating that the maximum number of conflict atoms lying on a path ending at C is equal to 3. The owner denotes the most recent conflict atom lying on that maximum path. Hence $label(C) = \langle C, 3 \rangle$. For a trivial atom, such as F, $label(F) = \langle B, 2 \rangle$ indicating that the maximum number of conflict atoms lying on a path ending at F is 2 and the most recent conflict atom on that path is B.

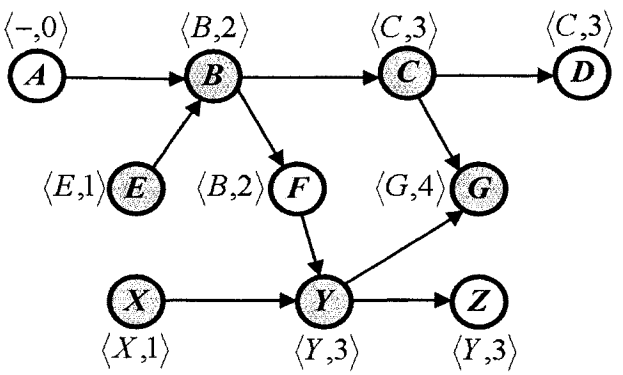

Figure 11: An example of a labeled WG.

Atom A is an *immediate predecessor* of atom B if $A \xrightarrow{w} B$; conversely B is the *immediate successor* of A. Conflict atom A is the *parent* of conflict atom B if B is reachable from A in the WG graph and *seq_no* of B = (*seq_no* of A) + 1; conversely,

B is a *son* of A. A conflict atom with no son is a *terminal* conflict atom. Similarly a conflict atom with no parent is an *initial* conflict atom. For example, in Figure 11, B is the parent of both C and Y, E is an initial atom and G is a terminal atom, X is both an initial and a terminal atom.

An atom, say A, knows that it does not have predecessor atoms if it is the first atom in the process and no message is received in it. An atom knows its immediate successors since it is assumed that the kernel tracks the processes to which it has sent messages from within the atom.

As an illustration of the algorithm, referring to Figure 11, atom B sends an explicit parent message $\langle E \rangle$ to atom E, after computing $label(B) = \langle B, 2 \rangle$. Once E receives this message, it knows that it has a son. Atom Y sends a parent message $\langle B \rangle$ to atom B, but it does not send a parent message to atom X (because *seq_no* of $Y \neq$ (*seq_no* of X) + 1). As a result, X becomes a terminal atom. Atom G receives messages $\langle C, 3 \rangle$ and $\langle Y, 3 \rangle$ from its immediate predecessors. As a result, it reports an error (error scenario 1). In this example there are two terminal atoms, G and X. Hence, an error is also reported upon termination (error scenario 3).

Lemma 12 *Algorithm* **2** *computes* $label(A)$ *correctly.*

Proof. We need to prove that the algorithm correctly computes $label(A) = \langle owner,$ *seq_no*$\rangle$ such that the maximum number of conflict atoms on a path leading to A is equal to *seq_no* and the most recent conflict atom on that path is *owner*. This can be proved using an induction on the value of *seq_no*.

Base case: (*seq_no* $= 1$) : According to the initialization step of the algorithm, if A is an initial conflict atom then it is labeled as: $label(A) = \langle A, 1 \rangle$.

Induction hypothesis: Suppose that the lemma holds for all atoms with *seq_no* $=$ k.

Algorithm 2 On-the-fly serialization error detection algorithm

Initialization step:
For each atom A without predecessor, $label(A)$ is initialized as follows:
if A is a conflict atom **then**
  $label(A) = \langle A, 1 \rangle$;
else
  $label(A) = \langle -, 0 \rangle$;
end if

Upon receipt of label (messages) from all immediate predecessors of A:
if two or more immediate predecessors have identical non-zero sequence numbers but different owners **then**
  report error; // Error scenario 1
else
  //Compute $label(A)$
  if it is a conflict atom **then**
    $label(A) = \langle A, 1 + max(seq\_no$ of immediate predecessors$) \rangle$;
    $parent =$ the owner with maximum value of seq_no;
    send a $\langle parent \rangle$ message to the process where parent is executed;
  else
    $label(A) = label(B)$ where B is the immediate predecessor with the largest seq_no;
  end if
  send $label(A)$ to each immediate successor of A;
end if

Upon receipt of more than one $\langle parent \rangle$ **message:**
report error; // Error scenario 2

Upon process termination:
if a process has a terminal atom **then**
  It will report it to a central monitor;
  The latter reports error if more than one terminal atom has reported to it;
  // Error scenario 3
end if

Induction step: Now consider the case with $seq\_no = k + 1$. Consider an earliest atom (in WG), say A, with $seq\_no = k + 1$. This atom must be a conflict atom. We show that $label(A)$ will be computed correctly. In WG, there must exist at least one atom, say B, which is the parent of A, such that $label(B) = \langle B, k \rangle$. The propagation of labels in the algorithm ensures that the label of B will be forwarded by those trivial atoms lying between B and A. Hence $label(A) = \langle A, k + 1 \rangle$. Subsequently, other trivial atoms reachable from A without traversing a conflict atom in between, will be correctly labeled with $\langle A, k + 1 \rangle$. This completes the induction step. ■

Now we proceed to prove the correctness of the algorithm by showing that the three scenarios of error report are both necessary and sufficient. In examining the algorithm, we notice that an error is reported when:

i. An atom sees two distinct predecessors with identical *seq_no*.

ii. An atom is the parent of more than one son.

iii. The system terminates with more than one terminal atom.

Theorem 13 *Algorithm* **2** *reports a serialization error if and only if it exists.*

Proof. ($\Rightarrow$): Lemma 12 establishes that the algorithm computes the labels of the atoms correctly. In reporting errors under scenario ***(i)***, when atom A sees two predecessor atoms with identical *seq_no*, then according to the definition of a *label*, it implies that these two conflict atoms cannot lie on the same path. Similarly, in scenario ***(ii)***, when the algorithm discovers that two atoms (say sons of A) have identical *seq_no*, the latter cannot lie on the same path. For the same reasoning, in scenario ***(iii)***, the terminal atoms cannot lie on the same path. Hence the serialization error report in each of the three scenarios is a correct report.

($\Leftarrow$): We use contradiction. Suppose there is a serialization error but the algorithm fails to report it. A serialization error involves two or more conflict atoms that cannot lie on a single path in WG. Take two such latest conflict atoms, say A and B. By latest we mean conflict atoms reachable from A are also reachable from B and vice versa. As a result, there are three possibilities:

i. A and B have a same son (both have a same *seq_no* seen by the same son), or

ii. neither A nor B have a son and hence both are terminal atoms, or

iii. A and B have different *seq_no* and one of them, say A, has a son while B does not have a son.

In case (***i***), the son reports failure. In case (***ii***) failure is reported at termination. In case (***iii***), B is a terminal atom while A must eventually reach a terminal atom. Hence failure is also reported at termination. A contradiction to the no-report assumption is reached. ■

3.6.1 Complexity Analysis

In analyzing the complexity of the algorithm, there are two types of explicit messages:

i. After an atom has computed its label, it forwards this information to its immediate successors. The total number of such messages is bounded by the number of edges $|E|$ in the WG graph.

ii. Subsequently, if it is a conflict atom then it sends an explicit message informing its parent. The total number of such messages is bounded by the number of conflict atoms.

Hence, the message complexity of the algorithm is bounded by $|E|$.

3.7 Summary

The execution of a distributed program generates a large state space, which needs to be considered to check the satisfaction of a property in a given distributed computation. Atoms are useful abstractions in reducing the state lattice of a distributed computation, we refer to the reduced lattice as the atomic state lattice.

With atoms as the basis in viewing a run, checking the satisfaction of a property in a given distributed program run (runtime verification) involves two steps. First, the atomicity of the run needs to be checked. Second, if the run is atomic, then the application specific properties of the program can be checked.

Using the atomic state lattice, one can check program requirements more easily. First, the atomic state lattice is much smaller. Second, the checking does not require expensive instrumentation in augmenting the code with additional auxiliary variables and statements.

Chapter 4

Consequences of Incomplete/ Incorrect Knowledge of Atoms

4.1 Introduction

In the previous chapter we have illustrated that the checking of a distributed program run involves two steps: checking the atomicity of the run and then checking the satisfaction of the application specific properties on the atomic state lattice. In both steps, the conclusion may depend on whether the programmer/tester has identified all the atoms of the program completely and correctly. For example, if only a subset of the atoms is identified, or some atoms are incorrectly identified, will the conclusion about the atomicity of a run remain valid? Will the conclusion about the satisfaction of the required program properties remain valid? Section 4.2 addresses the first question and Section 4.3 addresses the second question.

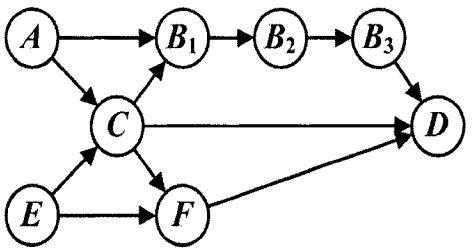

Figure 12: A weak-order graph WG with incompletely identified atoms

4.2 Atomicity Checking with Incomplete/Incorrect Knowledge of Atoms

We denote the sequence of events of an atom, say A, as $\langle a_1, \ldots, a_k \rangle$ where k is the total number of events in atom A. If A is not identified as an atom, then we can assume, without loss of generality, that each of its events will be interpreted as an atom, and hence, we will have k atoms instead of one. Consequently, for each run $\langle E, \rightarrow \rangle$, there are two possible weak-order graphs, $WG = \langle S, W \rangle$ and $WG' = \langle S', W' \rangle$. WG is the weak-order graph using the incompletely identified set of atoms S. WG' is the weak-order graph of the complete set of atoms.

For the example in Figure 5, if B is the atom that is not identified, then the resulting WG graph will be as shown in Figure 12. In comparison with the weak-order graph, say WG', shown in Figure 6, WG contains more atoms, since B is now split into three atoms, B_1, B_2 and B_3. In general, if the atoms of a run are incompletely identified, then some atom(s) in S', say A, contains events $\langle a_1, \ldots, a_k \rangle$ that are treated as individual atoms $A_1, \ldots, A_k$ in S.

Lemma 14 *If a run is non-atomic with respect to an incomplete set of atoms S, then it is also non-atomic with respect to the complete set of atoms S'.*

Proof. As a consequence of Lemma 1, we need to show that if WG is cyclic then WG'

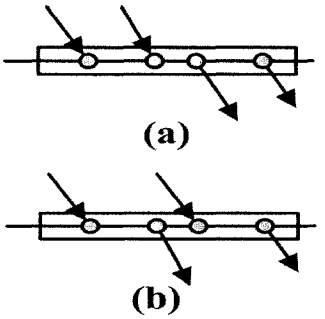

Figure 13: (a) A well-formed atom, (b) A non-well-formed atom.

is also cyclic. From the assumption on incompletely specified atoms, a node X in WG' is either the same node X or a sequence of nodes $\langle X_1, X_2, \ldots, X_k \rangle$ in WG. If WG contains a cycle that contains a sub-path $A \xrightarrow{w} X_i \xrightarrow{w} \ldots \xrightarrow{w} X_{i+u} \xrightarrow{w} B$ where X_i up to X_{i+u} are part of the atom X in WG', then according to the definition of the weak-order relation(Definition 2) we can infer a similar sub-path $A \xrightarrow{w} X \xrightarrow{w} B$ in WG'. Hence a cycle in WG' can be constructed from a cycle in WG. This completes the proof. ■

Definition 10 *An atom $A = \langle a_1, \ldots, a_k \rangle$ is **well-formed** iff whenever a_i is a send event, then a_j (for all $j > i$) is not a receive event.*

Figure 13 shows an example of a well-formed atom and an example of a non-well-formed atom.

Lemma 15 *If a run is atomic with respect to an incomplete set of atoms S and $S' - S$ contains only well-formed atoms, then the run is also atomic with respect to the complete set of atoms S'.*

Proof. Suppose the run is non-atomic with respect to the complete set of atoms S'. Hence WG' is cyclic (from Lemma 1). Next we show that WG must also be

cyclic (if $S' - S$ contains only well formed atoms). More specifically, given a cycle $A \xrightarrow{w} B \xrightarrow{w} C \xrightarrow{w} \ldots \xrightarrow{w} Z \xrightarrow{w} A$ in WG', we can construct a cycle in WG as follows. The cyclic path from A through $B, \ldots, Z$ and back to A traces a sequence of atoms in S'. Without loss of generality, suppose B is an atom in $S' - S$. Let $A = \langle a_1, \ldots, a_m \rangle$, $B = \langle b_1, \ldots, b_n \rangle$ and $C = \langle c_1, \ldots, c_t \rangle$. In other words, there are m events in A, n events in B and t events in C. In the sub-path $A \xrightarrow{w} B \xrightarrow{w} C$, since B is a well formed atom, there must exist events a_i, b_j, b_{j+u} and c_k in A, B and C respectively such that $a_i \rightarrow b_j$, $b_{j+u} \rightarrow c_k$ and $b_j \rightarrow b_{j+u}$ $(u \geq 0)$. If B is not identified as an atom, then the events in B will become individual atoms in WG (i.e., b_j becomes B_j etc). Now we can construct a corresponding sub-path $A \xrightarrow{w} B_j \xrightarrow{w} \ldots \xrightarrow{w} B_{j+u} \xrightarrow{w} C$ in WG. By repeating this splitting process iteratively on atom C (if it is in $S' - S$) and subsequent atoms on the cyclic path, we can identify a cycle in WG as well. Hence, the claim holds. ■

From Lemmas 14 and 15, we can conclude that the result of checking the atomicity of a run with an incomplete set of atoms remains valid as long as the unidentified atoms are all well formed atoms.

Next we consider the scenario where atoms may be incorrectly identified. Suppose the programmer/tester incorrectly identifies some atom by grouping the code blocks that belong to multiple atoms into a single atom. As a result, at runtime an atom may contain events that should have belonged to multiple atoms. Consequently, for a given run, there are also two sets of atoms, S and S' such that some atom in S is formed of a sequence of atoms, all of which belong to the same process, in S'. For example, in Figure 14, the four atoms B_1, B_2, B_3 and B_4 of a process are incorrectly grouped into a single atom B. As a result, the WG corresponding to S contains B instead of four separate atoms, B_1 to B_4, in the correct WG' corresponding to S'.

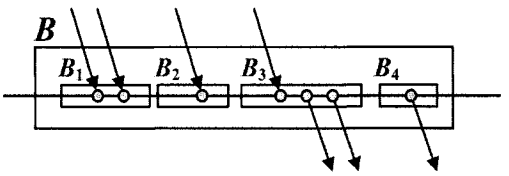

Figure 14: A sequence of well-formed atoms forming a well-formed atom.

Lemma 16 *An atom B formed by a sequence of well-formed atoms $\langle B_1, B_2, \ldots, B_m \rangle$ is a well-formed atom iff there exists an i such that $\{B_j | j < i\}$ are atoms without send events, and $\{B_k | k > i\}$ are atoms without receive events.*

Proof. This is immediate from the definition of well-formed atoms: B_i is the only atom that may contain both send and receive events and B_i is well-formed. Hence, all the receive events in B must happen at or before B_i. ■

Figure 14 illustrates a sequence of well-formed atoms that satisfies the condition in Lemma 16. This lemma asserts the necessary and sufficient conditions for grouping a set of well-formed atoms into a bigger well-formed atom.

In the next two lemmas, S' represents the correct set of atoms and S represents an incorrect set of atoms that is formed by grouping together some atoms in S'.

Lemma 17 *If a run is atomic with respect to an incorrect set of atoms S, then the run is also atomic with respect to the correct set of atoms S'.*

Proof. Suppose that the run is non-atomic with respect to the correct set of atoms S' (WG' is cyclic based on Lemma 1), we want to show that the run is also non-atomic with respect to an incorrect set of atoms S.

