Design and Implementation of A Pomset Automaton Based Runtime Verifier for Distributed Jade Programs

Yan Liu

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 2008

Your file Votre référence
ISBN: 978-0-494-42533-6
Our file Notre référence
ISBN: 978-0-494-42533-6

Abstract

Design and Implementation of

A Pomset Automaton Based Runtime Verifier for Distributed Jade Programs

Yan Liu

Monitoring and checking the execution of a distributed program incur significant overhead due to the large number of states that need to be considered when using interleaving semantics to model the concurrency. In this thesis, we use partial order semantics in modeling a distributed computation. Specifically, a Pomset automaton model is used to specify all the allowable partial orders of a given design. The distinct aspects of requirement are separately modeled: the ordering requirements among atoms (a set of source code statements that are expected to be performed atomically) and the correctness of each atom. And atomization is introduced into the abstraction to map the correspondence between events in the design layer and events in code space. Therefore, ordering requirements are specified among the so called abstract atoms; then, these atoms in the design space are mapped into the code space using atom variables. The correctness of an atom is specified by using predicates on a state that is reached upon the completion of an atom. In our Pomset model, the ordering is explicit and easily checkable, which is different from the more traditional model checking.

The proposed methodology is mechanized in a runtime verification tool on a multi-agent platform (Jade) to demonstrate its effectiveness. The runtime verifier accepts a user-specified Pomset automaton and the associated atom predicates for a given Jade source program. Implemented in AspectJ, the verifier monitors and checks both ordering requirements and atom requirements on-the-fly, and echoes appropriate results to the user for debugging and testing.

Acknowledgments

I would like to thank my supervisor Dr. R. Jayakumar and Dr. H. F. Li for their continuous support and valuable suggestions for this research work. Their insights on the topic of my project helped me a lot; especially their guidance and feedback drove me in the proper direction of this research all the time.

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 would like to thank my husband and my daughter for their moral support and encouragement.

I would also 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 accomplish this research.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Runtime verification is a testing/debugging technique that combines monitoring and formal verification during program execution. Runtime verification checks a program run against safety properties, unlike traditional testing techniques such as unit testing which are ad hoc and informal. The focus of this thesis is runtime verification of distributed programs. In order to verify that a run of a distributed program is correct or not, it is required to specify both the correct concurrent behavior of the distributed program and the correctness of the computation performed by it. There are different techniques reported in the literature to specify concurrent behavior of a distributed program like Petri nets, interleaving models and partial order models.

As we trace back into the distributed computing literature, Petri net is a very powerful model that people developed many years ago, and it is a means to represent concurrent systems with explicit concurrency and nondeterminism [Mu89]. It is very powerful due to the generality and permissiveness inherent in Petri nets. However, it is a tradeoff between modeling generality and analysis complexity. Although the power is there, the significant cost in analysis prevents Petri nets from wide-spread use [Mu89].

In adopting the interleaving semantics to model concurrency and use a state machine representation of a concurrent system, state explosion became an issue [God96]. Fortunately, recent results in formal verification have introduced same powerful partial-order reduction techniques [Pel96] to overcome such state explosion in large spectra of scenarios. Indeed it is possible to check a large class of temporal logic properties (LTL-X) in polynomial time [Spin]. However, the difficulty of formalizing a requirement in temporal logic is still a problem [CDHR01].

On the other hand, to specify the necessary ordering requirement in partial order semantic is easier [LM07]. For example, in agent protocols, the ordering in the interaction diagram is explicit. We can easily specify the ordering between events using recurrent partial orders in partial order semantics.

Then, we ask ourselves how nice it would be to develop a model using partial order semantic so that we can take advantage of the attractive benefit of usability from partial order without suffering the humongous cost in analysis as in Petri nets?

1.2 Research Goal

Generally speaking, runtime verification of a distributed program involves two parts of work: modeling the distributed computation formally; and specifying the properties to be verified at runtime. In order to do that, in the past twenty years, interleaving model inherent in temporal logic has successfully augmented model checking with concurrency in the form of partial reduction techniques [Pel96]. It uses the temporal logic to represent both the ordering requirements and the property requirements successfully. Several runtime verification tools have been developed correspondingly, like Eagle, Hawk, MOP, Java Pathfinder, etc. All of them model the distributed computation as a state lattice, and specify the properties by temporal logic.

However, the logic formulas are not natural to use and it is very expensive to specify the ordering among events in the code space [CDHR01]. To overcome this problem, we explore the partial order model which separates concerns in checking the necessary ordering among abstract events and the property requirements of each abstract event. Therefore, predicates can be used at each abstract action. This thesis tries to use the partial order model with the atomization technique to achieve this goal. It sets out to develop an interface specification of the partial order requirements and the computational requirements to be checked in the execution of a distributed program, together with an implementation that checks these requirements against a run on-the-fly.

We choose to check the execution on-the-fly, rather than off-line; because through on-the-fly verification, the program behavior is analyzed during its execution so that errors are detected and reported as they occur during the run. Hence there can also be an on-the-fly error recovery. Moreover, since the execution will stop when the error is detected, no more resources are needed. On the other hand, offline verification collects a log of events that occur during a program's execution and post-process the log to detect errors. The main disadvantage is that execution logs can be very large for parallel and distributed programs.

1.3 Contributions

The objective of this thesis is to develop a tool to check the partial order requirements and the computational requirements against the execution of a distributed program. In order to do that, we have modeled a distributed computation using partial order semantics by a Pomset automaton model. The Pomset automaton model allows us to specify all the allowable partial orders in the system. The complete specification involves two parts: the ordering requirements and the computation correctness requirements. In the literature, there are different techniques people have used to identify and capture relevant information in order to verify the execution [CR06] [AH05] [JPF]. We will review some existing runtime verification tools for introducing these techniques in Section 2.4.

After we successfully model the distributed computation and specify the requirements, in order to check the execution against these specifications, we will define atoms as abstract events and predicates of each atom as property requirements. An atom is a set of statements of the program that are expected to be performed atomically. The concept of atoms has been proposed as an effective state space reduction mechanism [LMG07]. Moreover, to capture information related to the atoms and the predicates, we will define relevant variables, including atom variables, which are relevant in detecting atoms for the purpose of ordering checking; and predicate variables which are relevant in representing the global states for the purpose of predicate checking. Finally, the tool performs ordering checking and predicate checking separately.

In order to keep the complexity of checking as little as possible, we use maxi atoms in our model, which could contain more than one relevant statement. A relevant statement is a statement in the source code whose execution may change the value of relevant variables. By using maxi atoms the number of states that need to be considered in property checking can be significantly reduced.

At runtime the monitor extracts the atoms through the execution of the instrumented code. The source code can be hand-instrumented by ourselves or automatically instrumented by other existing tools. We simply use AspectJ, a well-developed Aspect Oriented Programming (AOP) tool, which seamlessly extends aspect-oriented programming to the Java language [AspectJ], to do the instrumentation. AOP is a software development technique that aims to increase modularity of orthogonal programming concerns [Kic97]. An aspect is a module that characterizes the behavior of "cross-cutting concerns". It defines behavior that crosscuts different abstractions of a program, avoiding scattering code that is related to a single concept at multiple places of the program, and as a consequence, protecting the encapsulation of modules. The atom variables and the predicate variables are used as the inputs to AspectJ for the purpose of catching the relevant events from the atoms to be monitored.

It should be noticed that we choose multi-Agent systems just for the purpose of representing a specific platform of distributed systems to implement our tool. We do not fall into the field of the multi-agent systems testing and debugging. Our model can be used to facilitate the debugging of multi-agent systems to some extent; however, there are characteristics specific to agent systems that could be addressed but ignored here.

Thus, this research involves the design and implementation of a runtime verification tool for distributed programs written in the agent-based Jade platform. The ordering requirement of the concurrent program is modeled by a Pomset automaton specifying the correct partial order behaviors of the computation. The correctness of the computation performed by a set of source code statements that are expected to be performed atomically (an atom) is specified by a predicate on variables modified by the atom. The user is required to provide the Pomset automaton and the predicates to the tool using appropriate input files. The user also identifies appropriate atoms and atom variables in the computation so that the tool can observe the atoms using the specified atom variables at run time and check the validity of the predicates at the end of the atoms to verify the correctness of a run on-the-fly. The observation of atoms and variables is done through AspectJ.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 reviews the related research and relevant tools in the literature. Chapter 3 introduces the Pomset Automaton model to capture the correctness requirements. This model involves two parts: the ordering requirements among atoms and the correctness of each atom. Section 3.1 presents our Pomset automaton model that allows different behaviors of the system to be checked, and explains how to define the atoms in the requirement space and map them into the code space using atom variables. Section 3.2 defines the correctness of an atom using predicates on a state that is reached upon the completion of an atom.