If there is a cycle in WG' that contains a sub-path that traverses a sequence of atoms $\langle B_1, B_2, \ldots, B_m \rangle$ that are incorrectly merged to form a bigger atom, say B in

S, then the sub-path $\langle B_1, B_2, \ldots, B_m \rangle$ can be replaced by B to form a cycle in WG corresponding to S. Cyclic WG implies non-atomic run with respect to the incorrect set of atoms S. This completes the proof. ■

Lemma 18 *If a run is non-atomic with respect to an incorrect set of atoms S and $S - S'$ contains only well-formed atoms, then the run is also non-atomic with respect to the correct set of atoms S'.*

Proof. This follows the same line of proof as that in Lemma 15. The details are omitted. ■

From Lemmas 17 and 18, we can conclude that if some atoms are incorrectly grouped together in analyzing a run but these resultant atoms remain well-formed, then the result of checking the atomicity of the run will remain valid.

4.3 Checking Properties with Incomplete/Incorrect Knowledge of Atoms

In this section we will address the validity of property checking in the case where the programmer has incompletely or incorrectly identified the atoms. In the following, we assume that there are two distinct sets of atoms, S_1 and S_2, of a run. Some atoms in S_1 are formed by merging some atoms in S_2. In particular, some atom X in S_1 is formed by merging a sequence of atoms $\langle X_1, \ldots, X_k \rangle$ in S_2. For simplicity of presentation, we also use X_k (instead of X) to represent the merged atom in S_1. As a result of this simplification, $S_2 - S_1$ contains $\{X_1, \ldots, X_{k-1}\}$ whenever $\{X_1, \ldots, X_k\}$ in S_2 is merged to form a single atom in S_1. For example, in Figure 15, $S_1 = \{A, X_3, B\}$ and $S_2 = \{A, X_1, X_2, X_3, B\}$.

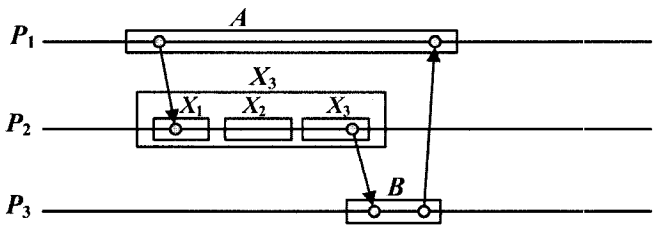

Figure 15: An example of an atomized run.

As a result of the merging of atoms, there are more atoms in S_2 than in S_1, or equivalently, there are more states in the atomic state lattice L_2^* associated with S_2 than in L_1^* associated with S_1. In the case of an incomplete set of atoms, S_1 represents the complete set and S_2 represents the incomplete set. In the case of an incorrect set of atoms, S_1 represents the incorrect set and S_2 represents the correct set.

Definition 11 *An atom whose execution does not affect a global predicate φ is said to be a φ-**independent** atom. Hence, its occurrence will not affect φ.*

We will focus on the same property used in the previous chapter $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$.

Lemma 19 *Suppose the atoms in $S_2 - S_1$ are both φ-independent and ψ-independent. $\boldsymbol{AG}[\varphi \Rightarrow \boldsymbol{AX}(\varphi \vee \psi)]$ holds in the atomic state lattice L_1^* associated with S_1 iff it holds in the atomic state lattice L_2^* associated with S_2.*

Proof. ($\Rightarrow$): S_1 and S_2 differ in that some atom(s), say X_k in S_1, is formed by merging a corresponding set of atoms, say $\{X_1, \ldots, X_k\}$ in S_2. Hence, for each path in L_1^*, there is a corresponding path in L_2^* such that the occurrence of X_k in L_1^* is replaced by the occurrence of atoms in $\{X_1, \ldots, X_k\}$ in L_2^*. The reverse also holds. We refer to this as the path-equivalent property.

Suppose $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold in L_2^*, we want to show that it does not

hold in L_1^*. Since the predicate does not hold in L_2^*, there is a path, say T_2, in L_2^* that traverses a state s and its next state s' via executing an atom X_k such that φ holds in s but $(\varphi \vee \psi)$ does not hold in s'. From the path-equivalent property, there is a corresponding path, say T_1, in L_1^* that reaches s' by executing the atom X_k. There are two cases to be considered:

i. X_k in S_1 is formed by merging $\{X_1, \ldots, X_k\}$ in S_2, and

ii. X_k in S_1 is not formed by merging some atoms in S_2.

Trivially, in case (ii), T_1 also traverses s and then s' via atom X_k, representing a violation of $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ in L_1^*. In case (i), the relevant portions of the paths T_1 and T_2 involving X_k and $\{X_1, \ldots, X_k\}$ respectively are shown in Figure 16. In Figure 16, there is a state s'' in both path T_1 in L_1^* and path T_2 in L_2^* such that s'' reaches s' by executing X_k and $\{X_1, \ldots, X_k\}$ respectively. Since $\{X_1, \ldots, X_{k-1}\}$ are in $S_2 - S_1$, from the condition stated in the lemma, each atom in $\{X_1, \ldots, X_{k-1}\}$ must be both φ-independent and ψ-independent. Hence φ must hold in s'', and subsequently in each state traversed via the atoms from $\{X_1, \ldots, X_{k-1}\}$ in T_2. Consequently, we have identified a state s'' in which φ holds in both L_1^* and L_2^*. But $(\varphi \vee \psi)$ does not hold in s' which is the next state of s'' in path T_1 in L_1^*, as illustrated in Figure 16, causing a violation of $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ in L_1^*.

($\Leftarrow$): Referring to Figure 16 where s'' changes to s' by executing X_k in L_1^*, we can apply the same argument as in the first part of the proof and establish that there is a corresponding state in L_2^* that reaches s', leading to a violation of $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$. Further details are omitted. ■

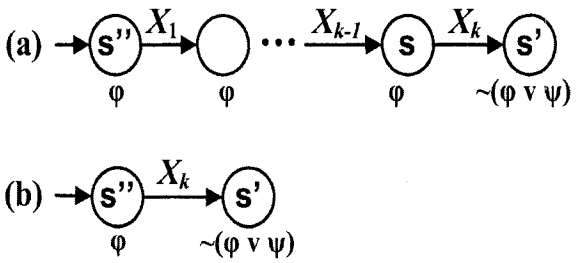

Figure 16: (a) A path in L_2^* in which $\mathbf{AG}[\varphi \Rightarrow \mathbf{AX}(\varphi \vee \psi)]$ does not hold, (b) The corresponding path in L_1^*.

4.4 Summary

Atoms are useful abstractions in reducing the state lattice of a distributed computation. With incomplete or incorrect knowledge of atoms, it is still a viable approach to conduct this reduction, provided that the atoms that are not identified or incorrectly identified are all well-formed atoms. However, the maximum state space reduction can only be achieved with a complete set of atoms.

The checking of program properties with an incomplete or incorrect set of atoms remains valid provided that the mis-identified atoms do not affect the properties concerned.

Chapter 5

Using Synchronized Atoms to Simplify Checking of Distributed Programs

5.1 Introduction

In the previous two chapters we have shown that atoms are useful abstractions in reducing the state lattice of a distributed computation. We have referred to the reduced lattice as the atomic state lattice. However, general predicates remain difficult to check if they are asserted over all states. This chapter presents a formulation to attack this problem involving the separation of two different concerns:

i. Synchronization requirement

ii. Computational requirement

When a number of processes has to maintain a shared global property, we expect that each process should be at some point aware of the changes that can affect the property,

made by other processes; otherwise it will be hard for these processes to maintain this property. This indicates that along with each computational requirement (property) that a distributed program must satisfy, there is a synchronization requirement that must also be satisfied. Consequently, a given property can be specified in a different manner by separating its synchronization aspects and computational aspects. The synchronization and the computational requirements will be specified by two different predicates, synchronization predicate p and computation predicate g. In this case, we do not need to check g in every state; instead we need to check it only on those selected states where the relevant processes are synchronized, i.e., satisfy the synchronization predicate p.

Serialization is the minimal avenue for a set of processes that must maintain a global property to propagate the causal knowledge needed to maintain the global property. This implies that each global state where a synchronization predicate p holds (we call it p-state) must be a proper subset of the next p-state. Consequently, the synchronization requirement can be modeled by the serialization of the p-states. With this modeling assumption, the computational requirement (global property g) needs to be checked only at each p-state. The serialization requirement guarantees that the number of p-states will be bounded by the number of atoms executed, as will be demonstrated in this chapter. Consequently, property g can be checked efficiently even if it is a general property.

The rest of this chapter is organized as follows: Section 2 introduces synchronized atoms, p-states and presents two examples to illustrate how the ideas described above can be useful for efficient property checking in cases where the property of concern does not belong to any of the classes described in section 2.6. In Section 3, two efficient algorithms for checking a general predicate g, in the cases where the synchronization

predicate p is conjunctive or disjunctive, are presented along with their proof of correctness. Section 4 summarizes the results introduced in this chapter.

5.2 Atoms and Serialization

A message passing distributed program consists of processes, each of which owns a set of local variables. The properties that a program must satisfy can be expressed in terms of these variables. Each process performs local computations and exchanges information with other processes through message passing. Atomicity of execution of a sequence of statements in a message passing program can be modeled by its effect on shared (global) properties. The effect of an atomic code block on a shared property should be as if the statements in this code block were executed in sequence without interleaving with the execution of any other statement.

The concepts of an atom and serialization can be used to simplify the checking of distributed programs' properties. In checking a run of a distributed program, we separate two distinct concerns:

i. Synchronization among atoms

ii. Proper computation dependency between atoms and within each atom

Figure 17 shows a simple distributed scheduling example illustrating these two separate concerns with atoms as the basis. P_1 initiates a meeting request by sending a proposed time x to P_2. In turn, P_2 examines its schedule with respect to x and forwards a proposed time $y \geq x$ to P_3. Similarly, P_3 examines its schedule with respect to y and forwards a proposed time $z \geq y$ to P_1. This token ring synchronization among the atoms circulates around the three processes until each process has received a proposed time that matches with the one that it has recently proposed.

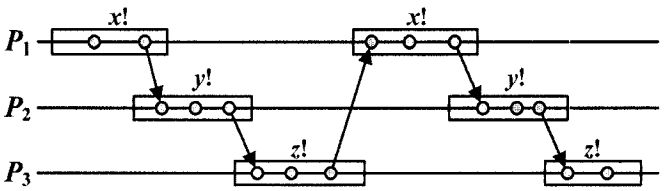

Figure 17: Distributed scheduling example.

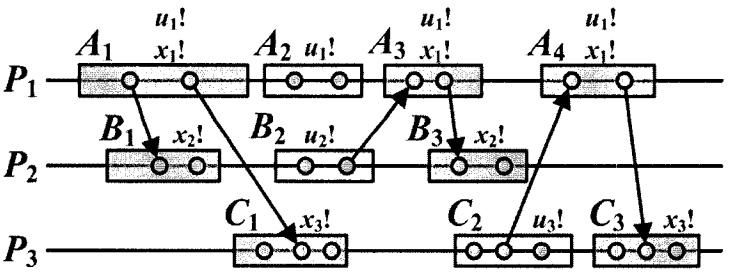

Figure 18: A simplified run of a distributed resource management system.

In Figure 17, the first atom of process P_2 contains the events that include the receipt of a proposed time, checking it against the local schedule and deciding on a proposed time to be forwarded to the next process. The notation $x!$ associated with an atom indicates that the atom modifies variable x one or more times. A proper run contains atoms that are ordered in the form of a token ring. In addition, each atom should compute its proposed time correctly. Specifically, upon completion of each atom, the program state should satisfy the shared property $x \leq y \leq z$.

Figure 18 shows a simplified run of a distributed resource management system that will be used in illustrating our model and checker algorithms. In this figure, there are three processes P_1, P_2 and P_3, executing the respective atoms as shown. The distributed resource management system involves the coordinator process P_1 and two participating processes P_2 and P_3. The actual use of resources is represented by the local variables $\{u_1, u_2, u_3\}$ with the global constraint $u_1 + u_2 + u_3 \leq n$ where n is a constant. Process P_i will optimize the value of u_i depending on other system

conditions known to it. The set of local variables $\{x_1, x_2, x_3\}$ represents the resources allocated to be locally managed by the corresponding processes. In general, it is expected that $u_i \leq x_i$ at all times.

Initially, the coordinator P_1 executes atom A_1. In A_1, P_1 decides to use u_1 units of resources and keep a total of x_1 units of resources to be managed locally. The rest of the resources are distributed to processes P_2 and P_3. Upon notification, P_2 and P_3 update their allocated resources x_2 and x_3 in atoms B_1 and C_1 respectively. Subsequently, these processes locally manage the allocated resources while maintaining the invariant $u_i \leq x_i$, within atoms A_2, B_2 and C_2. Sometime later, process P_2 asks P_1 for more allocation in atom B_2. As a result, P_1 releases more resources to P_2 via atoms A_3 and B_3. Similarly P_3 can ask for more allocations, leading to the run shown in Figure 18.

5.2.1 Well-formed Atoms

An ***atom*** is an identified sequence of events of a process such that the result of executing these events will be as if they were executed without interleaving with any event of any other process. An atom A is a ***well-formed atom*** if every receive event in A happened before every send event in A (Definition 10). The set of well-formed atoms in $\langle E, \rightarrow \rangle$ is denoted by S_E. In this section we will show how well-formed atoms can be useful in reducing the state space that needs to be considered for property checking.

For example, in Figure 18, A_3 is a well-formed atom that includes the events corresponding to the execution of the program statements in P_1 to ***(i)*** receive a request from process P_2, ***(ii)*** locally optimize u_1 and x_1, and ***(iii)*** send a new allocation of resources x_2 to P_2. Only a subset of these events is marked with small circles in

Figure 18. In this chapter the term atom will be used to refer to a well-formed atom unless otherwise specified.

Definition 12 *Given two atoms A and B in S_E, atom A is* ***causally ordered*** *before atom B (denoted by, $A \xrightarrow{c} B$) if there exists an event in A that happened before an event in B.*

A run of a distributed program designed with well-formed atoms will have the following nice property.

Lemma 20 *If S_E contains only well-formed atoms, then their causal order relation ($\xrightarrow{c}$) induces a partial order $\langle S_E, \xrightarrow{c} \rangle$.*

Proof. We only need to prove transitivity and anti-symmetry of $\xrightarrow{c}$.

Transitivity: Consider three well-formed atoms A, B and C in S_E such that $A \xrightarrow{c} B$ and $B \xrightarrow{c} C$. We want to show that $A \xrightarrow{c} C$. From Definition 12, there must exist some latest event a_i in A and some earliest event b_j in B such that $a_i \rightarrow b_j$. Similarly, there must exist some latest event b_k in B and some earliest event c_h in C such that $b_k \rightarrow c_h$. To establish these happened-before relations, b_j can be either the first event (through program order) or a receive event (through message interaction) in B. Similarly, b_k can be either the last event or a send event in B. From Definition 10 on well-formed atoms, we must have b_j happened before b_k. This leads to $a_i \rightarrow b_j \rightarrow b_k \rightarrow c_h$. This implies $A \xrightarrow{c} C$.

Anti-symmetry: Suppose $A \xrightarrow{c} B$ and $B \xrightarrow{c} A$. Then applying the previous reasoning with A replacing C, we have $a_i \rightarrow b_j$ and $b_k \rightarrow a_h$, and $b_j \rightarrow b_k$. Continuing further, a_i is either the last event or a send event in A and a_h is either the first event or a receive event in A. So $a_h \rightarrow a_i$, leading to $a_i \rightarrow b_j \rightarrow b_k \rightarrow a_h \rightarrow a_i$. This

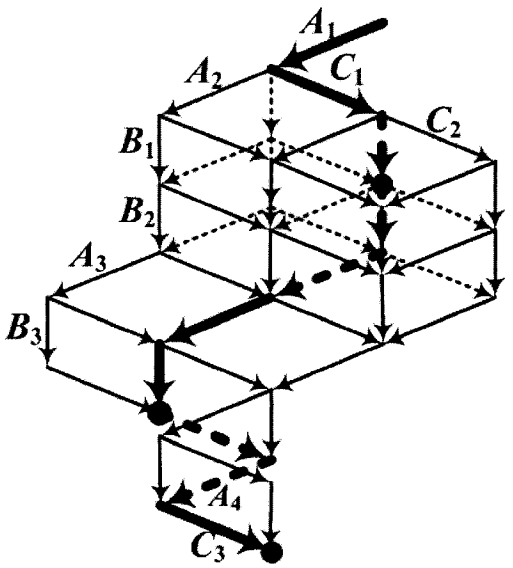

Figure 19: The atomic state lattice (L^*) associated with the run shown in Figure 18.

contradicts the happened-before relation among the events in E. Hence $\langle S_E, \xrightarrow{c} \rangle$ is a partial order on well-formed atoms. ■

As a result of Lemma 20, the partial order $\langle S_E, \xrightarrow{c} \rangle$ also induces a lattice; we refer to it as the atomic state lattice L^*. The atomic state lattice of the run in Figure 18 is shown in Figure 19. Some of the state transitions are labeled with the atoms that cause them.

Lemma 21 *L^* is the sub-lattice of L consisting of all the states reached upon completion of the corresponding prefixes in the partial order $\langle S_E, \xrightarrow{c} \rangle$.*

Proof. This is immediate: for every prefix of $\langle S_E, \xrightarrow{c} \rangle$ whose execution leads to state s in L^* , there is a corresponding sequence of events in $\langle E, \rightarrow \rangle$ whose execution leads to the same state in L. ■

The atomic state lattice L^* of $\langle S_E, \xrightarrow{c} \rangle$ is much smaller than the state lattice L of $\langle E, \rightarrow \rangle$. To illustrate the difference, consider the partial lattices shown in Figure 20.

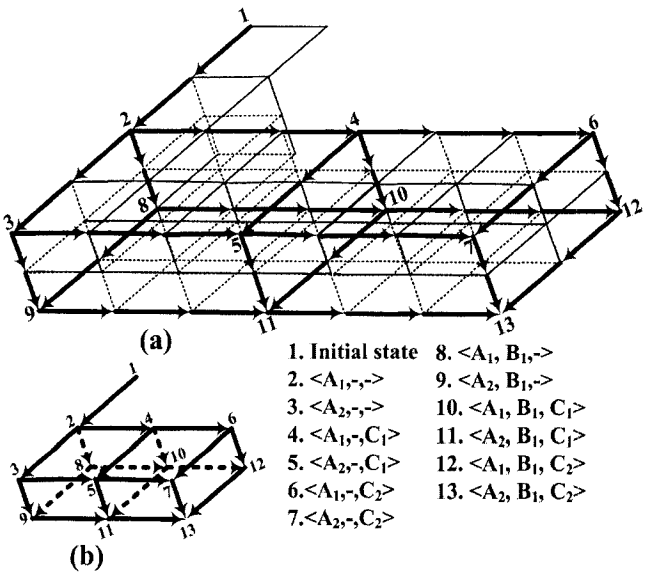

Figure 20: Correspondence between (a) The Event State Lattice (L) and (b) The Atomic State Lattice (L^*).

Figure 20(b) reproduces the atomic state lattice of Figure 19 spanned by the minimal prefix that contains the atoms $\{A_1, A_2, B_1, C_1, C_2\}$. The corresponding state lattice L is shown in Figure 20(a). In comparison, there are only **13** states in Figure 20(b) as compared with **71** states in Figure 20(a).

Even with this significant reduction, the size of L^* can still grow exponentially with the degree of concurrency among atoms in a run. As a result, checking for the satisfaction of a general predicate over L^* would still require exponential time unless some additional assumptions are made. One reasonable assumption is that general predicates are asserted not on all states but only on those selected states where the relevant processes are synchronized, as we have already demonstrated earlier in this chapter. In our example, the coordinator atom A_3 must be ordered before the participant atom B_3. If this is not satisfied, an ordering (synchronization) failure has occurred. However, with proper ordering between A_3 and B_3, computational errors could arise if A_3 does not compute u_1 and x_1 correctly. The latter will appear as a

failure to satisfy the required predicate in some synchronized state, as will be shown later.

5.2.2 Synchronized Atoms (p-atoms) and Serialized p-states

For modeling synchrony in maintaining a general predicate g defined on a set of distributed variables belonging to k processes $\{P_1, P_2, \ldots, P_i, \ldots, P_k\}$ relevant to the general predicate g, we use a synchronization predicate p defined on a set of Boolean synchronization auxiliary variables $\{sync_1, sync_2, \ldots, sync_k\}$. Process P_i owns variable $sync_i$. In some applications the synchronization predicate can be defined in terms of the application variables without the need for auxiliary variables. The system may contain other processes, say $\{P_{k+1}, \ldots, P_n\}$, which are irrelevant to the general predicate g.

Definition 13 *An atom in process P_i is a* ***p-atom*** *if its execution results in $sync_i$ = true. A state in L^* is a* ***p-state*** *if*

i. *It satisfies the synchronization predicate p, and*

ii. *It is the minimal prefix of $\langle S_E, \xrightarrow{c} \rangle$ of the p-atoms included in that state.*

Hence a p-state is reached by having just completed a set of p-atoms and includes only those atoms that are causally ordered before these p-atoms (minimal prefix).

In Figure 21, we have extended Figure 18 with an additional process P_4 that is irrelevant to the property under consideration. This extension is done to reveal a subtlety in condition (***ii***) of Definition 13. From understanding the design of the distributed resource management system, there are atoms that synchronously modify

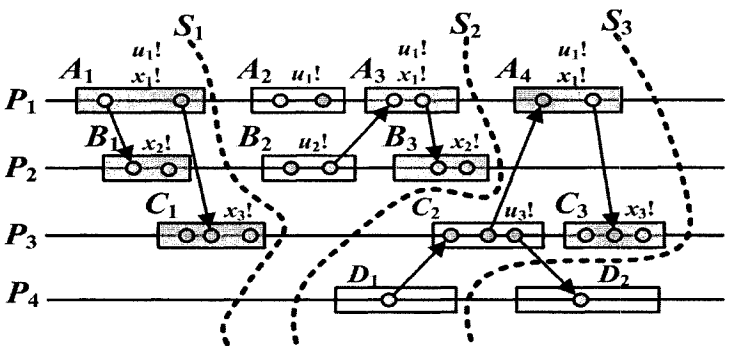

Figure 21: p-states of the example distributed run.

the resource variables $\{x_1, x_2, x_3\}$ so that changes to these variables must be synchronized in order to maintain the shared predicate $g \equiv x_1 + x_2 + x_3 = n$. Then internally, process P_i is responsible for maintaining the local predicate $u_i \leq x_i$.

The synchrony is represented by a synchronization predicate $p = sync_1 \wedge sync_2 \wedge sync_3$. Each process P_i owns an auxiliary variable $sync_i$. These variables are initialized to false at the beginning of each atom. When an atom in process P_i modifies the variable x_i, it will set $sync_i$ to true. In Figure 21, there are seven p-atoms drawn as shaded rectangles. These atoms leave the corresponding Boolean variables $sync_i$ $(i = 1, 2, 3)$ in the true state. Abstractly, we can interpret the synchronization predicate p to indicate that the variables $\{x_1, x_2, x_3\}$ are globally in synchrony.

According to Definition 13, there are precisely three p-states, S_1, S_2 and S_3, at which (i) $p = true$ and (ii) each is the minimal prefix of $\langle S_E, \xrightarrow{c} \rangle$ for the corresponding subset of p-atoms. S_1 contains p-atoms $\{A_1, B_1, C_1\}$, S_2 contains p-atoms $\{A_1, A_3, B_1, B_3, C_1\}$, and S_3 contains all of the p-atoms. For simplicity, we denote each p-state by the most recent p-atom executed in each process. For example, $S_2 = \langle A_3, B_3, C_1 \rangle$.

Notice that there are more states than the three p-states identified that satisfy p. For example, the state $\langle A_1, B_1, C_1, D_1 \rangle$ satisfies p, but is not a p-state, as it does not satisfy the minimal prefix condition (ii) in Definition 13. The changes that may affect g are all occurring in the p-atoms $\{A_1, B_1, C_1\}$. D_1 can not influence any of these changes as it is clear in Figure 21. Consequently, there is no need to consider $\langle A_1, B_1, C_1, D_1 \rangle$ as a p-state.

In the above, we have illustrated how to identify whether a state is a p-state or not. The next step is to see how we can check that these p-states are serialized, i.e., each p-state is a proper subset of the next p-state. As we have demonstrated earlier, serialization is a minimal avenue to propagate the causal knowledge needed to maintain the global predicate g.

Definition 14 *A run has serialized p-states if all the p-states of* $\langle S_E, \xrightarrow{c} \rangle$ *lie on a single path in the atomic state lattice* L^* *of* $\langle S_E, \xrightarrow{c} \rangle$.

In Figure 19, the three p-states $\{S_1, S_2, S_3\}$ are marked with black balls and they lie on a single path (highlighted in bold lines) in L^*. Hence this run has serialized p-states. The significance of p-states is that at those states where synchrony is reached, the general predicate g on the distributed variables $\{x_1, x_2, x_3\}$ must be satisfied.

Lemma 22 *The set of serialized p-states of a run can be ordered as a sequence of prefixes of* $\langle S_E, \xrightarrow{c} \rangle$ *under the subset relation. In other words, they can be ordered as* $S_1 \subset S_2 \subset S_3 \subset \ldots \subset S_m$.

Proof. The set $\{S_1, \ldots, S_m\}$ of serialized p-states of a run lie on a single path in L^*. S_2 is reached from S_1 in L^* by executing the sequence of atoms that form the sub-path between them. Hence $S_1 \subset S_2$. Repeating this argument inductively on S_i establishes the claim. ■

Definition 15 *A run satisfies* $\langle g, p \rangle$ *if*

i. *It has serialized p-states, and*

ii. *g is satisfied in every p-state.*

An intuitive justification can be provided for the requirement stated in Definition 15. The example has illustrated the relationship between synchronized atoms (p-atoms) in the form of serialized p-states and the general predicate g that should be satisfied at these states. For this to be feasible in a distributed system, generally we expect that the changes to the variables of g in a p-state should be known to some p-atom(s) in the next p-state. If these p-states are serialized, from Lemma 22, they form an ordered sequence of prefixes of $\langle S_E, \xrightarrow{c} \rangle$. As a result, there is a corresponding advancement from one causal cone (S_i) to the next causal cone (S_{i+1}) and so on. This is a minimal requirement for the changes to the predicate variables associated with g in S_i to be known to (S_{i+1}).

Figure 22 shows an erroneous run that differs from the one in Figure 18. This error is caused by the omission of process P_2 to request more allocation from P_1 before proceeding to increase its allocation. As a result, there are four p-states in this run, namely $S_0 = \langle A_1, B_1, C_1 \rangle$, $S_1 = \langle A_1, B_2, C_1 \rangle$, $S_2 = \langle A_3, B_1, C_3 \rangle$ and $S_3 = \langle A_3, B_2, C_3 \rangle$, these states are marked with black balls in the atomic state lattice shown in Figure 22. In particular, S_1 and S_2 cannot reach each other, these two states do not lie on the same path in the corresponding atomic state lattice. As a result, p-states are not serialized and hence $\langle g, p \rangle$ is not satisfied.

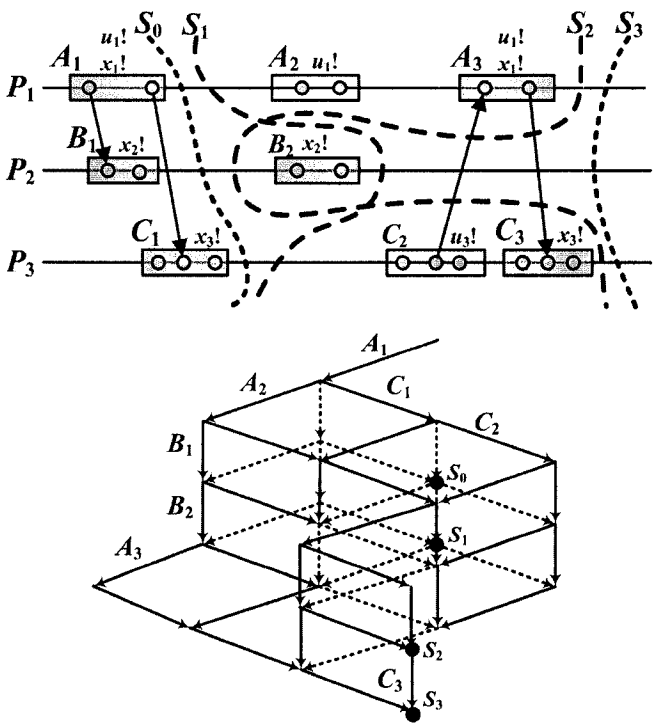

Figure 22: A run without serialized p-states and the corresponding atomic state lattice.

5.3 Checker Algorithms for $\langle g, p \rangle$

In this section, we will present and prove two checker algorithms for checking $\langle g, p \rangle$, one for a conjunctive p and the other for a disjunctive p. The two can be merged for checking more general forms of p, such as a disjunction of conjunctions.

5.3.1 Conjunctive p

Given a distributed computation $\langle S_E, \xrightarrow{c} \rangle$, a general predicate g and a conjunctive synchronization predicate $p = sync_1 \wedge sync_2 \wedge \ldots \wedge sync_k$, our task is to check:

i. The serialization of p-states.

ii. The satisfaction of g in each p-state.

In Figure 22, the p-atoms are shown in shaded rectangles. As before, we identify a p-state with the latest p-atom in each process. For two p-states S_i and S_j that satisfy p, it has been proved that $S_i \cap S_j$ (denoting the maximal prefix of $\langle S_E, \xrightarrow{c} \rangle$ common to both S_i and S_j) also satisfies p [Gar02]. As a result of this property, there must exist a unique earliest p-state that can reach every p-state in the atomic state lattice L^*. The checker strategy proceeds by detecting this first p-state S. Successively it moves forward by pruning this prefix from the current $\langle S_E, \xrightarrow{c} \rangle$ and continues to search for its successor p-state(s) in L^*. Satisfaction of g is checked separately on each identified p-state. Algorithm 3 shows the details of the checker.

In Algorithm 3, we have omitted the details of finding the next p-state of $\langle S_E, \xrightarrow{c} \rangle$. This is accomplished by Algorithm 4.

As an illustration of the checker algorithm, consider the run shown in Figure 22. Initially $S_0 = \langle A_1, B_1, C_1 \rangle$, which is a p-state. Omitting A_1 leads to $S_2 = \langle A_3, B_1, C_3 \rangle$.

Algorithm 3 The checker algorithm for $\langle g, p\rangle$ where p is conjunctive

Notation: A p-state is denoted by $S = \langle p_1, p_2, \ldots, p_k\rangle$ where p_i is the last p-atom in process P_i in S.
For a p-atom p_i in process P_i, $next(p_i)$ is the next p-atom after p_i in process P_i (if it exists).
$C(p) = \{p' | p'$ in S_E and $p' \xrightarrow{c} p\}$
//Notice that $C(p)$ includes atom p.

$S =$ the first p-state of $\langle S_E, \xrightarrow{c}\rangle$;
if S does not satisfy g **then**
  report a failure;
end if
repeat
  $S^*_E = S_E$;
  $S' = S_E$;
  for $i = 1$ to k **do**
    $S'_E = S^*_E - C(next(p_i))$;
    $S_i =$ the first p-state of $\langle S'_E, \xrightarrow{c}\rangle$;
    if S_i exists **then**
      if S_i does not satisfy g **then**
        report a failure;
      else
        if $S_i \subset S'$ **then**
          $S' = S_i$; //keep the earliest next p-state
          $S_E = S'_E$;
        else
          if $\neg(S' \subset S_i)$ **then**
            report a failure; //S' and S_i cross-over
          end if
        end if
      end if
    end if
  end for
  $S = S'$;
until $S'_E = \emptyset$;

Algorithm 4 Algorithm to find the first p-state of $\langle S_E, \xrightarrow{c} \rangle$

Initially: For a given $\langle S_E, \xrightarrow{c} \rangle$, $\langle f_1, f_2, \ldots, f_k \rangle$ = minimal prefix in $\langle S_E, \xrightarrow{c} \rangle$ that contains $\{f_i | f_i$ is the first p-atom in $P_i, i = 1, \ldots, k\}$

repeat
  $S = \langle f_1, f_2, \ldots, f_k \rangle$;
  if $\exists i$: the last atom a_i of P_i in S differs from f_i **then**
    f_i = the next p-atom in P_i;
    // $f_i = \emptyset$ if there are no more p-atoms in P_i
  end if
until (S is a p-state) or ($\exists i : f_i = \emptyset$);
if $\exists i : f_i = \emptyset$ **then**
  report no p-state found;
else
  return S;
end if

The checker replaces S' with S_2. Omitting B_1 leads to $S_1 = \langle A_1, B_2, C_1 \rangle$. Since $\neg(S_1 \subset S')$ and $\neg(S' \subset S_1)$, the checker reports a failure of serialization between p-states.