Chapter 4 contains the details of the system and algorithm design of our tool for the purpose of monitoring and checking. It describes the critical issues in the design; for example, how to map from design space to code space, how we use AspectJ to take care of the monitoring, how to unfold the input automaton properly, how we allow for indexing in the model to overcome the capability problem, and how to allow the merging of atoms in the run, etc.

Chapter 5 gives a sample use of the tool through a multi-agent E-market application program. By giving a well selected snapshot of some part of the code, we explain how to relate the code to the atoms, and how to write the user input file. We also give two sample outputs of the tool at the end. Finally, Chapter 6 concludes the thesis.

Chapter 2 Background and Related Work

2.1 Lamport's Atomicity Theorem

Lamport has developed an atomicity theorem to simplify verification of distributed systems [Lam90]. Lamport adopted the common approach of formally defining an execution of a distributed program to be a sequence of atomic actions. At the lowest level of abstraction in the code level each event that results from executing a statement in the distributed program is considered an atomic action. Reducing the number of atomic actions makes reasoning about a concurrent program easier because there are fewer interleaving to consider. This is the main goal of Lamport's atomicity theorem. According to this theorem, a sequence of statements in a distributed program can be grouped together and be treated as a single 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.

2.2 Pomset model

An important issue in runtime verification of distributed programs is the model used to specify the distributed computation. In [LM07], H. F. Li and E. Al-Maghayreh proposed a runtime verification methodology that exploits the concept of atoms and the partial order semantics of a distributed computation. According to this methodology, a distributed computation is modeled as a partially order multi-set (Pomset). Pomsets are attracting more attention in modeling and analyzing distributed programs due to the fact that they model true concurrency [LRG04, LL94, PL93, PL91b, PL91a, Pra86]. The Pomset model promotes the separation of two different concerns in specifying and checking properties: (i) the necessary ordering among the atoms and (ii) the correct execution of each atom (in its effects on the global state). In this section, we review this Pomset model.

2.2.1 Model of Distributed Computation

A distributed computation is modeled by an atomized run. An *atomized run* is a Pomset of well-formed atoms. Each atom can be labeled with the set of relevant variables associated with its relevant events. A *well-formed atom* is an atom in which there is no receive event after a relevant event, and no relevant or receive event after a send event, and the atom is maximal in length. Thus, if a well-formed atom is extended by including either a preceding event or a succeeding event 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, the run in Figure 1 is partitioned into five atoms $\{E_1, \ldots, E_5\}$. Rectangles are used to depict atoms, and circles and black squares are used to depict events. The events in process P_1 are split into two atoms: E_1 and E_2, the events in process P_2 are split into two atoms: E_3 and E_4, whereas all the events in process P_3 form a single atom E_5. Atom E_2 is not well-formed since 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, atom 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.

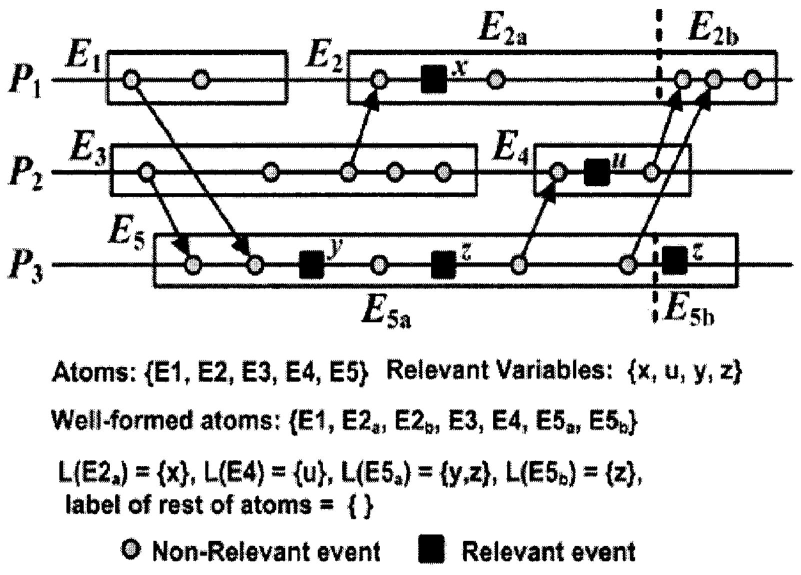

E_{2a}
E_{2b}
P_1
E_1
E_2
x
P_2
E_3
E_4
u
P_3
E_5
y
z
z
E_{5a}
E_{5b}
Atoms: {E1, E2, E3, E4, E5} Relevant Variables: {x, u, y, z}
Well-formed atoms: {E1, $E2_a$, $E2_b$, E3, E4, $E5_a$, $E5_b$}
$L(E2_a) = \{x\}$, $L(E4) = \{u\}$, $L(E5_a) = \{y,z\}$, $L(E5_b) = \{z\}$,
label of rest of atoms = { }
○ Non-Relevant event
■ Relevant event

Figure 1: An example run and its resulting atoms/well-formed atoms. [LM07]

The *atom slice* of a run is the projection of the atomized run onto the set of relevant atoms.

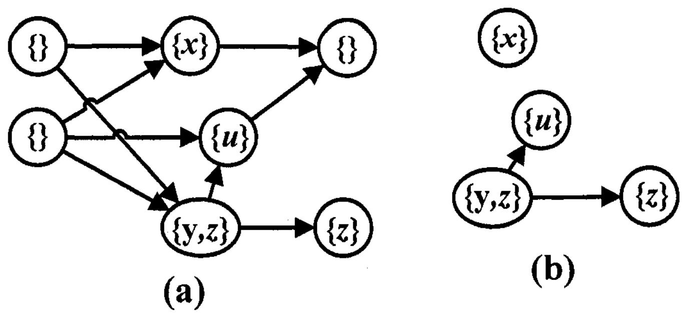


Figure 2: (a) Atomized run (b) Atom slice [LM07]

For example, Figure 2(a) shows the atomized run (Pomset) involving the relevant variables $\{u, x, y, z\}$ of the run in Figure 1. Each node represents a well-formed atom and is labeled with its relevant variables. Figure 2(b) shows the corresponding atom slice, which corresponds to the projection of the atomized run onto $\{u, x, y, z\}$ and contains only those atoms with non-empty labels. This slice contains atoms that are relevant to the required order checking and computation checking of atoms.

2.2.2 Specification and Checking of Ordering Requirements

A representation of a Pomset based on a set of recurrent sequences $\hat{S} = \{S_1, S_2, \ldots, S_n\}$ was proposed in [LM07]. Each recurrent sequence S_i is of the form

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

(b) $[A_1 + A_2 + \ldots + A_k]^*$.

Form (a) uses the sequence operator ";". For any two atoms A and B, $(A ; B)$ means that atom A should be causally ordered before atom B in any run. Atom A is causally ordered before atom B (denoted by, $A \rightarrow^{c} B$) if there exists an event in A that happened before an event in B. The "*" denotes the recurrence 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 $A_1 ; A_2 ; \ldots ; A_k ; A_1 ; A_2 ; \ldots ; A_k ; \ldots$ etc. Hence form (a) recurrence represents a fixed ordering among a recurrent set of atoms. 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 $A_1 \rightarrow^{c} A_2 \rightarrow^{c} \ldots \rightarrow^{c} A_k \rightarrow^{c} A_1 \rightarrow^{c} A_2 \rightarrow^{c} \ldots$ 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, $A_1 \rightarrow^{c} A_3 \rightarrow^{c} A_k \rightarrow^{c} A_2 \rightarrow^{c} A_1 \ldots$ 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.

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 the atom slices (Pomsets) shown in Figure 3(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 3(b) contains more ordering (such as $C \rightarrow^{c} A$) than is required. However, the atom slice in Figure 3(c) fails $\hat{S}$; $C \rightarrow^{c} Y$ is required by S_2 but is not satisfied by the slice.