We proceed now to prove the correctness of the checker algorithm.

Lemma 23 *If p-state(s) exist in $\langle S_E, \xrightarrow{c} \rangle$, then the first p-state (not reachable by another p-state) is unique.*

Proof. This follows immediately from the property that if S_i and S_j are p-states, then $S_i \cap S_j$ is also a p-state. Hence if there are multiple first p-states, say, S_i and S_j, this is a contradiction: $S_i \cap S_j$ is a first state that can reach both of them. ■

Lemma 24 *If $\langle S_E, \xrightarrow{c} \rangle$ does not contain serialized p-states, the checker algorithm reports the first failure represented by two successor p-states of a parent p-state and these successor p-states cannot reach each other in L^*.*

Proof. From Lemmas 23, the checker algorithm detects the first p-state $S = \langle p_1, p_2, \ldots, p_k \rangle$

of $\langle S_E, \xrightarrow{c} \rangle$. There are at most k next p-states $\{S_1, S_2, \ldots, S_k\}$ of S, such that S_i is the next p-state reached by having executed at least the p-atom after p_i in P_i. If there are two successor p-states of S that cannot reach each other, they must be in this set. In the for-loop of the algorithm, $S_1, S_2, \ldots, S_k$ are generated iteratively. In iteration 1, it retains S_1 (if it exists) as the potential next p-state of S (coded also as S' in the algorithm). In iteration 2, it checks if S_1 can reach S_2 or vice versa (via $S_1 \subset S_2$ or $S_2 \subset S_1$). In the latter case, it retains S_2 as the potential next p-state of S. If neither holds, then a failure is correctly reported. By repeating this for all k potential next states of the initial p-state S, the algorithm guarantees reporting of failure if two successor p-states of S cannot reach each other. Furthermore, if none is detected among the k potential next p-states of S, then the retained state in the for-loop becomes the unique next p-state of S and the whole process repeats with this new p-state as S until eventually either $\langle S_E, \xrightarrow{c} \rangle$ contains serialized p-states or the first failure will be reported. ■

Theorem 25 *The checker algorithm (Algorithm 3) correctly reports a failure if a run does not satisfy* $\langle g, p \rangle$.

Proof. From Lemma 24, the checker reports serialization failure. In addition, the algorithm also checks g in each p-state and reports a failure if it is not satisfied in any p-state. ■

The checker algorithm can be adapted into an online monitoring system in which a run is checked for its satisfaction of $\langle g, p \rangle$. We will describe briefly some relevant details of the adaptation. These include **(*a*)** use of vector time stamp to track $\xrightarrow{c}$ among well-formed p-atoms, and **(*b*)** forwarding of the time-stamp of each atom and the value (v_i) of the predicate variable x_i associated with that atom to a checker monitor that implements the checker algorithm dynamically, as $\langle S_E, \xrightarrow{c} \rangle$ unfolds.

Associated with process P_i is a monitor kernel that assigns a vector timestamp $\langle t_1, t_2, \ldots, t_i, \ldots, t_k \rangle$ to each p-atom executed by P_i, with the following details:

Upon executing the first event in a p-atom:

$\langle t_1, t_2, \ldots, t_i, \ldots, t_k \rangle = \langle t_1, t_2, \ldots, t_i + 1, \ldots, t_k \rangle$

Upon receiving a message tagged with $\langle t'_1, \ldots, t'_k \rangle$:

$\langle t_1, \ldots, t_k \rangle = \langle max(t_1, t'_1), \ldots, max(t_k, t'_k) \rangle$

Upon sending a message to process P_j:

Tag $\langle t_1, t_2, \ldots, t_k \rangle$ on the message if it was not done before.

Upon completing the last event of a p-atom:

Send $(\langle t_1, t_2, \ldots, t_k \rangle, v_i)$ to the checker monitor.

Using the above distributed monitors, the ordering among p-atoms can be established by comparing the vector timestamps of each p-atom. Consequently, the checker monitor can verify $\langle g, p \rangle$ by constructing $\langle S_E, \xrightarrow{c} \rangle$ and applying the previous algorithm.

5.3.2 Disjunctive p

Checking that the p-states are serialized where p is a disjunctive predicate $p = sync_1 \vee sync_2 \vee \ldots \vee sync_k$ is rather straight forward. We will present a fully distributed on-the-fly checker of $\langle g, p \rangle$. To support a fully distributed checker, we make a stronger assumption on well-formed atoms and refer to this as strictly well-formed atoms.

Definition 16 *A well-formed atom is strictly well-formed if every event that changes a predicate variable associated with the atom happened before every send event in the atom.*

The significance of a strictly well-formed atom lies in the fact that the effect of the atom to a predicate variable, say x_i, can be communicated to other p-atoms through tagging the necessary causal information on application messages. Abstractly each process maintains a vector $\langle t, owner, v \rangle$ that reflects ***(i)*** the *owner* process of the most recent p-atom whose sequence number (in the serialization order) is t, and ***(ii)*** the values (v) of the variables relevant to the predicate g. Algorithm 5 shows the details of the $\langle g, p \rangle$ checker where p is disjunctive.

Now we will proceed to prove the correctness of Algorithm 5.

Lemma 26 *If a run contains p-states that are not serialized, then a failure will be reported.*

Proof. There are two cases: ***(i)*** the first p-state in L^* is unique, and ***(ii)*** there are two first p-states S_1 and S_2 that cannot reach each other in L^*.

Case *(i)*: Consider the first p-state in L^*, say S, which can reach two successor p-states S_1 and S_2 but neither of the latter can reach the other. Because p is a disjunctive predicate, these three states are entered by the execution of three separate atoms, say p_1, p_2 and p_3 respectively. Since S is a first such p-state and neither S_1 nor S_2 reach each other, the following causal relations $p_1 \xrightarrow{c} p_2, p_1 \xrightarrow{c} p_3, \neg(p_2 \xrightarrow{c} p_3)$ and $\neg(p_3 \xrightarrow{c} p_2)$ must hold. With well-formed atoms and causal forwarding of $\langle t, owner, v \rangle$, the monitors associated with the processes that execute p_2 and p_3 will identify the process that execute p_1 as its parent and will send an acknowledgment to the monitor associated with it. Hence a serialization failure will be reported by the latter monitor.

Case *(ii)*: Since S_1 and S_2 are two first p-states, they have identical sequence number $t = 1$. From the algorithm, both will identify P_1 as its parent and send an acknowledgement to the monitor of P_1, who will report the failure. ■

Algorithm 5 The checker algorithm for $\langle g, p \rangle$ where p is disjunctive.

Monitor/Checker Kernel at P_i**:**
***Initially*:**
$owner = P_1$;
$previous = 0$;
$t = 0$;

Upon receipt of a message tagged with $\langle t', j, v' \rangle$**:**
if $t < t'$ **then**
  $\langle t, owner, v \rangle = \langle t', j, v' \rangle$;
end if

Before executing the first send event in a p***-atom*:**
$parent = owner$;
$previous = t$;
$\langle t, owner \rangle = \langle t + 1, P_i \rangle$;
if $\neg g(v)$ **then**
  report a failure;
end if

Upon executing a send event to P_j**:**
Tag the message with $\langle t, owner, v \rangle$ if it was not done before;

Upon completing the last event of a p***-atom*:**
Send $acknowledge(previous)$ to the process identified by $parent$;

Upon receipt of acknowledge(t' ***)*:**
if $acknowledge(t')$ has been received before **then**
  Report a failure;
end if

Lemma 27 *If a run contains a p-state that does not satisfy g, a failure will be reported.*

Proof. Under strictly well-formed atoms assumption, the values of the predicate variables associated with each p-state are correctly obtained by merging ***(i)*** those propagated from the parent p-state via causal forwarding, and ***(ii)*** those updated in the current p-atom. Upon completion of the current p-atom, the general predicate g will be checked in the new p-state and a failure will be reported if g is not satisfied.■

Theorem 28 *The checker algorithm (Algorithm **5**) correctly reports a failure if a run does not satisfy* $\langle g, p \rangle$.

Proof. Follows directly from Lemmas 26 and 27. ■

5.4 Summary

We have presented a model using well-formed atoms as the basis in obtaining a reduced state lattice L^* for monitoring and checking a distributed program run. In analyzing the relationship between a general predicate on local variables and synchronization among the atoms executed, we postulate that synchronized states (p-states) reached upon executing p-atoms must be serialized. Intuitively, this is justified by the causal cone representing one p-state migrating to the causal cone representing the next p-state (Lemma 22). Serialization is a minimal avenue for a set of atoms that moves the system from one p-state to another p-state to propagate causal knowledge needed to maintain the global predicate g.

We have developed two efficient algorithms for checking $\langle g, p \rangle$, where g is a general predicate and p is a conjunctive or disjunctive synchronization predicate. The number

of the states where a general predicate g needs to be checked will be bounded by the number of atoms executed.

Chapter 6

Checking Distributed Programs with Partially Ordered Atoms

6.1 Introduction

Partially ordered multi-sets (POMSETs) are attracting more attention in modeling and analyzing distributed programs due to that fact that they model true concurrency [LRG04, LL94, PL93, PL91b, PL91a, Pra86]. This contrasts with the implicit modeling of concurrency in the interleaving model where concurrency is simulated by commuting the corresponding events in separate linear traces. This gives rise to additional complexity in analysis and checking in the interleaving model as has been described in Section 2.10.

In Chapter 3 we have illustrated the effectiveness of exploiting design time knowledge to identify a set of atoms that can be used to abstract a distributed program computation [LMG07a, LMG07b]. In this chapter, we address both the atomization problem and the checking of program properties using a partially-ordered multi-set

(POMSET) model of atoms.

Atomization can be performed with a hierarchy of conditions. The coarsest condition simply requires the result of executing a sequence of events forming an atom to be as if these events were executed in sequence without interleaving with any event of any other process. This condition does not distinguish events that are relevant to the required program properties from other events and hence will lead to the coarsest granularity of atoms. By distinguishing relevant events from other events, stricter characterizations of atoms will result. In particular, by considering the multiplicity of relevant events and their ordering with respect to other events within an atom, atomization can lead to well-formed mini or maxi atoms that are relevant in program checking.

Atomization is a rather intuitively applicable technique for analyzing a distributed program. For example, most of the agent protocols [FIP] can be easily atomized. These can be abstractly viewed as protocols that execute a number of atoms. The correctness of a protocol involves two aspects, the correctness of the order in which its atoms are executed and the correctness of the computation performed by each atom.

Slicing is a filtering technique used to extract relevant entities for analysis, as used in program comprehension and property checking [LRG04, MG01, Wei84]. Since not every atom is directly relevant to a required program property, an atomization can also be sliced to create a simpler atom slice for checking, as will be illustrated in Section 2.

The rest of this chapter is organized as follows: Section 2 contains the formulation of the atomization model. In section 3, we present our order requirements specification model and the checking procedure. In section 4, a simple example is presented to illustrate the results stated in sections 2 and 3. In section 5 we use a POMSET

automaton to specify ordering requirements. Section 6 presents some guidelines that can help programmers to use the methodology introduced in this chapter, followed by the summary in section 7.

6.2 Atomization

Atomization is an abstraction mechanism that can effectively compress all the events of a run into a smaller number of atoms to simplify monitoring and checking the run. There are many different layers of abstractions that can be used in creating atoms. This section presents our structured abstractions and their respective properties, followed by an interesting optimization using event reordering and a theorem relating these abstractions.

6.2.1 Partial Order of Atoms

Following the notations used in [LRG04, PL91b, Pra86], a run of a distributed program can be defined as follows:

Definition 17 *A **run** of a distributed program is a partially ordered multiset (POMSET)* $\sigma = \langle \bar{a}, \bar{e}, \rightarrow, L \rangle$ *where*

- ***i.*** $\bar{a}$ *is the set of program statements.*
- ***ii.*** $\bar{e}$ *is the set of events each corresponding to the execution of a statement. An event can be an internal event or a send/receive event.*
- ***iii.*** $\rightarrow$ *is the happened-before relation [Lam90] between events induced by program order and send/receive order in message passing.*

iv. *$L : \bar{e} \rightarrow \bar{a}$ is the mapping of events to program statements.*

Figure 23 shows an example run with three processes. For simplicity, the event labels are not explicitly shown in the figure.

We present below a hierarchy of different types of atoms. The base level is the coarsest type of atoms that satisfy a simple condition defined as follows.

Definition 18 *An **atom** E is a contiguous sequence of events $\langle e_1, \ldots, e_k \rangle$ of a process that satisfies the following condition: if e_i is a send event then e_j $(j > i)$ must not be a receive event.*

Intuitively, an atom is allowed to be influenced by other processes by receiving messages before it sends any message that can influence other processes. Once it has sent a message, a receive event of the same process will mark the beginning of the next atom. Hence, once an atom has sent something that may affect others, it is not allowed to perform a receive event that brings in new effects from other processes.

The run in Figure 23 is partitioned into five atoms $\{E_1, \ldots, E_5\}$. Rectangles are used to depict atoms, circles and black squares are used to depict events. According to Definition 18, the events in process P_1 are split into two atoms: E_1 and E_2, whereas all the events in process P_3 form a single atom E_5.

The casually ordered relation $\xrightarrow{c}$ among atoms introduced in Definition 12 of Chapter 5 is applicable to the set of atoms that satisfy the condition stated in Definition 18. As a result the set of atoms along with the causally ordered relation induces a partial order among these atoms as we have proved in Lemma 20.

As a consequence of Lemma 20, atoms are effective abstractions of the events in a run as the partially ordered set of atoms has a corresponding atomic state lattice that can be significantly smaller than the state lattice of the original run, as we have

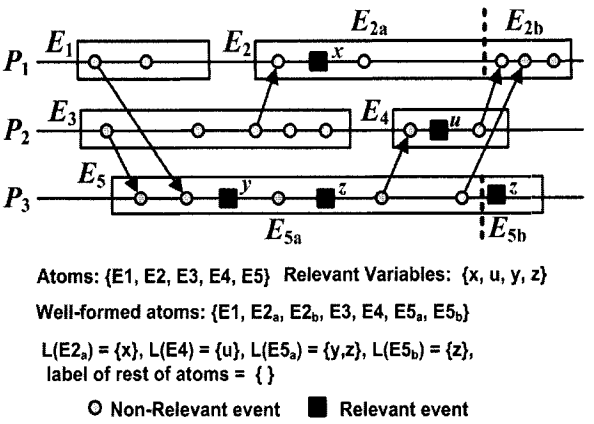

Figure 23: An example run and its resulting atoms/well-formed atoms.

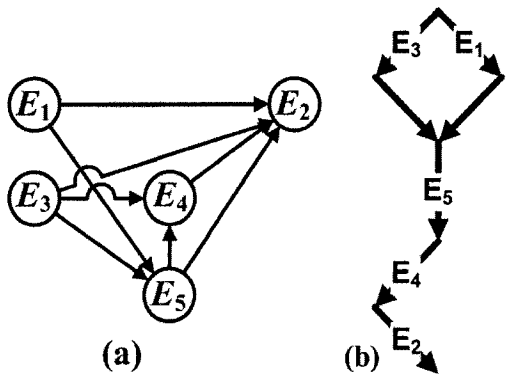

Figure 24: (a) Partial order of the atoms shown in Figure 23 (b) Atomic state lattice.

demonstrated in Chapter 3. Figure 24(a) shows the partial order among the five atoms of the run shown in Figure 23. The corresponding atomic state lattice contains only 7 states and is shown in Figure 24(b).