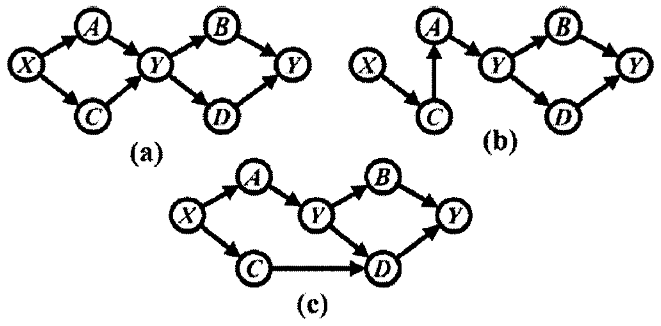


Figure 3: (a) An atom slice that satisfies $\hat{S}$, (b) An atom slice that is stricter than $\hat{S}$, (c) An atom slice that fails $\hat{S}$. [LM07]

2.2.3 Specification and Checking of Computational Requirements

Assuming that the ordering requirements among atoms are satisfied, the next step is to check the computational requirements. The computational correctness of an atom is modeled by an invariant that should be satisfied at each minimal state of each instance of the atom in the atom slice. The minimal state is the state reached upon executing the atom itself and all the atoms that are causally ordered before it.

2.3 Multi-agent System in Practice

The Java Agent DEvelopment Framework (JADE) is a software Framework fully implemented in Java language [JADE]. We choose JADE programs as the programs under test for our verification tool and choose JADE platform to implement the tool for several reasons. First, JADE simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications [FIPA]. So it is widely used by the multi-agent community. Furthermore, JADE supports many possible application areas, such as

1) Mobile applications, which could use personal agents to support users on the move (personal agents facilitate search and discovery of information through interaction with peers (people or service providers);
2) Internet applications that enable end users to deal with available resources' complexity and to allow seamless access to remote resources and services;
3) Corporate applications to simplify collaboration and cooperation between systems and people to achieve better results; and
4) Machine-to-machine applications, such as automatic control or traffic management systems.

Second, JADE implements all basic FIPA specifications which makes it a normative framework. FIPA promotes agent-based technology and the interoperability of its standards with other technologies.

Moreover, FIPA provides AUML as a standard agent-oriented modeling technique and methodology based on the agent software development process [AUML]. From AUML diagrams, the user can easily and intuitively extract the ordering requirement specification to use our tool. AUML Sequence Diagrams were initially adopted by FIPA to express agent interaction protocols. Sequence diagrams are defined as a diagram that shows interactions among agent roles arranged in time sequence. In particular, it shows synchronization among roles via message exchanges. Two parts can be considered in sequence diagrams: a frame, which delimits the sequence diagram and the message flow between roles through a set of lifelines and messages. Figure 4 shows the sequence diagram of Iterated Contract Net Interaction Protocol in FIPA Interaction Protocols specifications.

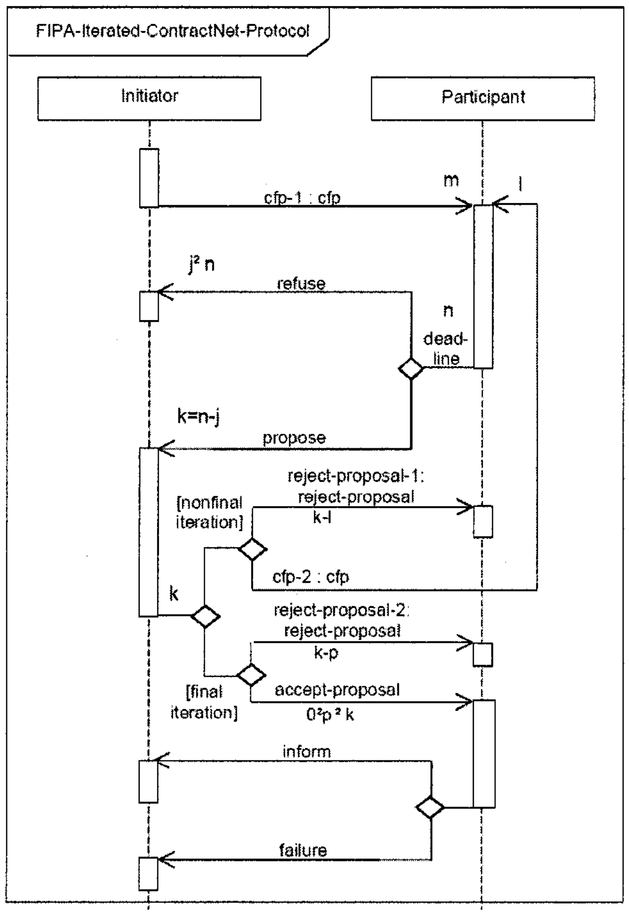


Figure 4: Sequence diagram of Iterated Contract Net Interaction Protocol

In the FIPA Contract Net Interaction Protocol, the Initiator issues m initial calls for proposals with the *cfp* act [FIPA]. Of the n Participants that respond, k are propose messages from Participants that are willing and able to do the task and the remaining j are from Participants that refuse.

Among the k proposals, the Initiator may decide this is the final iteration and accept p of the bids ($0 \leq p \leq k$), and reject the others. Alternatively the Initiator may decide to re-iterate the process by issuing a revised *cfp* to l of the Participants and rejecting the remaining k-l Participants. The intent is that the Initiator seeks to get better bids from the Participants by modifying the call and requesting new (equivalently, revised) bids. The process terminates when the Initiator refuses all proposals and does not issue a new *cfp*, or accepts one or more of the bids or the Participants all refuse to bid.

Any interaction using this interaction protocol is identified by a globally unique, non-null conversation-id parameter, assigned by the Initiator. The agents involved in the interaction must tag all of their ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities; for example, it allows an agent to identify individual conversations and to reason across historical records of conversations. Additionally, the messages may specify other interaction related information such as a timeout in the reply-by parameter that denotes the latest time by which the sending agent would like to have received the reply message in the protocol flow.

We choose to translate the AUML requirement to Pomset rather than use AUML itself to specify the ordering requirement for the reason that AUML is visual notation which is not suitable for automatic checking.

Class diagrams, collaboration diagrams, sequence diagrams, and the interaction overview diagrams all are the design artifacts specified by AUML notation that contains the ordering requirement for run-time error detection. We choose sequence diagrams to achieve our goal in this demonstration because it is the most intuitive and natural one from which we can capture the ordering easily. After we capture the ordering requirements among the abstract actions in the design space, these abstract actions can be mapped into the code space using relevant variables.

A sample use of our tool to illustrate how interaction protocols specified using AUML notation is translated to the ordering requirement of Pomset using relevant variables will be presented in Chapter 5.

2.4 Relevant Tools

In this section, we will review some existing runtime verification tools from the usage perspective. Generally, in order to monitor safety properties against a trace of events emitted by the running program, users need to specify safety properties in some temporal logic and instrument the program for event identification.

2.4.1 Eagle

Eagle is a rule-based runtime verification framework [AH05]. Rules, which are the properties that need to be satisfied by the program, can be parameterized with formulas and data-values; thus supporting specifications that can reason about data which can span over an execution trace. Given a finite sequence of program states and a set of rules written in temporal logic, Eagle checks if the trace satisfies the set of rules. Monitoring is done on a state-by-state basis.

The user is responsible for specifying the rules and monitors, and identifying relevant events of the program to be verified. The monitors define the points at which the rules should be checked. The relevant events are identified by an instance of a record having a pre-specified schema, using a set of relevant variables. For example, LoginLogoutEvent {*userId:* string; *action:* int; *time:* double} is the schema of an event and {*userId* = "Bob"; *action* = login; *time* = 18:7} is an event representing the fact that user "Bob" has logged in at time 18:7. This set of relevant variables represents the current state of the program, denoted by a user-defined Java object. In order to capture the relevant events, the source code needs to be hand-instrumented in a few places. When an instrumentation point is hit during program execution, the EAGLE state is updated (shown as Step 1 in Figure 5). The instrumentation point actually maps to a block of code, which could contain several relevant statements, and captures a relevant event. Then, the observer corresponding to the specified properties (Spec) is notified (Step 2). In response, the observer evaluates the formulae in the current state (Step 3) and derives new obligations for the future which are stored in its internal state.

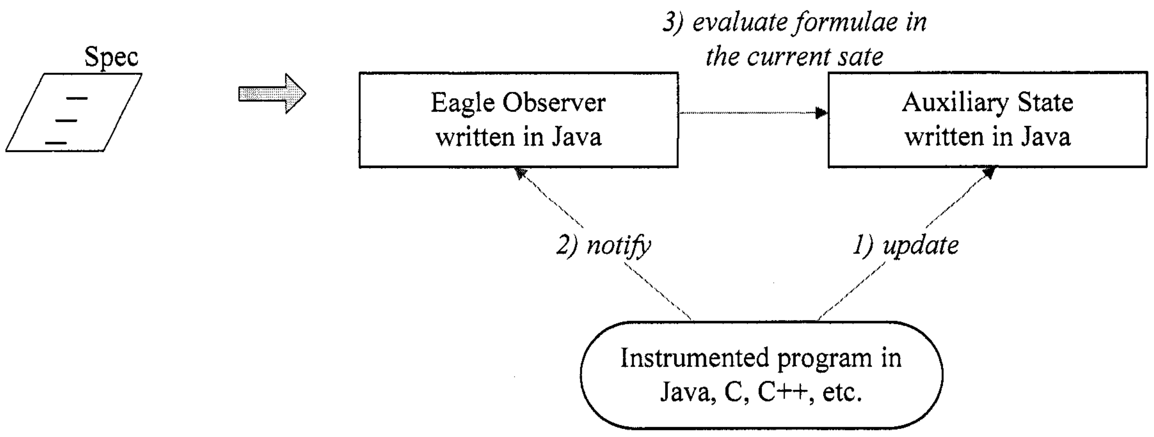

Spec
Eagle Observer written in Java
3) evaluate formulae in the current sate
Auxiliary State written in Java
2) notify
1) update
Instrumented program in Java, C, C++, etc.

Figure 5: Eagle Architecture [AH05]

As an example, consider the temporal property: *"whenever at some point x = k > 0 for some k, then eventually y = k".* This can be expressed as follows in quantified LTL:

$$\Box(x > 0 \rightarrow \ni k :(x = k \wedge \rightarrow y = k)).$$

Eagle uses a parameterized rule to state this property, capturing the value of x when $x > 0$ as a rule parameter.

$$\underline{\text{min}}\ R(\underline{\text{int}}\ k) = \text{Eventually}(y = k)$$

$$\underline{\text{mon}}\ M = \text{Always}(x > 0 \rightarrow R\ (x))$$

Rule R is parameterized with an integer k, and is instantiated in M. The detailed Eagle interface is shown in Figure 6. Besides a set of temporal rules and a set of monitoring formulas (M1, M2), the user needs to define the relevant variables (x, y) as class data members to present a state of the program. The source code has to be first instrumented so that the assignments of *x* and *y* will form an event. During the execution, those events are captured, recording the values of *x* and *y*. Only when $x > 0$, M1 and M2 apply on the state.

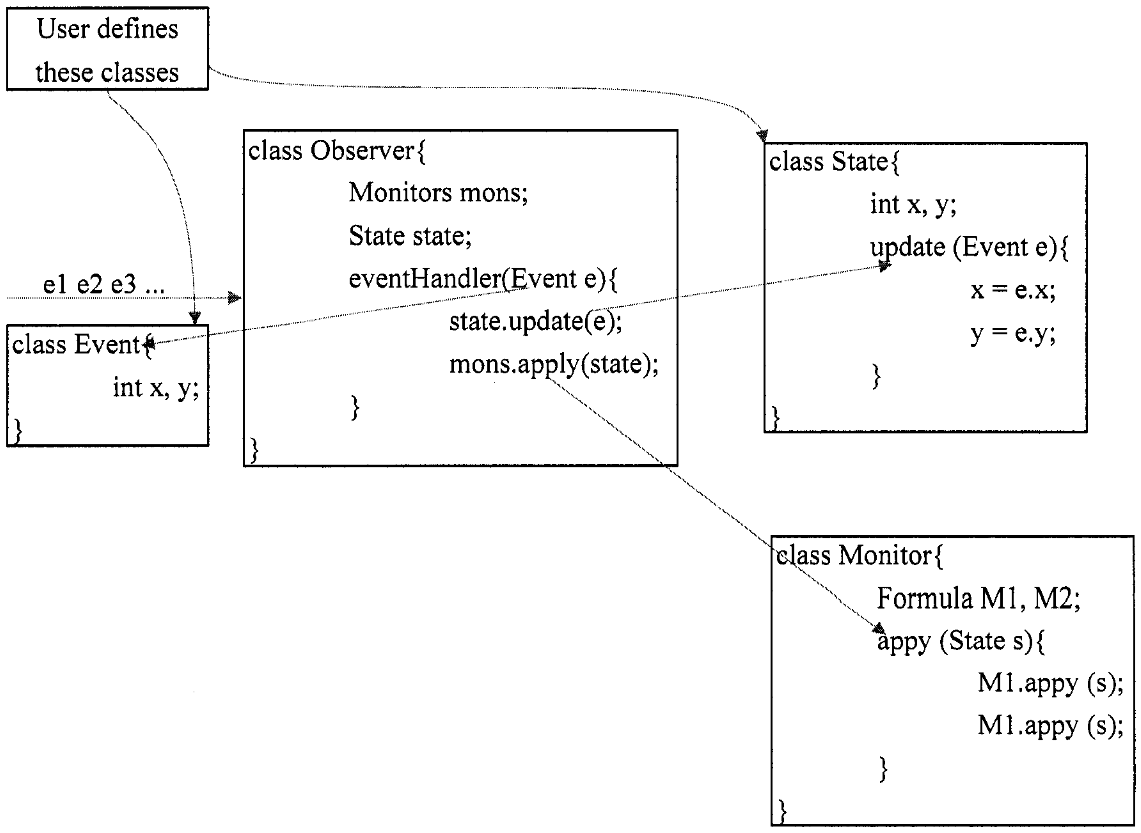


Figure 6: Eagle interface [BGHS04]

2.4.2 Hawk

HAWK is built on top of EAGLE [AH05]. HAWK specifications are ultimately translated to EAGLE monitors. HAWK follows an event-based approach to runtime verification in contrast to EAGLE which is state-based. It extends EAGLE with event expressions that allow one to bind data values from parameterized program events. Event expressions actually are expressions of methods; these methods can have parameters and return values, etc.

Hawk integrates EAGLE with AspectJ to realize automatic instrumentation. The integration of Eagle with AspectJ is by supporting temporal cutpoints. Temporal EAGLE formulae now become part of the AspectJ cutpoint language, and can function as triggers for actions to be executed. However, the Hawk specification becomes more complex, from which Eagle specification and instrumentation aspects are generated.

To use Hawk, the user needs to identify relevant methods (events), and specify monitors (properties). Figure 7 illustrates the format of a logic observer specification in HAWK.

```
observer BufferObserver {

classPath = C:/downloads/src
targetPath = C:/downloads/src
terminationMethod = befferexample.Barrier.end()

var Buffer b;
var object o;
var object k;

mon B =
          Always ( [b?.put(o?)]
                    Eventually ( <b.get() returns k?>(o = = k) ) )

}
```

Figure 7: An example of Hawk specification [AH05]

In Figure 7, monitor B states the property that whenever an object o is inserted into a buffer b, eventually it is taken out from that buffer. The `put` and `get` methods are events.

The actual parameter of the `put` method and the return value of the `get` method can be captured. In Hawk, the states of the program are checked through the captured parameters of method invocation unlike the relevant variables in Eagle. Thus, Hawk is event-based. Hawk compiles the Hawk specification to generate the Eagle specification and Eagle state, and AspectJ aspects. Then the Eagle monitoring engine runs on a trace.

2.4.3 MOP

Monitoring-Oriented Programming (MOP) is a formal framework for software development and analysis [CR06]. Like Hawk, AspectJ is integrated into JavaMOP to realize automatic instrumentation, aiming at reducing the gap between formal specification and implementation via runtime monitoring. JavaMOP is a MOP development tool for Java. In MOP, the developer specifies properties using definable specification formalisms. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the recovery code into the original system.