Compressing the event space into atoms using the condition stated in Definition 18 alone may be too crude, leading to a partial order set of atoms that omits some important details of the program that should be checked. To model these fine-grain and important details, the notion of relevant variable associated with the required program properties is needed.

Definition 19 *A program variable is a* ***relevant variable*** *if it is used in specifying the ordering or the computational requirements of the distributed program. The set of relevant variables is denoted by* $\bar{A}$. *A* ***relevant statement*** *is a program statement whose execution may change the value of a relevant variable. Correspondingly the execution of a relevant statement is a* ***relevant event***. *A* ***relevant atom*** *is an atom that contains at least one relevant event.*

For example, in Figure 23, black squares labeled with the corresponding relevant variables are used to depict relevant events. The set of relevant variables is $\{x, u, y, z\}$ and the set of relevant atoms is $\{E_2, E_4, E_5\}$.

Relevant variables are used to identify fine-grain details represented by relevant events that should not be overlooked in the abstraction of atoms. This leads to the next restriction on an atom to become a well-formed atom. The following definition extends Definition 10 by considering relevant events.

Definition 20 *An atom* $E = \langle e_1, \ldots, e_k \rangle$ *is* ***well-formed*** *if it satisfies the following conditions:*

i. *if* e_i *is a relevant event then* $e_j (k \geq j > i)$ *cannot be a receive event.*

ii. *if* e_i *is a send event then* $e_j (k \geq j > i)$ *cannot be a relevant or receive event.*

iii. *E is maximal in length: if E is extended by including either the event preceding* e_1 *or the event succeeding* e_k *in the same process then the extended sequence of events is no longer a well-formed atom.*

A well-formed relevant atom is called a ***mini atom*** *if it contains at most one relevant event. Otherwise it is called a* ***maxi atom***.

For example, in Figure 23, E_2 is not a well-formed atom; there is a receive event after the relevant event x. Hence, E_2 must be split into two well-formed atoms E_{2a} and E_{2b}, separated by a dashed vertical bar in the figure. Similarly, E_5 must be split into two well-formed atoms E_{5a} and E_{5b}. Since E_{5a} contains two relevant events, it is a maxi atom, whereas the rest of the well-formed atoms are mini atoms.

Intuitively, a well-formed atom has the following nice properties:

i. It has received all external influence through receive events before the occurrence of any relevant event within the atom.

ii. All relevant events have occurred before it sends any message that can influence other processes.

iii. It includes a maximal (coarsest grain) number of events.

Hence a well-formed atom is as if it is instantaneously executed with respect to all other processes: it has listened to others before doing the "relevant changes" and then making some relevant results known to others. Restricting the number of relevant variables within a well-formed atom to at most one allows further fine-grain details to be examined, and it leads to the finest type of atoms called *mini atoms.*

In the remaining part of this chapter, it will be assumed that atoms are well-formed, unless otherwise explicitly stated. Some interesting properties of well-formed atoms are proved below.

Let $\bar{E}^i$ denote the sequence of well-formed atoms in process P_i.

Lemma 29 *$\bar{E}^i$ is unique.*

Proof. Suppose there are two different partitions of the events in process P_i into sequences of well-formed atoms, say $\bar{E}^i$ and $\bar{E}^{i*}$. Suppose the first difference occurs

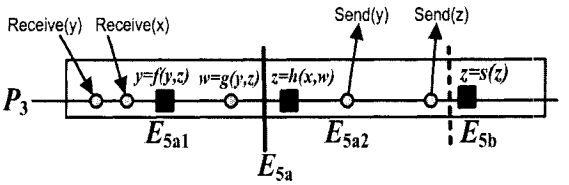

Figure 25: A Maxi Atom = A Sequence of Mini Atoms ($E_{5a} = \langle E_{5a1}, E_{5a2} \rangle$).

at the j^{th} atom of these two sequences, say E and E^*. Since the atoms preceding E and E^* are identical, E and E^* must start with the same event. If one of them is shorter than the other, then the two atoms cannot be well-formed atoms without violating the maximal length condition (Definition 20). Hence E and E^* cannot be different and the claim must hold. ■

Lemma 30 *A maxi atom E is equivalent to a sequence of mini atoms $\langle E_1, \ldots, E_j \rangle$, where j is the number of relevant events in E.*

Proof. Suppose the maxi atom $E = \langle e_1, \ldots, e_k \rangle$ contains a sequence of j relevant events $e_{r_1}, e_{r_2}, \ldots, e_{r_j}$. From Definition 20, $e_i (i > r_1)$ cannot be a receive event, and $e_i (i > r_j)$ must be a send or a non-relevant event. Hence it can be directly verified that $E_1 = \langle e_1, \ldots, e_{r_2 - 1} \rangle$, $E_2 = \langle e_{r_2}, \ldots, e_{r_3 - 1} \rangle$, $\ldots$, $E_j = \langle e_{r_j}, \ldots, e_k \rangle$ are j mini atoms equivalent to the maxi atom E. ■

Figure 25 illustrates the equivalence of the maxi atom E_{5a} and the two mini atoms E_{5a1} and E_{5a2}.

From Lemmas 20 and 29, a run of a distributed program can be atomized and represented by another POMSET defined as follows.

Definition 21 *An **atomized run** is a POMSET $\Sigma = \langle 2^{\bar{A}}, \bar{E}, \xrightarrow{c}, L \rangle$ where*

i. *$\bar{A}$ is the set of relevant variables.*

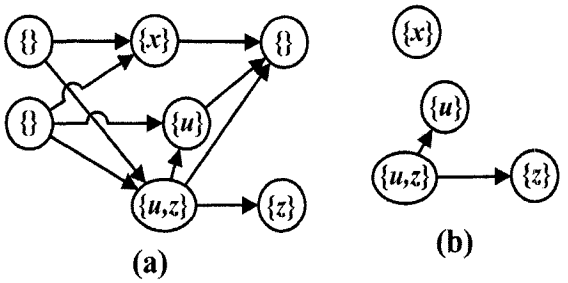

Figure 26: (a) Atomized run (b) Atom slice.

ii. *$\bar{E}$ is the set of well-formed atoms that partitions $\bar{e}$.*

iii. *$\xrightarrow{c}$ is the causally-ordered relation between atoms in $\bar{E}$.*

iv. *$L : \bar{E} \rightarrow 2^{\bar{A}}$ is a labeling function that labels each atom with the set of relevant variables associated with its relevant events. For example, $L(E_{5a}) = \{y, z\}$. In case an atom E is not a relevant atom, $L(E) = \{\}$.*

Definition 22 *The **atom slice** of a run is the projection of the atomized run $\Sigma = \langle 2^{\bar{A}}, \bar{E}, \xrightarrow{c}, L\rangle$ onto the set of relevant atoms $\bar{E}_r$ and is denoted by $\Sigma_r = \langle 2^{\bar{A}}, \bar{E}_r, \xrightarrow{c}, L\rangle$.*

Figure 26(a) shows the atomized run (POMSET) of the run shown in Figure 23. Each node represents a well-formed atom and is marked with its label. Figure 26(b) shows the corresponding atom slice, which contains the smallest number of atoms that need to be monitored and checked.

A more restricted atomization of a run can be performed by decomposing each maxi atom of the run into a sequence of mini atoms, according to Lemma 30. Hence the following mini-atomized run can be defined.

Definition 23 *A **mini-atomized run** is a POMSET $\Sigma_m = \langle \bar{A}, \bar{E}_m, \xrightarrow{c}, L\rangle$ where*

i. *$\bar{A}$ is the set of relevant variables.*

ii. *$\bar{E}_m$ is the set of mini atoms that partitions $\bar{e}$.*

iii. *$\xrightarrow{c}$ is the causally-ordered relation between atoms in $\bar{E}_m$.*

iv. *$L : \bar{E}_m \rightarrow (\bar{A} \cup \{\})$ is the mapping of mini atoms to relevant variables. In case an atom E is not a relevant atom, $L(E) = \{\}$.*

6.2.2 Event Reordering

Reordering of independent events in a process can be utilized to merge a sequence of well-formed atoms into a larger well-formed atom. As a result, the atomization will contain a smaller number of atoms. Consequently, the cost of monitoring/checking a run will be reduced.

Definition 24 *Two events e_i and $e_j (j > i)$ of a process are independent if there is no data or control dependency from e_i to e_j.*

From dependency theory [KKP+81], it is known that the result of these events will be the same if e_j is executed before e_i. Atomization can be further optimized by reordering of independent events. Specifically, we allow reordering between relevant events and send/receive events if they are independent so that a maxi atom can be grown as much as possible.

Algorithm 6 makes use of event reordering to partition the sequence of events of a single process into a sequence of well-formed atoms.

In Figure 27, by applying Algorithm 6, the top three well-formed atoms A_1, A_2 and A_3 can coalesce into the bottom well-formed atom A. In the first iteration, the $Receive(x)$ event does not depend on the relevant event $y = f(y, z)$ and it is reordered

Algorithm 6 Atomization with reordering of independent events.

Input: The sequence of events of a process $\langle e_1, \ldots, e_m \rangle$.
Output: Equivalent sequence of well-formed atoms.

$i = 1$;
repeat
  detect the current well-formed atom $E = \langle e_i, \ldots, e_{i+k-1} \rangle$; //according to Definition 20.
  if e_{i+k} is a receive event **then**
    if e_{i+k} and all the relevant events in E are independent **then**
      reorder* e_{i+k} before the latter;
    else
      $i = i + k$;
      report E;
    end if
  else
    if e_{i+k} is a relevant event **then**
      if e_{i+k} and all the send events in E are independent **then**
        reorder* e_{i+k} before the latter;
      else
        $i = i + k$;
        report E;
      end if
    end if
  end if
until $(i > m)$;

$(*)$ Reorder of e_i and $e_j (i < j)$ requires also moving before e_i those events between e_i and e_j that e_j depends on. Hence if e_j depends on events $\langle e_{i+j_1}, e_{i+j_2}, \ldots, e_{i+j_k} \rangle$ then all the events in $\langle e_{i+j_1}, e_{i+j_2}, \ldots, e_{i+j_k}, e_j \rangle$ are reordered before e_i.

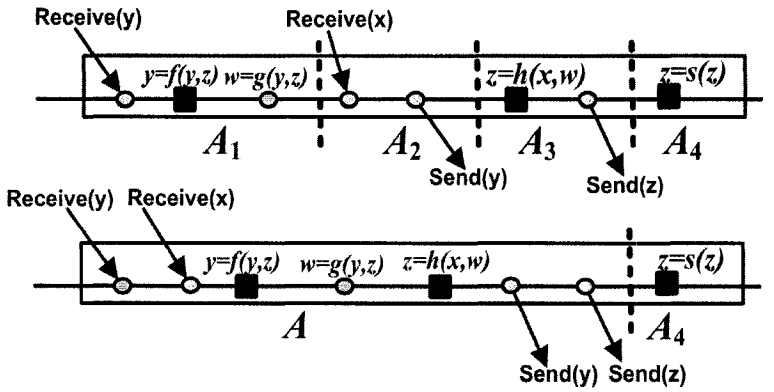

Figure 27: Event reordering to grow a maxi atom

before it. As a result of this reordering, the maxi atom has extended to the $Send(y)$ event. In the next iteration, reordering is possible between the independent events $Send(y)$ and $z = h(x, w)$, leading to atom A in Figure 27. In the third iteration, since the relevant event $z = s(z)$ depends on the previous event $Send(z)$, reordering is not possible. Now the maxi atom A is reported and the event pointer i is advanced to point to the event $z = s(z)$ for the next maxi atom.

As a result, the atomization with reordering of independent events results in a smaller set of atoms to be monitored and checked. The correctness of Algorithm 6 follows directly from dependency theory and will not be further elaborated.

6.2.3 Order Equivalence

As we have already demonstrated, there are two different granularities of atomization: maxi atoms can involve multiple relevant events and mini atoms can involve only one relevant event. Certainly, it is beneficial to consider the former that involves fewer atoms. However, it is important to analyze what one can deduce using the former rather than the latter. We will show an important order equivalence theorem here. First we establish the following lemma.

Lemma 31 *If e_r and $e'_{r'}$ are two of the relevant events in well-formed atoms E and E' respectively and $E \xrightarrow{c} E'$, then $e_r \rightarrow e'_{r'}$.*

Proof. From Definition 12, $E \xrightarrow{c} E'$ implies there exists e_i in E and e'_j in E' such that $e_i \rightarrow e'_j$. In case multiple such candidates exist, take the latest such event in E as e_i and the earliest such event in E' as e'_j. e_i must be the last event or a send event in E and e_j must be the first event or a receive event in E'. Since E and E' are well-formed atoms, we must have $e_r \rightarrow e_i$ and $e'_j \rightarrow e_{r'}$. Hence $e_r \rightarrow e'_{r'}$. ■

Consider two maxi atoms E and E' and the sequence of mini atoms $\langle E_1, \ldots, E_k \rangle$ and $\langle E'_1, \ldots, E'_{k'} \rangle$ contained in them. The following order equivalence theorem is established.

Theorem 32 $E \xrightarrow{c} E' \equiv E_i \xrightarrow{c} E'_j$ $[i \leq k$ *and* $j \leq k']$.

Proof. ($\Rightarrow$): From Lemma 31, $E \xrightarrow{c} E'$ implies the relevant events of any two mini atoms in E and E' are ordered under the happened-before relation $\rightarrow$. From the definition of $\xrightarrow{c}$, the latter mini atoms are ordered under $\xrightarrow{c}$.
($\Leftarrow$): This follows immediately from the definition of $\xrightarrow{c}$. ■

The theorem asserts that ordering requirements between mini atoms can be checked by examining ordering requirements between maxi atoms. This property is useful in reducing the cost of checking the ordering requirements among atoms.

6.3 Property Checking

There are two separate aspects in modeling the correctness of an atomized run. One aspect is related to the required synchronization (ordering) among the atoms in the

run. The other aspect is related to the actual changes of the relevant variables in each atom. We refer to the former as the ordering requirement and to the latter as the computational requirement.

In this section we will present a simple model to specify ordering requirements and an efficient algorithm to check the satisfaction of the ordering requirements in a given atomized run. We will also present a model to specify and to check the correctness of the computation performed by each atom. A more sophisticated model to specify ordering requirements will be presented in Section 6.5.

6.3.1 Ordering Requirements Specification and Checking

In this subsection we will use a simple representation of a POMSET that is based on a set of recurrent sequences $\hat{S} = \{S_1, S_2, \ldots, S_n\}$. Each recurrent sequence S_i is of the form

(a) $[A_1; A_2; \ldots; A_k]^*$ or

(b) $[A_1 + A_2 + \ldots + A_k]^*$ where A_i is in $2^{\bar{A}}$.

Form (a) uses the sequence operators ";". For any two atoms A and B, $(A; B)$ means that atom A should be causally ordered before atom B in any run (i.e., $A \xrightarrow{c} B$). The "*" denote the repetition operator; it indicates that the sequence can be repeated any number of times.

Each A_i corresponds to the label of an atom in the POMSET. Semantically, $[A_1; A_2; \ldots; A_k]^*$ represents the *recurrent* sequence $A_1; A_2; \ldots; A_k; A_1; \ldots; A_k; \ldots$ *etc.* and the ordering in the sequence (denoted by ";") corresponds to the causally-ordered relation between atoms. Hence form (a) recurrence represents a fixed ordering among a recurrent set of atom labels. In other words, all occurrences of atoms in the POMSET whose labels belong to the set $\{A_1, A_2, \ldots, A_k\}$ must be ordered according to

$A_1 \xrightarrow{c} A_2 \xrightarrow{c} \ldots \xrightarrow{c} A_k \xrightarrow{c} A_1 \xrightarrow{c} A_2 \xrightarrow{c} \ldots$ *etc.*

Form (b) recurrence $[A_1 + A_2 + \ldots + A_k]^*$ uses "+" as a dynamic choice operator among the atoms whose labels belong to the set $\{A_1, A_2, \ldots, A_k\}$. As a result, all occurrences of atoms in the POMSET whose labels are in $\{A_1, A_2, \ldots, A_k\}$ must be serialized in arbitrary order. For example, $A_1 \xrightarrow{c} A_3 \xrightarrow{c} A_k \xrightarrow{c} A_2 \xrightarrow{c} A_1 \ldots$ is a possible serialization. So "+" allows for dynamic ordering of atoms whereas ";" allows for fixed ordering of atoms. It should be noted that A_i need not be distinct. For example, $S_i = [X; Y; X; Z]^*$ is a valid recurrent sequence where atom X appears two times.

We assume the following additional condition on the set of recurrent sequences $\hat{S}$ to form the POMSET:

(c) If S_i contains atom X then S_i must generate all occurrences of atom X in the POMSET.

Condition (c) is not a necessary condition but it allows for a simpler presentation of the relevant ideas and procedures developed in this chapter.

Definition 25 *An atomized run is an* ***allowable run*** *with respect to a set of recurrent sequences $\hat{S}$ if the ordering among its atoms satisfies the ordering specified by $\hat{S}$.*

As an example, $\hat{S} = \{S_1 = [X; A; Y; B; Y]^*, S_2 = [X; C; Y; D; Y]^*\}$ is satisfied by both atom slices (POMSETs) shown in Figure 28(a) and (b). In both POMSETs, the ordering among the atoms satisfies the requirements in S_1 and S_2, although the one in Figure 28(b) contains more ordering (such as $C \xrightarrow{c} A$) than is required. However, the atom slice in Figure 28(c) fails $\hat{S}$; $C \xrightarrow{c} Y$ is required by S_2 but is not satisfied by the slice.

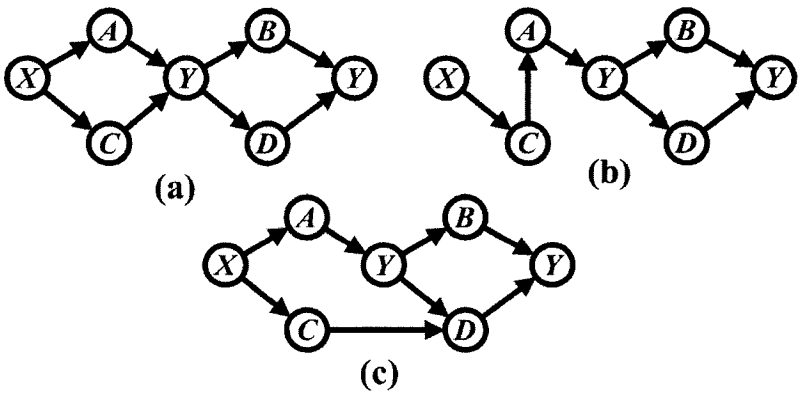

Figure 28: (a) An atom slice that satisfies $\hat{S}$, (b) An atom slice that is stricter than $\hat{S}$, (c) An atom slice that fails $\hat{S}$.

Algorithm 7 can be used to check if an atom slice $\Sigma_r = \langle 2^{\bar{A}}, \bar{E}_r, \xrightarrow{c}, L \rangle$ is an allowable run with respect to $\hat{S}$.

In the algorithm, consistency between $L(E)$ and current(S_i) involves two conditions:

i. $L(E) = \text{current}(S_i)$.

ii. $E' \xrightarrow{c} E$ in Σ_r where E' is the atom that corresponds to previous(S_i); the atom before the current atom in S_i.

As an illustration, consider the atom slice in Figure 28(c). The iteration proceeds as follows:

i. $\bar{E}_r^0 = \{X\}$ and X is consistent with both S_1 and S_2. So advance S_1 and S_2 to A and C respectively. Now current(S_1) = A, and previous$(S_1) = X$. X will be removed from the atom slice along with all of its outgoing edges.

ii. $\bar{E}_r^0 = \{A, C\}$. Consider A first. A is not in S_2 so we need to consider only S_1. Consistency is checked: (i) current$(S_1) = A$ and (ii) previous$(S_1) = X$ and

Algorithm 7 Checking if an atom slice Σ_r is an allowable run with respect to $\hat{S}$.

Input: Atom slice $\Sigma_r = \langle 2^{\bar{A}}, \bar{E}_r, \xrightarrow{c}, L \rangle$ and a set of recurrent sequences $\hat{S} = \{S_1, \ldots, S_n\}$.

current(S_i) = the first atom label in sequence S_i;
$\bar{A}(S_i)$ = the set of labels of atoms in S_i
$//L(E)$ is the label of atom E

repeat
  $\bar{E}_r^0$ = the largest subset of atoms in Σ_r without any atom ordered-before them;
  for each atom E in $\bar{E}_r^0$ **do**
    for each S_i such that $L(E) \cap \bar{A}(S_i) \neq \Phi$ **do**
      if $L(E)$ is consistent with current(S_i) **then**
        advance S_i;
      else
        report failure;
      end if
    end for
    $\Sigma_r = \langle 2^{\bar{A}}, \bar{E}_r - \bar{E}_r^0, \xrightarrow{c}, L \rangle$;
  end for
until Σ_r is an empty set or a failure is reported;

$X \xrightarrow{c} A$. Advance S_1. Similarly C is checked and consistent. Advance S_2. A and C will be removed from the atom slice along with all of its outgoing edges.

iii. $\bar{E}_r^0 = \{Y\}$. (i) Y is equal to both current(S_1) and current(S_2). (ii) previous$(S_2) = C$ but $(C \xrightarrow{c} Y)$ is not satisfied. So a failure is reported.

6.3.2 Specification and Checking of Computational Requirements

If the atoms are properly ordered, errors can still occur if the computation performed within an atom is erroneous. This aspect of an atom can be specified using an invariant on the state reached upon execution of each atom, without taking into account execution of other atoms that are not ordered-before it. This is our basis for reasoning about the correctness of the coding of an atom, assuming that the ordering between atoms is satisfied separately as modeled in Section 6.3.1.

The minimal state reached upon execution of an atom E is the state reached by the ***minimal*** prefix of the run $\Sigma = \langle 2^{\bar{A}}, \bar{E}, \xrightarrow{c}, L \rangle$ that contains E. This minimal prefix is denoted by Σ_E and contains all the atoms that are causally-ordered before atom E. These are formally defined as follows.

Definition 26 *The **minimal state** associated with a relevant atom E is represented by the values of the relevant variables attained at Σ_E. This minimal state is denoted by $V(\Sigma_E)$. For example, in Figure 23, $\Sigma_{E_{5b}}$ includes the minimal set of atoms $\{E_1, E_3, E_{5a}, E_{5b}\}$ that are causally-ordered before E_{5b} and $V(\Sigma_{E_{5b}})$ = the values of the relevant variables in $\{x, u, y, z\}$ attained upon execution of these atoms.*

The modeling of computational correctness of an atom is represented by an invariant that should be satisfied at each minimal state of each instance of the atom

in the atom slice $\Sigma_r = \langle 2^{\bar{A}}, \bar{E}_r, \xrightarrow{c}, L\rangle$. The invariant I is represented by a general predicate on $\bar{A}$. For example, $I(x, y, u, z) \equiv (x + y \leq c) \wedge (x = u = z)$, and I is satisfied at Σ_E iff $I(V(\Sigma_E)) = true$.

It is rather apparent that the computational correctness of atoms can be checked in simple polynomial time (with respect to the number of atoms in the run). Indeed, if the (relevant) atom slice contains k relevant atoms, only k states need to be checked, without any restriction on the property to be checked. Moreover, the number of relevant atoms can also be much smaller than the number of atoms or events in a run.

6.4 Example: Resource Management Application

In this section we will present a simple example to illustrate the model used to specify both of the ordering and computational requirements. This example involves three processes, P_1, P_2 and P_3. P_1 is the main manager, P_2 and P_3 are local managers. The total number of resources under management is a constant (say c). A simple request-acknowledge protocol is maintained between the main manager and each local manager. A local manager can request a change of allocation and will be acknowledged by the main manager at any time. The allocations to the three managers are represented by local variables x, y and z respectively. Two additional relevant variables, $y.transit$ and $z.transit$, are needed for representing the change of allocations to y and z sent by P_1 (and in transit through the corresponding message channel). The system is represented by the set of relevant atoms described in Table 2.

Based on the above description of the application, the ordering requirements that must be satisfied by a run are specified by two recurrent sequences of form (a) and

Table 2: Relevant atoms executed by the resource management application.

Agent	**Atom** (Relevant Variables)	**Purpose**
P_1	$\{x, y.transit\}$	re-allocates x and y
P_2	$\{y\}$	accepts change of y
P_1	$\{x, z.transit\}$	re-allocates x and z
P_3	$\{z\}$	accepts change of z

one recurrent sequence of form (b) as follows:

$$\hat{S} = \{[\{x, y.transit\}; y]^*, [\{x, z.transit\}; z]^*, [\{x, y.transit\} + \{x, z.transit\}]^*\}$$

The recurrent sequence $[\{x, y.transit\}; y]^*$ requires that the change of allocation at P_1 is ordered before the corresponding change at P_2. The dynamic recurrent sequence $[\{x, y.transit\} + \{x, z.transit\}]^*$ requires that the changes of allocation at P_1 must be serialized, one at a time. However, the serialization is decided at runtime, depending on the requests arriving from P_2 and P_3.

The computational correctness of each atom is specified by the satisfaction of the following invariant at each minimal state:

$$I(x, y, z, y.transit, z.transit) \equiv [x + y + z + y.transit + z.transit = c].$$

This invariant simply asserts that upon completion of each atom, the sum of all allocations (including those in transit) must be equal to c.

Figure 29 shows an atom slice of a partial run that satisfies the required ordering in $\hat{S}$. Only relevant atoms in the slice are depicted (rectangles). Events outside the rectangles belong to non relevant atoms. In particular, the changes of allocation

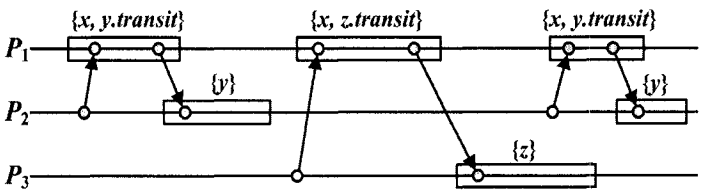

Figure 29: Resource management example.

at P_1 are dynamically serialized while the corresponding changes in P_2 and P_3 are properly ordered according to $[\{x, y.transit\}; y]^*$ and $[\{x, z.transit\}; z]^*$ respectively. The checking of computational correctness of each atom can be performed by checking the invariant I at each minimal state associated with each atom shown in Figure 29.

6.5 A POMSET Automaton to Specify Ordering Requirements

In this section we will present a more sophisticated way to specify the ordering requirements that a distributed program must satisfy.

In each execution of a given distributed program, there are necessary ordering and unnecessary (accidental) ordering between atoms. A necessary ordering between two atoms means that they must always occur in a certain order. An unnecessary ordering between two atoms means that they can occur in any order or simultaneously.

As a result, a binary relation can be defined between two executions: two executions are related if and only if they do not differ in the necessary orderings among the atoms in them. This binary relation is an equivalence relation (i.e., it is reflexive, symmetric and transitive). This relation induces equivalence classes on the set of possible executions of a distributed program. Each such equivalence class is called

a ***behavior*** of the distributed program. Such a behavior is represented by a partial order on the events, where the partial order represents the necessary ordering [LL94]. The formal tool for such a representation is a POMSET (partially ordered multiset) [Pra86].

Definition 27 *A **behavior** of a distributed program is represented by a POMSET* $\Sigma = \langle A, E, \Gamma, L \rangle$ *where*

i. *A is a set of relevant atoms.*

ii. *E is the set of events. Each event corresponds to the execution of a relevant atom.*

iii. *$\Gamma : E \rightarrow E$ is the partial order mapping among the events in E. Γ can be viewed as a set of pairs, a pair of events $(E_1, E_2) \in \Gamma$ if $E_1 \xrightarrow{c} E_2$.*

iv. *$L : E \rightarrow A$ is a labeling function that maps events in E to atoms in A, each event in E is an instance of some atom in A.*

There is an arrow from event e to event f in POMSET $\langle A, E, \Gamma, L \rangle$ if $(e, f) \in \Gamma$. The partial order Γ is represented in a transitively reduced form, i.e., if there are arrows from event e to event f and from event f to event g, then the arrow from event e to event g will not be represented explicitly.

The ***specification*** of a distributed program is the set of behaviors (POMSETs) of the distributed program.

To finitely represent the set of infinite behaviors of a distributed program, a POMSET generator, called ***POMSET automaton*** is used. In this section we adopt the use of this automaton as a model of the ordering requirements that must be satisfied

by a given distributed program. The tool developed by other members of our research group accepts the ordering requirements specified by a POMSET automaton.

The POMSET automaton described here is an adapted version of a POMSET generator called *behavior machine* introduced by Probst and Li [PL91b, PL91a, PL93, LL94].

The POMSET automaton has a finite number of *slots*. A non-empty subset of these slots represents the set of *initial slots*. There are *transitions* that can be fired when a certain subset of slots is satisfied and as a result a subset of slots will be reached (satisfied). A slot represents a condition that must be satisfied before a transition can be fired or will be reached (satisfied) when a transition is fired. Each transition is a finite POMSET denoting the behavior of a distributed program when there is a transition from the subset of slots preceding the transition to the subset of slots following it.

A behavior of a distributed program is generated by concatenating transitions (POMSETs) of the POMSET automaton associated with the program, starting from a transition following the initial subset of slots. A transition T can be concatenated to a behavior b if a subset of the slots, say s, following the behavior b is the same as the subset of slots preceding the transition T. The subset of slots s is for expressing causalities between events from the behavior b and events from the transition T. If event e in b precedes a slot L, and event f in T is preceded by slot L, then, after concatenation, event e will precede event f.

The POMSET automaton of a given distributed program may have more than one transition emanating from a subset of slots. Such a subset of slots is called a choice subset. A transition ending at a choice subset can concatenate with any one of the transitions following the choice subset. This property ensures that more than

one behavior can be generated by the automaton.

We assume a further restriction: Let C be the set of condition slots reached any time during unfolding (behavior generation). The set C will be partitioned into *disjoint subsets* each of which owns a set of transition rules (possibly a singleton if there is no choice in the transition and empty if no transition rule is applicable). We refer to this assumption as the ***disjoint choice assumption.***

Now we will present a simple example to illustrate the above concepts. A more comprehensive example will be presented in the next section.

We will consider a simple distributed multi-agent application (product buyer) that involves three agents, one buyer agent $Buyer_1$ and two seller agents $Seller_1$ and $Seller_2$.

Seller agents keep a catalogue of the products they are selling along with the related information such as price, production date, etc. When the buyer wants to buy a product, he will send a call for proposals to the seller agents and wait for a reply from them. When a seller agent receives a call for proposals, he will reply to the buyer agent with a message containing the product information or he will refuse the proposal if he is no longer selling the required product. When the buyer agent receives all the replies from the seller agents, he will apply his criteria to choose one seller. If none of the seller agents' proposals satisfy the criteria, the buyer will send again a call for proposals to all of the sellers. When a seller agent is selected, the buyer agent will send him a purchase order, the seller will receive the purchase order, and if the product is still available, he will update his catalog and send a confirmation to the buyer. Otherwise he will send a refuse message to the buyer. If the buyer receives a refuse message, he will send again a call for proposals to all of the sellers. This process will continue until the buyer agent buys his product.

According to the above description, a buyer agent can execute one of the following atoms:

1. Send a call for proposals (we will call it atom A).
2. Receive the replies from the sellers, process it and take the necessary action. For example, if all the replies are "Refuse" messages, the buyer will send a call for proposals (atom B).
3. Receive the reply of a purchase order, process it and take the necessary action (atom C).

Each seller agent can execute one of the following atoms:

1. Receive a call for proposals, process it and send the suitable reply (Atom D).
2. Receive a purchase order, process it and send the suitable reply (Atom E).

Figure 30 shows an example run of the described multi-agent application.