The user is responsible for identifying relevant fields and methods, specifying temporal logic, and specifying validation handlers and violation handlers in MOP specification. In Figure 8, *the specification processor* extracts MOP specifications from the program and dispatches them to appropriate modules on the lower layer to process, and then collects the monitoring code generated from the lower layer and integrates them into the original program. Aspect-Oriented Programming (AOP) plays a critical role here: the *specification processor* synthesizes AOP code and invokes AOP compilers to merge the monitors within the program. The instrumented program applies the logic along with the validation handlers and violation handlers on a trace.

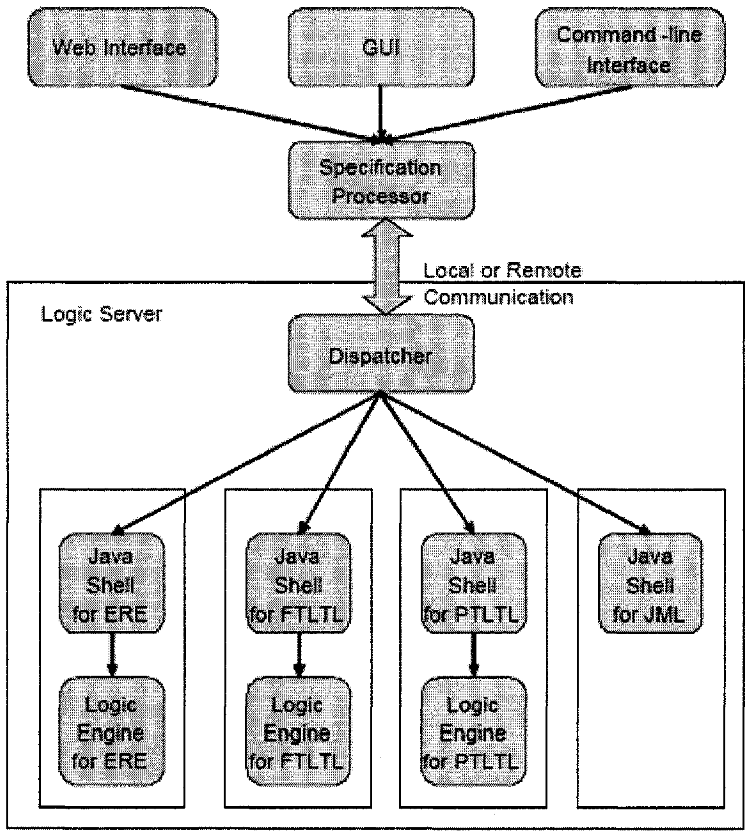


Figure 8: The Architecture of JavaMOP [CR06]

The following example is a simplified cruise control system whose behavior only concerns the actions of setting and canceling the cruise mode. The specification can be informally described as follows: "Once the cruise control has been set, the speed of the car should not be 5 miles more than or less than the selected cruise speed until the cruise control is released."

Suppose that the car control system is implemented in the `CarController` class in Figure 9, which contains the operations for starting/stopping cruise control (`setCruiseControl()` and `releaseCuriseControl()`), as well as the fields for recording speeds. Then Figure 10 gives the FTLTL-based specification to formally specify the desired behavior of the system.

```
class  CarController {
        int currentSpeed;
        ...
        int targetSpeed = 0;
        void setCruiseControl(){
                ... targetSpeed = currentSpeed; ...
        }
        void releaseCruiseControl(){
                ... targetSpeed = 0; ...
        }
        void doBrake();
}
```

Figure 9: Car controller class [CR06]

```
/*@
scope = class
Logic = FTLTL
{
Event setCC : end(exec(* setCruiseControl()));
Event releaseDD : end(exec(* releaseCruiseControl));
Predicate upperBounded : currentSpeed < (targetSpeed + 5);
Predicate lowerBounded : currentSpeed > (targetSpeed + 5);
Formula : □ (setCC → ((upperBounded ∧ lowerBounded) U releaseCC));
}
Violation Handler :
            @this.releaseCruiseControl();
            @Reset;
@*/
```

Figure 10: MOP specification for cruise control [CR06]

We can see that MOP specifies the mapping from the design-level events that represent the actions of starting and stopping the cruise mode to the method execution in the code space, and the mapping from two predicates defined to check the proper range of the car speed to the relevant variables in the code space.

In the instrumentation, the monitoring code will be inserted after the two cruise mode related methods to get every update of currentSpeed and targetSpeed, and check the predicates.

Chapter 3 Pomset automaton model

The Pomset automaton model promotes the separation of two different concerns in checking a distributed computation: the ordering between the executed atoms and the correctness of the computation performed by each atom. Therefore, the model falls into two separate subsections. In the first subsection (Section 3.1), we will introduce Pomset automaton which generates the set of allowable behaviors of a distributed program. In the second subsection (Section 3.2), we will present the correctness of an atom by using predicates on a state that is reached upon the completion of an atom.

3.1 Pomset automaton model

3.1.1 Partial order model

A partial order model explicitly models the ordering among events or atoms. In our work, an event is the execution of a program statement; an atom is a sequence of statements executed, which can be treated as an atomic action under some stated conditions. Using partial order model, ordering requirements involve a set of allowable partial orders, each representing a run of the distributed program. The set of partial orders is generated by a partial order automaton.

We will illustrate the partial order model through the following example of E-Market application. Table 1 summarizes the different roles in this application and describes their responsibilities and objectives, according to the system specification. The role diagram is shown in Figure 11.

Table 1: Roles in E-Market application and their responsibilities

ROLE	**DESCRIPTION**
Client	The person who conducts a purchase with the help of a broker and an accounts manager.
Broker	People who broker a purchase for a client with a wholesaler.
Wholesaler	A wholesaler sells an item through a broker and informs the accounts manager to complete the transaction (receipt of money and delivery of product).
Account Manager	An accounts manager completes a transaction with the client, as instructed by a wholesaler.

Interaction Protocols are used for maintaining relationships between roles. In this application the FIPA Iterated ContractNet Protocol is used. For example, Figure 12 shows the interaction protocol diagram between the client and broker roles. These diagrams can be combined to create a single diagram that depicts all the interaction protocols used between different roles in the application.

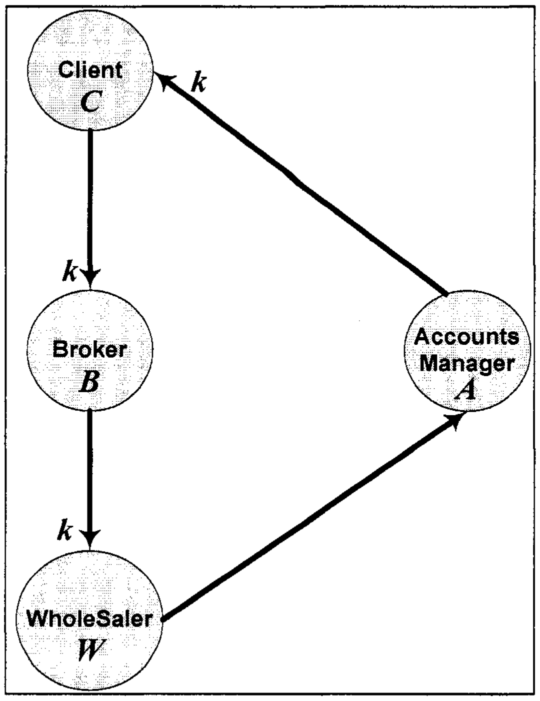


Figure 11: Role diagram of the E-market application.

Figure 13 shows that complete interaction protocols diagram for the E-market application described below.

1. Initially a client *C* (who wants to purchase a product *P*) will send a call for proposal to all of the known brokers ($B_1, \ldots, B_m$). We call this protocol session as session 1.
2. As protocol session 1 is evolving, when a broker receives a call for proposal from Client *C*, he will send a call for proposal to all of the wholesalers he knows ($W_1, \ldots, W_n$). This will embed another protocol session identified as session 2_broker(i).
3. During session 2_broker(i), when a wholesaler receives a call for proposal from a broker *B*, he will check his catalogue to see if product *P* is available and now he has the following two choices: if the product is available, the wholesaler will send a

propose message to the broker (Step 3a), otherwise the wholesaler will send a refuse message to the broker (Step 3b).

4. If the broker receives a non-empty set of proposals he will choose the best one (session 2_broker(i) pause, but the session 1 keeps evolving.) and send it in a propose message to the client *C* (Step 4a), otherwise he will send a refuse message to the client (Step 4b).
5. If the client receives a non-empty set of proposals from the brokers known to him, he will choose the best one and send a purchase order to the corresponding broker (Step 5a: session 1 keeps evolving.); otherwise the client will send another call for proposals (Step 5b: session 1 ends and a new session starts).
6. When the broker receives a purchase order, he will forward it to the corresponding wholesaler (session 2_broker(i) proceeds).
7. When the wholesaler receives a purchase order, he will check his catalogue and send an inform message to the accounts manager.
8. When the accounts manager receives an inform message from a wholesaler, he will send a bill to the corresponding client if the purchase has been approved by the wholesaler (Step 8a), otherwise he will send the client a failure message (Step 8b).
9. When a client receives a bill from an accounts manager he will send the specified amount to the accounts manager (Step 9a); if the client receives a failure message from the accounts manager, he will send another call for proposal (Step 9b: session 1 ends and a new session starts).

10. When the accounts manager receives the payments he will deliver the product to the client (all sessions end).

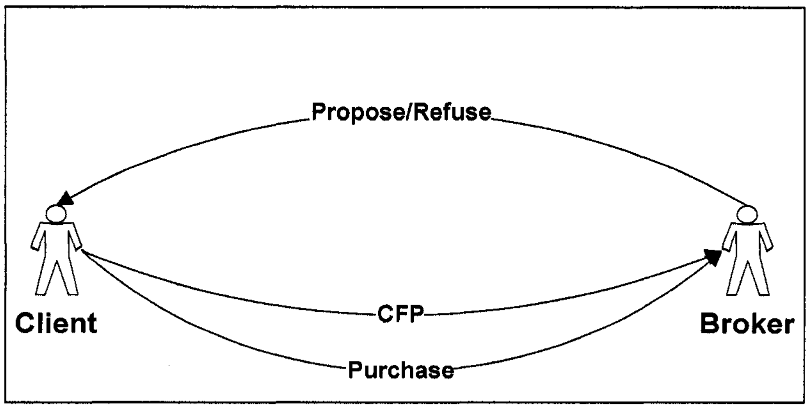


Figure 12: Interaction Protocol Diagram (Client-Broker)

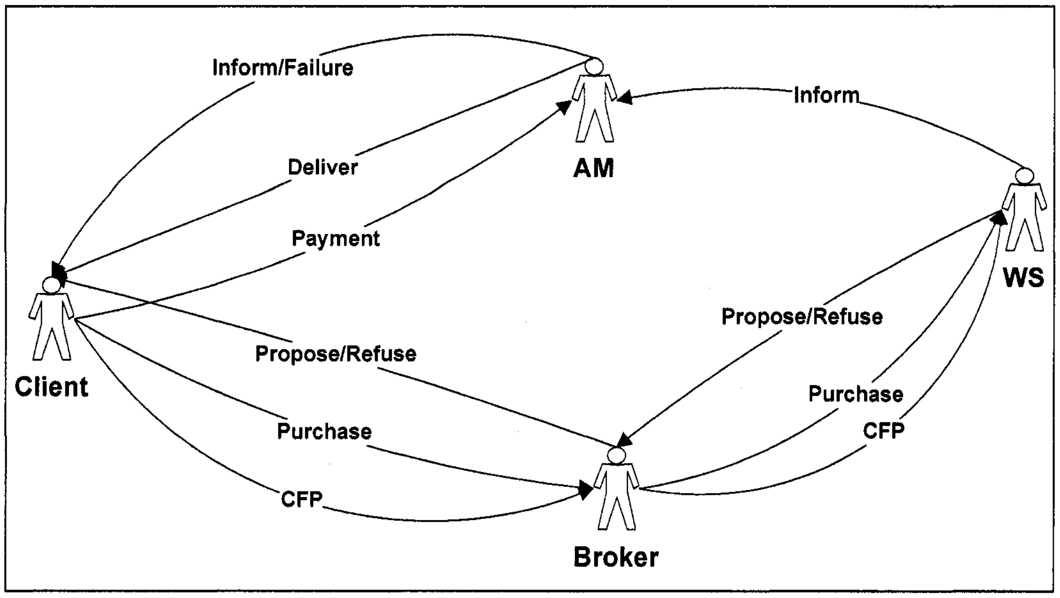


Figure 13: The complete interaction protocols diagram for the E-market application.

According to the protocol, a run of the E-Market application could be as shown in Figure 14, where a node is an abstract event in the protocol space and an edge represents the ordering requirement between two events. In the figure, some events are un-ordered. For example, 2 and 2', and 3, 3' and 3" are allowed to be concurrent.

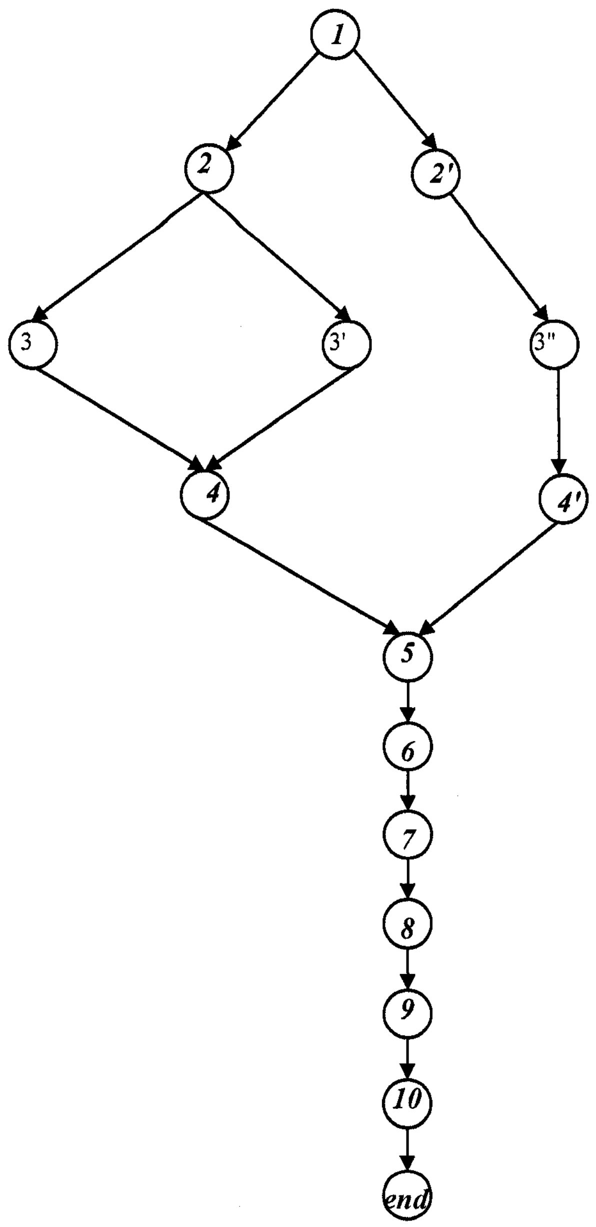


Figure 14: A run of the E-market application

In this case, the client has two brokers. Broker1 has three wholesalers and broker2 has one. Wholesaler1 and wholesaler2 sent a propose message to broker1 in Step3; while wholesaler3 sent a refuse message. And wholesaler4 sent a propose message to broker2. In Step 4 broker1 sent a propose message to the client, while broker2 sent a refuse message to the client. Then client sent a purchase order to one of the brokers, the broker forwarded it to the corresponding wholesaler, the wholesaler sent an inform message to the accounts manager, the accounts manager sent a bill to the client, the client sent the specified amount to the accounts manager, and the accounts manager received the payments he delivered the product to the client at the end.

Another run of the application could be a little different from the above one as shown in Figure 15: in Step 4, broker2 also sent a propose message to the client; and in Step 8, however, the purchase had not been approved by the wholesaler, the account manager sent the client a failure message, and the run ended. This different run leads to the required ordering shown in Figure 15.

The two runs form a subset of all allowable runs from the nested protocol sessions. When viewed together, such a set can be represented by a behavior tree. Each path in the behavior tree contains an allowable partially ordered run, such as the one in Figure 14 or Figure 15.