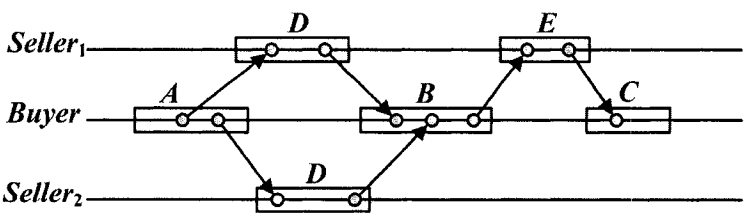

Figure 30: An example of an atomized run of a simple distributed multi-agent application "product buyer".

The ordering requirements of this application can be simply described by a POMSET automaton. Figure 31 shows the POMSET automaton that models the ordering requirements of this simple application. We will use circles to depict slots and squares to depict atoms. The arrows represent the necessary orderings among the atoms.

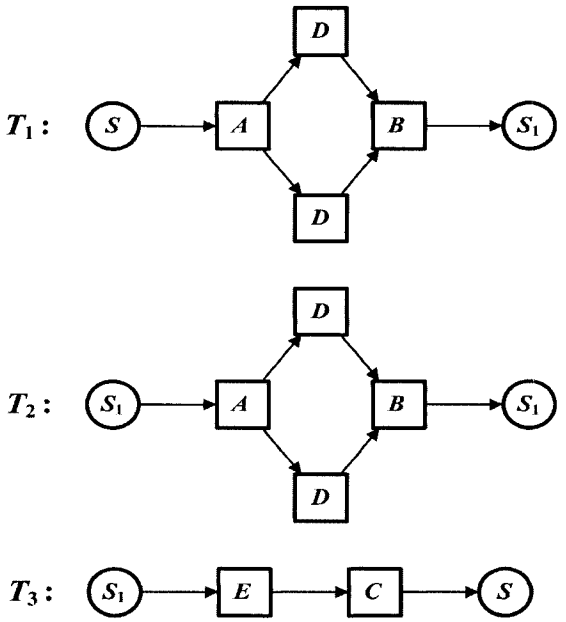

Figure 31: The POMSET automaton used to model the ordering requirements of the product buyer application.

There are two slots $\{S, S_1\}$ and three transitions $\{T_1, T_2, T_3\}$ in the POMSET automaton shown in Figure 31. S_1 is a choice slot, when it is reached T_2 or T_3 can be fired. The example run shown in Figure 30 can be produced by firing transition T_1 then transition T_3. In general, a distributed program run $\Sigma = \langle A, E, \Gamma, L \rangle$ is admissible by a POMSET automaton M if there exists a prefix $\Sigma' = \langle A', E', \Gamma', L' \rangle$ of a behavior generated by M such that

i. $A = A'$,

ii. $E = E'$,

iii. $\Gamma' \subseteq \Gamma$, and

iv. $L = L'$.

In other words, Σ contains all the ordering of the prefix of a behavior Σ' generated

by M.

In this example we assume that the number of agents involved in the application is fixed; however, it is not the case in real life applications. In the following section we will present a more detailed example to demonstrate the property checking methodology for distributed programs that has been introduced in this chapter.

6.6 Example Usage

To help the programmers quickly and effectively figure out how to use the methodology presented in this chapter, this section will serve as a guideline on how to identify atoms and to specify ordering and computational requirements of a given distributed program. The methodology will be demonstrated through an illustrative example of a multi-agent E-market application. The ultimate goal of this section is to give the user of the checking tool an example of how he can prepare the input files necessary to use the tool.

6.6.1 A Brief Description of the E-market Application

There are four types of agents $\{C, B, W, A\}$ playing different roles in this application.

1. C : A client role is to conduct a purchase with the help of a broker and an accounts manager.

2. B : A broker role is to broker a purchase for a client with a wholesaler.

3. W : A wholesaler is to sell an item through a broker and to inform the accounts manager to complete the transaction (receipt of money and delivery of product).

4. A : An accounts manager completes a transaction with the client, as instructed by a wholesaler.

- Initially a client C (who wants to purchase a product P) will send a call for proposals to all of the brokers he knows $(B_1, \ldots, B_n)$.
- When a broker receives a call for proposals from Client C, he will send a call for proposals to all of the wholesalers he knows $(W_1, \ldots, W_m)$.
- When a wholesaler receives a call for proposal from a broker B, he will check his catalogue to see if product P is available, if yes, the wholesaler will send a propose message to the broker. Otherwise the wholesaler will send a refuse message to the broker.
- If the broker receives a non empty set of proposals he will choose the best one (minimum price) and send it in a propose message to the client C. Otherwise he will send a refuse message to the client.
- If the client receives a non-empty set of proposals from the brokers known to him, he will choose the best one and he will send a purchase order to the broker who has sent him the best proposal. Otherwise the client will send another call for proposals.
- When the broker receives a purchase order, he will forward it to the corresponding wholesaler.
- When the wholesaler receives a purchase order, he will check his catalogue and send an inform message to the accounts manager.
- When the accounts manager receives an inform message from a wholesaler, he will send a bill to the corresponding client if the purchase order has been

approved by the wholesaler, otherwise he will send the client a failure message.

- If the client receives a failure message from the accounts manager, he will send another call for proposals.
- When the client receives a bill from an accounts manager he will send the specified amount to the accounts manager.
- When the accounts manager receives the payments he will send a confirmation to the client.

6.6.2 Input

Now we will start describing the required input that is needed in order to be able to identify atoms and to specify ordering and computational requirements.

1. Design Time Knowledge: A set of interaction diagrams that capture the interaction patterns between agents will be very helpful to the programmer to identify atoms and ordering requirements. The starting point for developing the interaction diagrams is a set of use-case scenarios that describe at a high level the steps the application will follow to accomplish the desired functionality. Figure 32 shows the interaction protocols diagram for the E-market application described in Section 6.6.1. This diagram can assist the programmer in identifying the atoms and specifying the ordering requirements as it will be demonstrated in this section.

2. We need the source code of the application; this will be used by the programmer to identify the relevant variables. There will be two types of relevant variables in a given application; variables relevant to the ordering requirements and variables

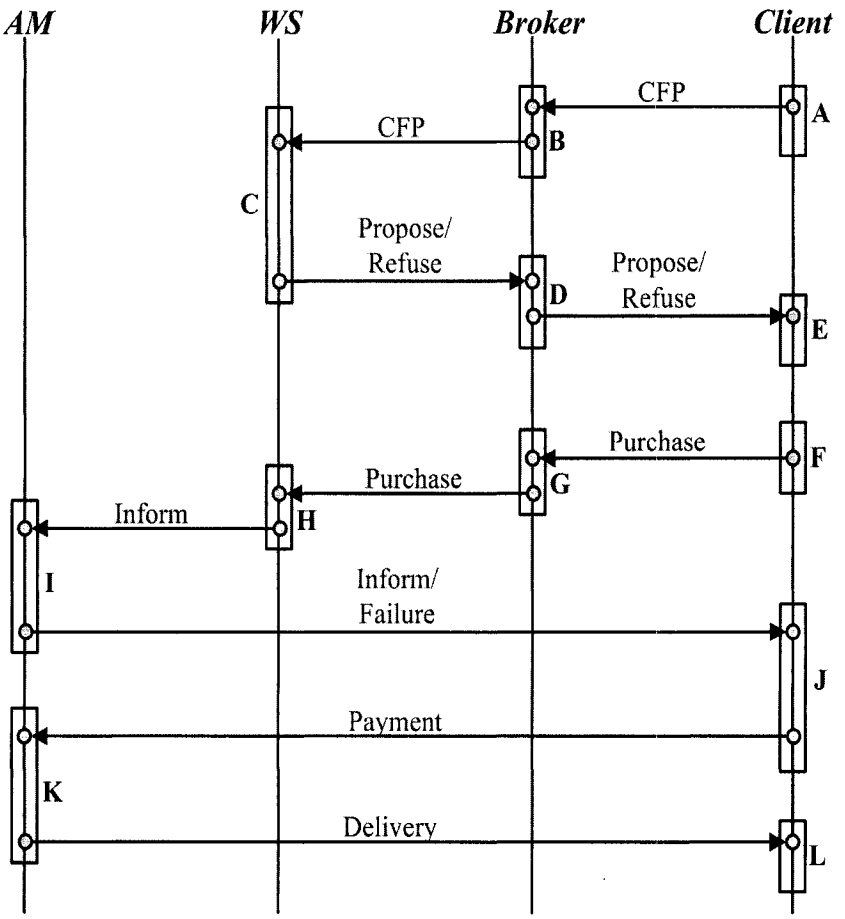

Figure 32: The atomized interaction protocols diagram of the E-market application.

relevant to the computational requirements. Examples of ordering requirements, computational requirements and relevant variables are presented later in this section.

6.6.3 Process

Given the inputs described in the previous section, we will describe the steps that the programmer will go through to identify atoms and specify the ordering and computational requirements.

6.6.3.1 Step 1: Atomization

The programmer can identify the atoms that can be executed by an application through constructing the atomized interaction diagram. This can be done based on the description of the application given in section 6.6.1 and the interaction diagram. The atomized interaction diagram for the E-market application is shown in Figure 32. Rectangles are used to depict atoms, each atom has a corresponding label.

Once this done, the programmer will have the set of all atoms that can be executed by the application. Table 3 lists the set of atoms that can be executed by the E-market application along with a brief description of the functionality performed by each atom.

Figure 33 shows an example of an atomized run of the E-market application. In this run we have assumed that there are two brokers B_1 and B_2 known to client C. Broker B_1 can contact wholesalers W_2 and W_3, while broker B_2 can contact all of the wholesalers. In the first round of the run the client sent a call for proposals to the brokers but he did not receive any proposal, so he sent another call for proposals, and this time he has received some proposals, the client proceeded and sent a purchase order. The run continued according to the described interaction protocols until the client received the purchased product.

6.6.3.2 Step 2: Specification of Ordering Requirements

In this section we will specify the ordering requirements among the atoms, identified in step 1, using the POMSET automaton model. The atomized interaction protocols diagram shown in Figure 32 will assist the programmer in specifying the ordering requirements properly. We assume that there are i Brokers known to the client, and for each broker there are j wholesalers known to him. The POMSET automaton shown in Figure 34 specifies the ordering requirements that must be maintained

Table 3: The set of atoms executed by the E-market application.

Agent	**Atom**	**Description**
Client	A	Sending a call for proposals to brokers.
Broker	B_i	The i^{th} Broker receives the client call for proposals and sends a call for proposals to wholesalers.
Wholesaler	C_{ij}	The j^{th} wholesaler of the i^{th} broker receives the broker request, if the product requested is available, the wholesaler will send a proposal for the broker; otherwise he will send him a refusal message.
Broker	D_i	In this atom, the i^{th} broker receives all the proposal/refusal messages from the wholesalers and sends a proposal/refusal message to the client accordingly.
Client	E	The client receives all the proposal/refusal messages from the brokers and selects the best proposal if any.
Client	F	The client sends a purchase message to the broker who has provided the best proposal.
Broker	G	The broker receives a purchase order from the client and forwards it to the corresponding wholesaler.
Wholesaler	H	The wholesaler receives a purchase order from the broker, makes sure that the product is available, if yes he will approve the purchase order, otherwise he will reject it, in any case he will forward the result to the account manager
Account Manager (AM)	I	The account manager will receive the purchase order, if it is approved, he will send an invoice to the client, otherwise he will send a Failure message to the client.
Client	J	The client receives a message from the AM, this message can be an inform or a failure message. If it is a failure message then the client will make another call for proposals in the next atom. If the message is an inform message, the client will pay the amount specified in the invoice and will send a proof of payment to the AM. (The payment issues are not handled in the application)
AM	K	The AM receives the payment proof and sends a confirmation to the client.
Client	L	The client receives the confirmation.

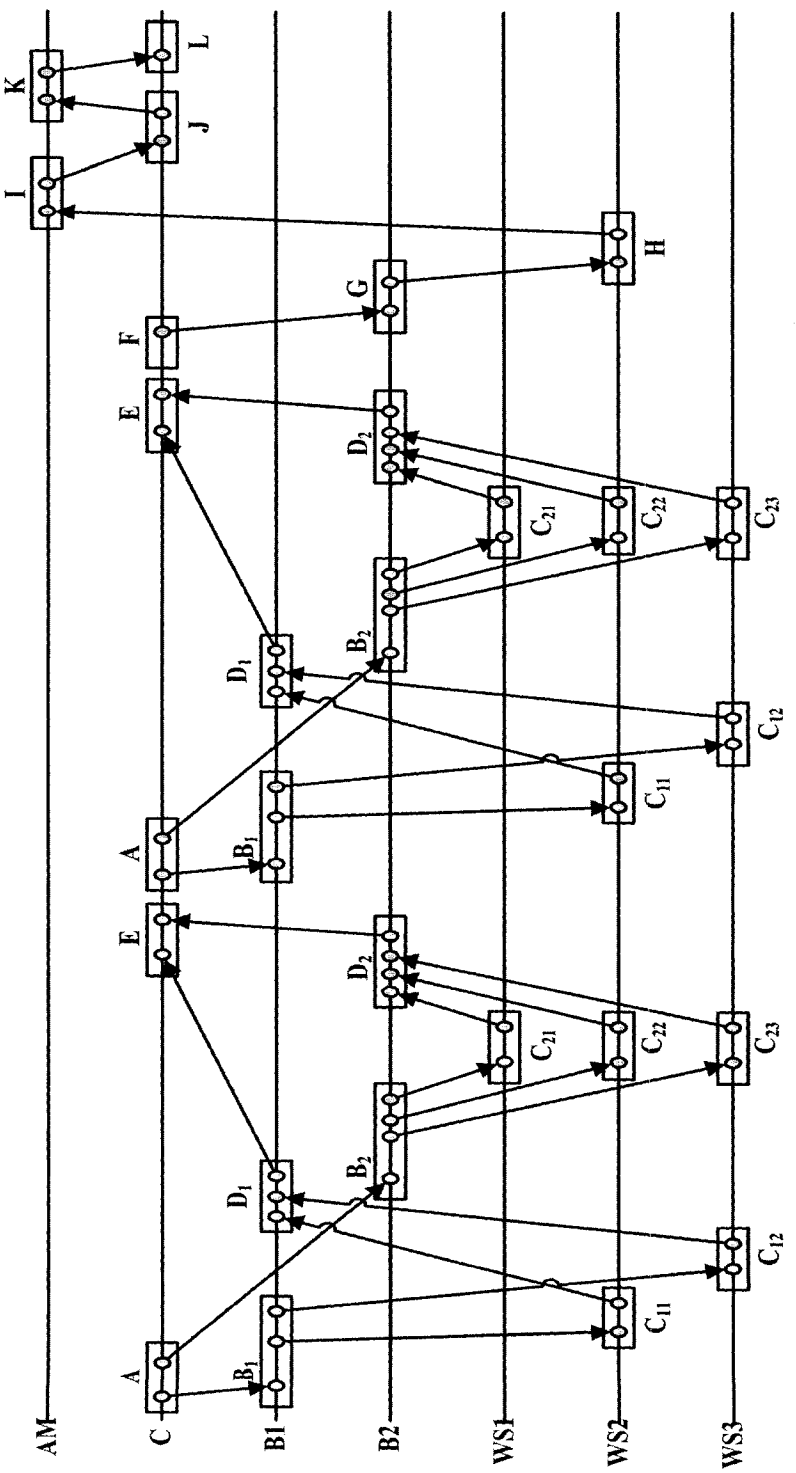

Figure 33: An example of an atomized run of the E-market application.

among the atoms executed by the E-market application. There are 8 transitions in this POMSET automaton. The three vertical dots appearing in some of the transitions indicates that some of the atoms or slots are not depicted in the transition. For example, in transition T_1, there are three vertical dots between atoms B_1 and B_i, this indicates that transition T_1 will involve also atoms $\{B_2, \ldots, B_{i-1}\}$. Moreover, there is an edge between atom A and each atom $B_j \in \{B_2, \ldots, B_{i-1}\}$, and there is an edge from each atom $B_j \in \{B_2, \ldots, B_{i-1}\}$ to a slot S_j, as it the case for the two atoms B_1 and B_i that are already depicted in transition T_1.