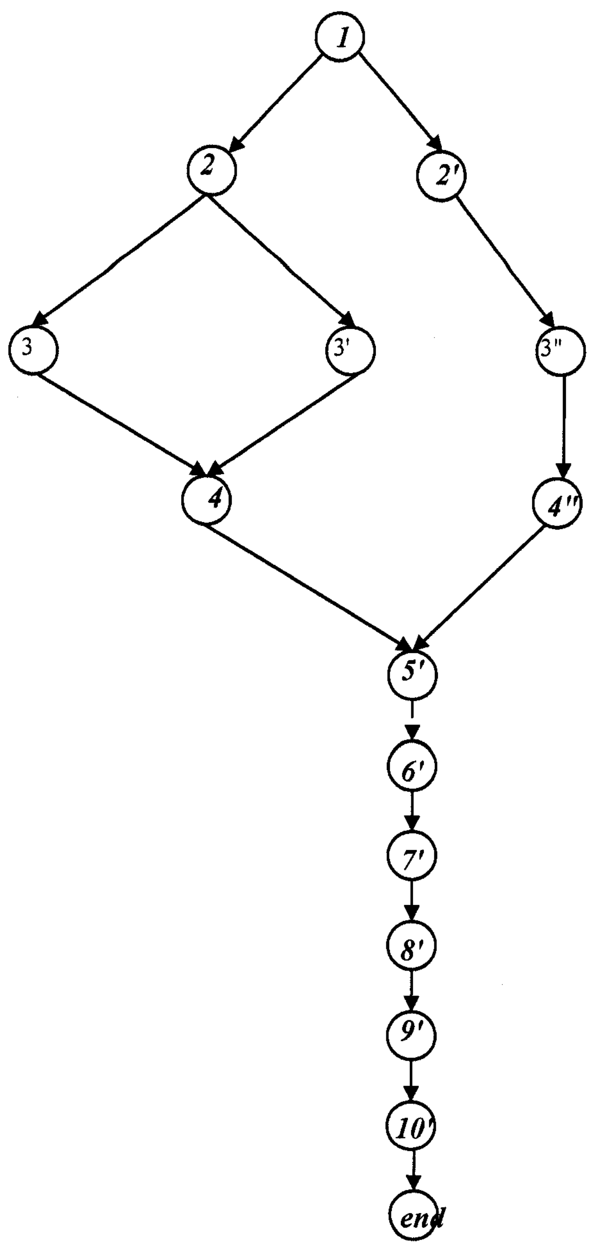


Figure 15: Another run of the E-market application

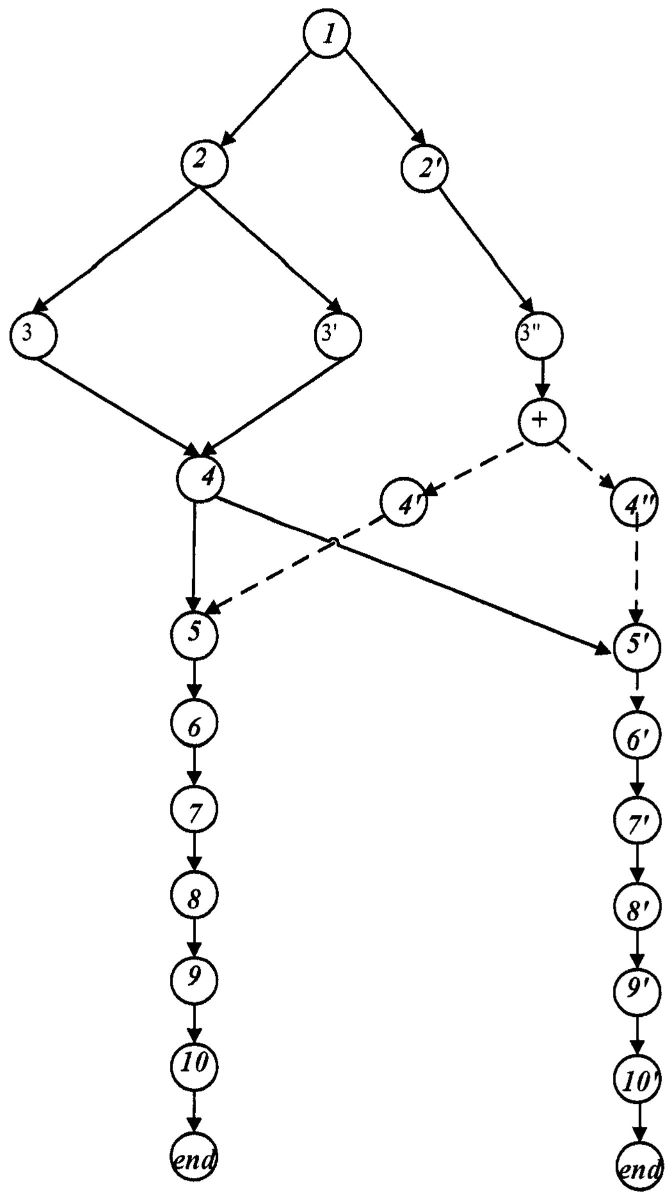


Figure 16: Combination of two runs

3.1.2 Semantics of Pomset automaton

Observably, the set of partial orders in the ordering requirement can be viewed as a behavior tree of partial orders. The initial states of the program are the root of the tree. The tree will grow as the program is executing. Some nodes lead to a choice 'condition' from where different 'branches' of the behavior may be followed; this is where branching in the partially ordered behavior tree occurs. We denote such instances of branches with a '+' as exemplified in Figure 16. When a run gets to a choice condition, it will choose one of the choices/branches and follow that path, until the next condition.

With the view that the ordering of atoms is represented by a behavior tree of partial orders, we need to derive a finite representation of the behavior tree. In general, the behavior tree can be infinite for a non-terminating system. This is accomplished by identifying ‘slot’ or conditions in a path of the behavior tree which recur: the same subtree of behaviors will emanate from such slots whenever a run reaches these slots. For example, a call-for-proposal (slot) in the former example may repeat indefinitely, generating an arbitrarily expanding subtree of partial orders.

The finite representation we have developed makes use of such recurrent conditions, which can be choice (such as the + in Figure 16) or non-choice conditions in a behavior tree to form a partial order automaton, which will be referred to as a Pomset automaton. The Pomset automaton model has a starting condition denoting the initialization of all involved processes. As these processes progress forward, atoms relevant to ordering requirements are identified. Their required ordering is specified in the form of a set of transition rules, each rule containing a finite Pomset emanating from a set of slots and ending at a set of slots. Since a set of slots can represent a recurring condition which can have multiple choice rules to be applied, the automaton will lead to a behavior tree.

An execution (run) satisfies the ordering required by a specified Pomset automaton only if every order in the specification is satisfied in the execution. Hence it is conceivable that monitoring and checking an execution against the specification is fairly straightforward and can be done simply in polynomial time, provided that the identification of atoms is instrumented properly in the code space.

We will provide an informal description of the Pomset automaton below. The basic entities in a Pomset automaton include: (i) a set of slots, and (ii) a set of transition rules. A transition rule includes: (a) one or more start slots, (b) one or more end slots, and (c) a finite Pomset concatenating the start slots to the end slots. To illustrate this, let us consider a simplified scenario of the previous E-market example by slicing the role diagram and focusing on the sub-space involving a buyer agent and two seller agents. As a result, the buyer agent conducts the contract net protocol with the two seller agents, which is captured by the Pomset automaton shown in Figure 17.

In Figure 17, transition T_1 involves (a) start slot S, (b) end slot S_1, and (c) the finite Pomset connecting S to S_1, representing the call-for-proposal (A), the replies (D_1 and D_2), and the processing of the replies (B). In the figure, slots are drawn in circles and atoms are drawn in squares.

Multiple rules can emanate from the same set of start slots, in which case, the slots form a choice condition. When a choice condition is reached, the behavior will evolve forward with choice, by following one of the rules in the automaton. In Figure 17, S_1 is a choice condition (set of start slots) from which three rules, T_2, T_3 and T_4, may follow, depending on the runtime choice made. Via T_2, the call-for-proposal repeats. Via T_3, the contract is struck and committed, involving the purchase acceptance (E_1) and final commit (C). Via T_4, the contract is struck and committed, involving the purchase acceptance (E_2) and final commit (C).