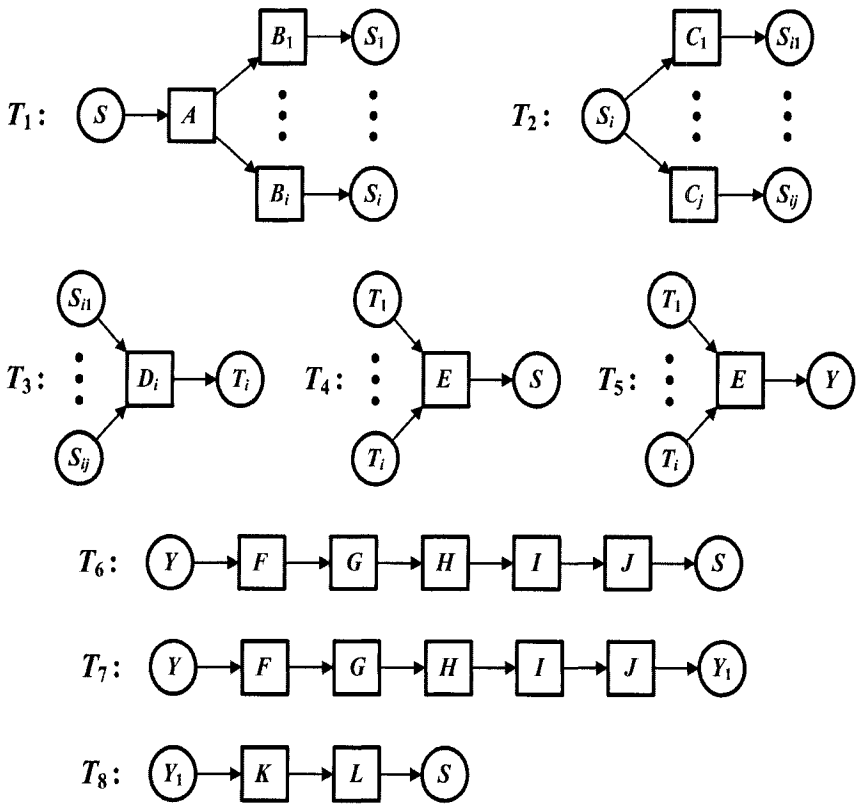

Figure 34: The POMSET automaton of the E-market application.

6.6.3.3 Step 3: Identifying Relevant Variables

In order to be able to track at runtime which atoms are being executed we need to map the abstract atoms of a protocol into the set of relevant variables (such as the "price" and the "product type" in our E-market application) used in the source code. The set of relevant variables used to label the atoms in this step represents the set of the variables relevant to the ordering requirements. At runtime, the events triggered by any modification to any of these variables will be used to identify the atom currently being executed and report its label and timestamp to the monitor. The collected data will be used later by the checker to check the ordering requirements specified in the previous section.

6.6.3.4 Step 4: Computational Requirements Specification

In this step we need to identify the predicate that captures the correctness of the incremental function performed by each atom. For example, at the end of each instance of atom D, the selected proposal should be the one with the minimum price. The set of variables used in these predicates represent the set of variables relevant to the computational requirements. Any modification to any of these variables must be reported to the monitor along with its time stamp and the new values of the relevant variables that have been modified.

6.6.4 Output

After completing the process described above, the programmer we will have the following items that are needed to check the ordering and the computational requirements of a given application.

1. The set of labeled atoms

2. The POMSET automaton that specifies the ordering requirements among relevant atoms of a correct run that can be checked against the actual run of the application

3. The atom predicates that specify the computational correctness of each atom

6.7 Summary

Atoms are useful abstractions in reducing the size of the state lattice of a distributed computation. Maxi atom is a suitable abstraction whereby a process advances locally as if its effect to everyone else occurs only upon completion of the atom. Relevance to the required program properties is applicable to further abstract a run into an atom slice for property checking.

Partially ordered multi-set (POMSET) models of distributed programs model true concurrency, as a result they are attracting more attention in modeling and analyzing distributed programs. This chapter proposes a distributed programs checking methodology that views the distributed computation as a partially ordered multi-set of atoms.

The POMSET model promotes the separation of two different concerns in checking the correctness of a distributed computation. Firstly, the ordering between the atoms needs to be specified and checked. Secondly, the correctness of the computation performed by each atom needs to be checked. In this chapter we have presented a methodology to specify and to check both of the ordering and computational requirements of a distributed computation.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Checking whether a given distributed program computation satisfies a given property or not is a fundamental problem that has applications in the testing and debugging of distributed programs. However, due to concurrency, the number of global states that need to be examined when checking the properties of distributed programs can be exponential. In this thesis, various techniques have been proposed to alleviate the state explosion problem in distributed programs dynamic property checking.

The concept of atoms has been proposed as an effective state space reduction mechanism. It has been demonstrated through examples that atoms can significantly reduce the number of states that need to be considered in property checking. This idea has been illustrated by showing the difference between the number of states in the atomic state lattice L^* and its corresponding original state lattice L. An efficient on-the-fly distributed algorithm to check the atomicity of a given distributed computation has been presented along with its proof of correctness. The algorithm

can detect atomicity violations without the need to propagate the probes through all of the atoms involved in a cycle. Consequently, we conclude that it is a viable approach to use atoms in reducing the state space for property checking. Property checking in this case will involve two steps. First, the atomicity of the run will be checked; if it is atomic then we can move to the second step where the required properties can be checked on the reduced lattice. If the run is not atomic then we conclude that the program contains some bugs, and hence the reported cycle can help the programmer locate and fix the bug. Alternatively, we could conclude that the programmer has incorrectly labeled the code blocks associated with some atoms.

A programmer may incompletely or incorrectly identify the set of atoms that can be executed by a given distributed program. The consequences of incomplete and/or incorrect knowledge of atoms on the conclusions made about the atomicity of a distributed program run and the satisfaction of the desired properties has been studied. We determine that the conclusion made about the atomicity of a run is not affected by incomplete knowledge of atoms and the reduced state lattice can be used to check the satisfaction of the desired properties by that run. However, the maximum state space reduction can be achieved only with a complete set of atoms. If some atoms are incorrectly identified, then the conclusion made about the satisfaction of a property ϕ remains valid if all of the incorrectly identified atoms are ϕ-independent.

Even with atomization, the size of the state space that needs to be considered in property checking can still grow exponentially with respect to the number of atoms. Some ideas for further state space reduction have been explored. When a set of processes have to maintain a shared property, we expect that at some point in time, processes in that set must be aware of the changes made by each other that may affect the property concerned. We conclude that properties need to be checked only at

synchronized states where the processes have already exchanged the information necessary to maintain the property. We have illustrated the idea through some examples. Synchronization can be modeled by a synchronization predicate. Serialization of the synchronized states is the minimal avenue to ensure that the relevant processes have exchanged the necessary information needed to maintain the concerned property.

Based on the above analysis, a property checking mechanism that involves three tasks has been proposed. A given distributed computation satisfies a property g if all of the synchronized states are serialized and g is true in each of them. Two efficient algorithms to check the satisfaction of a property g in a given distributed computation, where the synchronization predicate is conjunctive or disjunctive, have been presented along with their proof of correctness.

Since not every atom is relevant to the properties concerned, we have introduced the notion of an atom slice that contains the fewest number of atoms to be monitored and checked. For further state space reduction, independent events that belong to different atoms can be reordered to merge two or more atoms and consequently reduce the number of states that need to be considered in property checking. An efficient algorithm to atomize a distributed computation with reordering of independent events has been developed.

Finally, the concept of well-formed atoms and a partially ordered multi-set (POMSET) model of a distributed computation have been exploited to propose a checking methodology for distributed programs. According to this methodology, checking of distributed programs involves the following two steps: checking that the atoms executed by the program are properly ordered, and checking that the computations within each atom are performed correctly.

A POMSET automaton model has been used to specify the ordering requirements

that the atoms executed by a given distributed program must satisfy. The computational requirement that must be satisfied by each relevant atom is specified by a predicate that needs to be checked in the minimal prefix of each instance of the atom in a given distributed program computation. An example has been presented to demonstrate the methodology.

7.2 Future Work

Theses are never complete, however, at some point you just stop. Consequently, a number of avenues for continuation of this work can be suggested.

In our work we have advocated the use of dynamic on-the-fly analysis to detect atomicity errors, serialization errors, etc. As we have mentioned in Chapter 2, there are other approaches for distributed programs analysis, such as static analysis and dynamic off-line analysis. On-the-fly analysis has the advantage of early error reporting and avoiding the need for large trace files. However, it has the disadvantages of incurring more space and time overhead during execution. Off-line analysis of distributed programs does not incur significant space and time overhead during execution but it has the disadvantage of generating very large trace files. The work presented in this thesis can be extended by using the off-line analysis approach and comparing the results with the on-the-fly analysis approach.

In this thesis we consider atoms that involve a sequence of events belonging to a single process. This work can be extended to consider atoms that involve events belonging to more than one process. This may result in a more significant state space reduction. However, the atomicity error detection algorithm needs also to be extended and the reduced state space has to be clearly defined.

Finally, in Chapter 5 we have developed two algorithms to check the satisfaction of a property g in a given distributed computation, where the synchronization predicate p is conjunctive or disjunctive. The work in this chapter can be extended by considering other classes of synchronization predicates.

Bibliography

[AP98] Vangalur S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer-Verlag New York, Inc., 1998.

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer, New York, NY, USA, 2001.

[Bes90] Eike Best. Partial order semantics of concurrent programs. In *CONCUR '90: Proceedings on Theories of concurrency : unification and extension*, page 1. Springer-Verlag New York, Inc., 1990.

[Bou87] Luc Bougé. Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP. *Theor. Comput. Sci.*, 49:145–169, 1987.

[CBDGF95] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1):157–179, 1995.

[CG95] Craig M. Chase and Vijay K. Garg. Efficient detection of restricted

classes of global predicates. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 303–317. Springer-Verlag, 1995.

[CG98] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CM91] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *SIGPLAN Not.*, 26(12):167–174, 1991.

[Eme90] E. Allen Emerson. *Temporal and modal logic.* MIT Press, Cambridge, MA, USA, 1990.

[Esp93] Javier Esparza. Model checking using net unfoldings. In *TAPSOFT '93: Selected Papers of the Colloquium on Formal Approaches of Software Engineering*, pages 151–195, Amsterdam, The Netherlands, 1993. Elsevier Science Publishers B. V.

[FF04] Cormac Flanagan and Stephen Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, 2004.

[FIP] http://www.fipa.org/.

[Gar02] Vijay K. Garg. *Elements of distributed computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[GCKM97] V. K. Garg, C. M. Chase, Richard Kilgore, and J. Roger Mitchell. Efficient detection of channel predicates in distributed systems. *Journal of Parallel and Distributed Computing*, 45(2):134–147, 1997.

[GLM07] Patrice Godefroid, Michael Y. Levin, and David Molnar. Active Property Checking. Technical Report MSR-TR-2007-91, Microsoft Research, 2007.

[GM01] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 322, 2001.

[GMS06] Vijay K. Garg, Neeraj Mittal, and Alper Sen. Using order in distributed computing. In *American Mathematical Society National Meeting*, 2006.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gou00] Eric Goubault. Geometry and concurrency: a user's guide. *Mathematical. Structures in Comp. Sci.*, 10(4):411–425, 2000.

[Gup94] Vineet Gupta. *CHU spaces: a model of concurrency.* PhD thesis, Stanford University, USA, 1994.

[GW92] Vijay K. Garg and Brian Waldecker. Detection of unstable predicates in distributed programs. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, London, UK, 1992. Springer-Verlag.

[GW94] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):299–307, 1994.

[GW96] Vijay K. Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1323–1333, 1996.

[HC96] G. Hughes and M. Cresswell. *A New Introduction to Modal Logic.* Routledge, London, 1996.

[HPR93] Michel Hurfin, Noel Plouzeau, and Michel Raynal. Detecting atomic sequences of predicates in distributed computations. In *Workshop on Parallel and Distributed Debugging*, pages 32–42, 1993.

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, pages 166–175, Los Alamitos, CA, USA, 1988.

[HW97] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *ESEC '97/FSE-5: Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 450–467. Springer-Verlag New York, Inc., 1997.

[JMN95] Roland Jegou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In Desel, J., editor, *Structures in Concurrency Theory, Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, 11–13 May 1995*, pages 175–189, 1995.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, 1981.

[Lad01] M. Al Ladan. A survey and a taxonomy of approaches for testing parallel and distributed programs. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 273–279, 2001.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam90] L. Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LL94] S. C. Leung and H. F. Li. A syntax-directed translation for the synthesis of delay-insensitive circuits. *IEEE Trans. VLSI Syst.*, 2(2):196–210, 1994.

[LM07a] H. F. Li and Eslam Al Maghayreh. Checking distributed programs with partially ordered atoms. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 518–525. IEEE Computer Society, 2007.

[LM07b] H. F. Li and Eslam Al Maghayreh. Using synchronized atoms to check distributed programs. In *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*, Hsinchu, Taiwan, 2007.

[LMG07a] H. F. Li, Eslam Al Maghayreh, and D. Goswami. Detecting atomicity errors in message passing programs. In *PDCAT '07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 193–200. IEEE Computer Society, 2007.

[LMG07b] H. F. Li, Eslam Al Maghayreh, and D. Goswami. Using atoms to simplify distributed programs checking. In *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 75–83, 2007.

[LRG04] H. F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engg.*, 11(1):63–89, 2004.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, October 1989.

[Maz87] Antoni W. Mazurkiewicz. Trace theory. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, pages 279–324, 1987.

[McM92] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1992. Springer-Verlag.

[MG01] Neeraj Mittal and Vijay K. Garg. Computation slicing: Techniques and theory. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 78–92, London, UK, 2001. Springer-Verlag.

[MI92] Yoshifumi Manabe and Makoto Imase. Global conditions in debugging distributed programs. *J. Parallel Distrib. Comput.*, 15(1):62–69, 1992.

[MR97] Albert R. Meyer and Alexander Rabinovich. A solution of an interleaving decision problem by a partial order technique. In *POMIV '96: Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*, pages 203–211, New York, NY, USA, 1997. AMS Press, Inc.

[Net91] Robert Netzer. *Race Condition Detection for Debugging Shared-Memory Programs.* PhD thesis, University of Wisconsin, USA, 1991.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[OBFR96] Özalp Babaoğlu, Eddy Fromentin, and Michel Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *J. Syst. Softw.*, 33(3):287–298, 1996.

[Pel96] Doron Peled. Partial order reduction: Model-checking using representatives. In *MFCS '96: Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, pages 93–112, 1996.

[PG04] Christoph Von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[PL91a] David K. Probst and H. F. Li. Partial-order model checking: A guide for the perplexed. In *CAV*, pages 322–331, 1991.

[PL91b] David K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 146–155, 1991.

[PL93] David K. Probst and H. F. Li. Verifying timed behavior automata with input/output critical races. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 424–437, London, UK, 1993. Springer-Verlag.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS : Proceedings of the 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[Pnu79] Amir Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, 1979.

[Pra86] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[SAWS05] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–94, 2005.

[SG02] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates on the happened-before model. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 116, 2002.

[SG03] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *OPODIS*, pages 171–183, 2003.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, pages 382–388, 1986.

[SLS05] Usa Sammapun, Insup Lee, and Oleg Sokolsky. Rt-mac: Runtime monitoring and checking of quantitative and probabilistic properties. In *In the Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 147–153, 2005.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[SS95] Scott D. Stoller and Fred B. Schneider. Faster possibility detection by combining two approaches. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 318–332, 1995.

[TG93] Alexander I. Tomlinson and Vijay K. Garg. Detecting relational global predicates in distributed systems. In *Workshop on Parallel and Distributed Debugging*, pages 21–31, 1993.

[TG98] Ashis Tarafdar and Vijay K. Garg. Predicate control for active debugging of distributed programs. In *IPPS/SPDP*, pages 763–769, 1998.

[Val91] Antti Valmari. A stubborn attack on state explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, 1991.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[WS06a] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, 2006.

[WS06b] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.

[WY04] Fangjun Wu and Tong Yi. Slicing Z specifications. *SIGPLAN Not.*, 39(8):39–48, 2004.