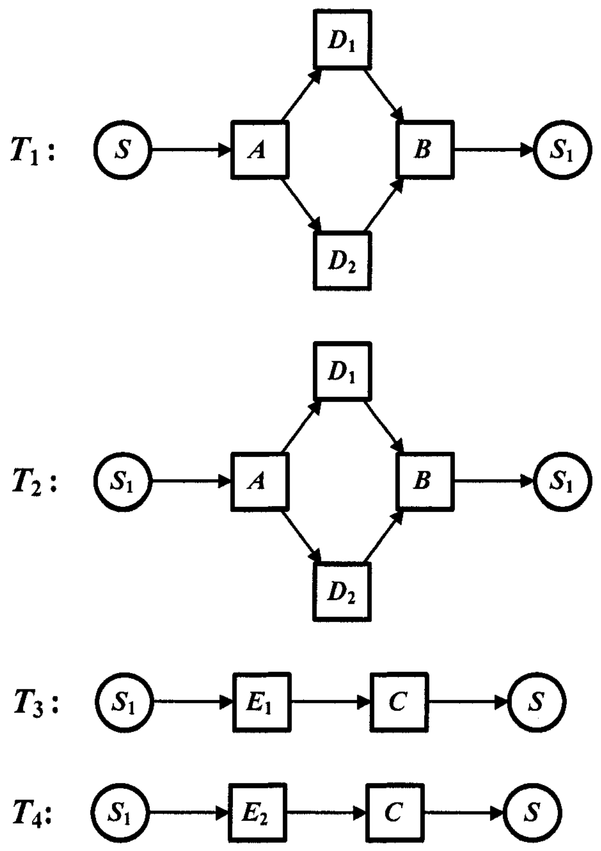


Figure 17: The POMSET automaton used to model the ordering requirements of the product buyer.

3.1.3 Unfolding of the automaton

Given a Pomset automaton, it is rather apparent that a behavior tree can be unfolded from the initial slot of the automaton by applying every rule whose start slots have been reached in the unfolding performed so far. For example, the unfolding of the automaton in Figure 17 can lead to a behavior tree partially shown in Figure 18. The partial behavior tree in Figure 18 actually shows eleven distinct partially ordered behaviors of the system.

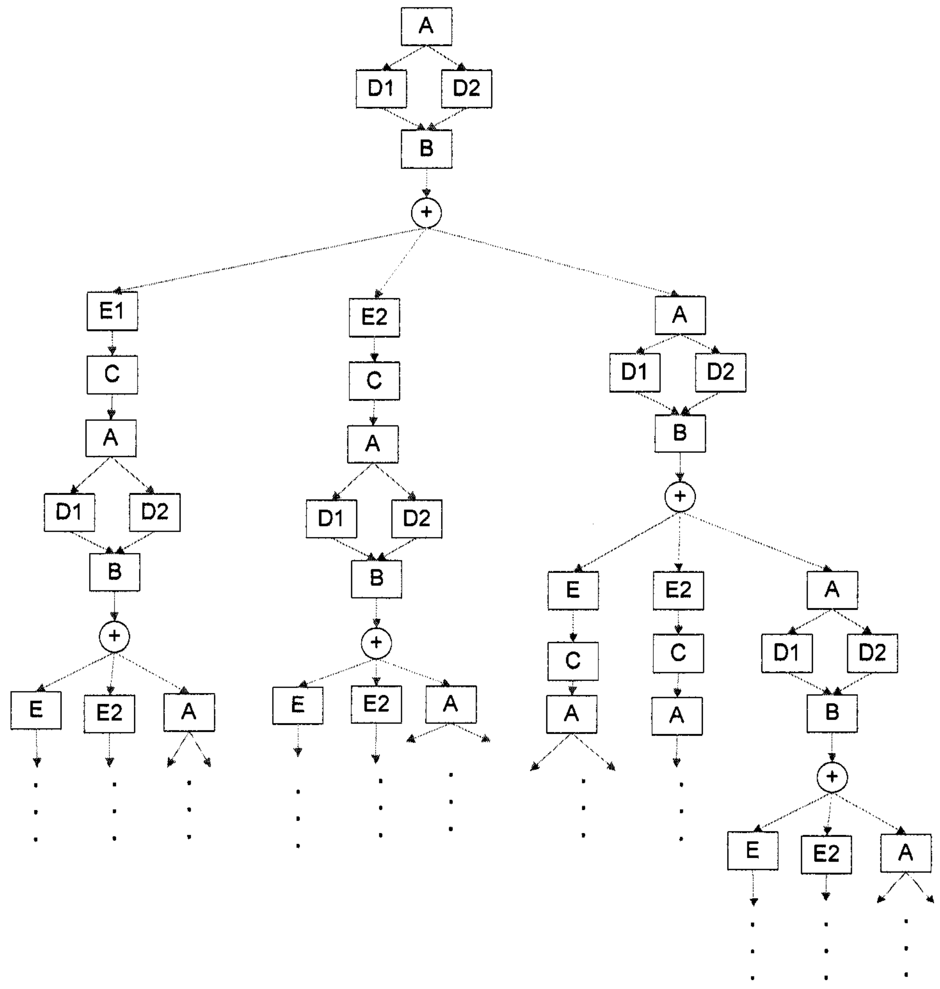


Figure 18: Behavior tree of partial orders in the product buyer application by unfolding its automaton.

3.1.4 Checking the run against the automaton

Generally, checking a run against a Pomset automaton is easy: the run Pomset must contain more ordering than the requirement Pomset. Mechanization of this will be rather straight forward, and its details will depend on the details of the automaton syntax.

Putting this into practice, Figure 19 shows an example run of a "product buyer".

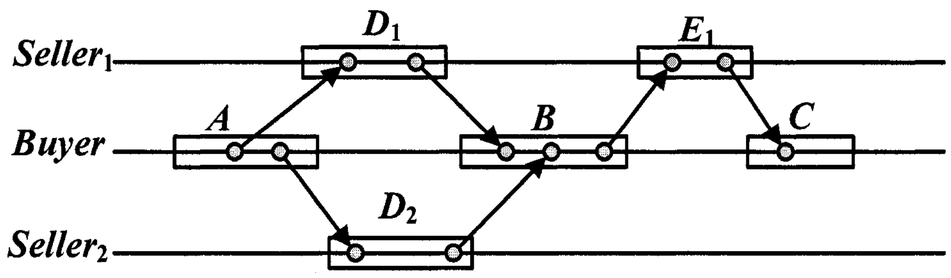


Figure 19: An example of an atomized run of "product buyer".

The example run shown in Figure 19 can be produced by the automaton by firing transition T_1 then transition T_3. So, it is an admissible run.

3.1.5 Specification language

The Pomset model specifies a set of allowable partial orders of a program; however, the atoms in the partial orders depend on the atomization of the program. In our work, the atomization of a run uses maxi atoms.

A *relevant variable* is a program variable which is used to define a global property of the distributed program. The identification of an atom is by means of 'relevant variables', which are variables modified within an atom. The correspondence between atoms in the Pomset automaton and atoms in the execution of program code is established by means of relevant variables. Specifically, an atom in an execution run is identified with a label corresponding to the set of relevant variables that are modified by the relevant events in the atom. In the instrumentation, a user of the monitoring and checking tool is expected to specify the Pomset automaton using labels with relevant variables.

For example, relevant variables for the E-market atoms in Figure 17 are:

A: {Buyer. currentOrder},

D_1: {Seller_1. receivedOrder},

D_2: {Seller_2. receivedOrder},

B: {Buyer. purchaseOrder},

E_1: {Seller_1. payment},

E_2: {Seller_2. payment},

C: {Buyer. purchased}.

As an implementation language, the finite Pomset fragment contained in each rule of the Pomset automaton is specified as a set of linearized threads which when weaved together (via union of sets) form the Pomset fragment. For example, the Pomset fragment in T_1 of Figure 17 is expressed as the following two threads between the start slot S and end slot S_1 as follows:

Thread 1: S: A; D_1; B : S_1

Thread 2: S: A; D_2; B : S_1

Together, these two threads weave together to form the transition rule T_1. This thread decomposition of each Pomset fragment enables arbitrary partial orders to be used within each transition rule.

As a result, the Pomset automaton of Figure 17 can be equivalently expressed in the form of a list of decomposed threads, as shown below:

Start Slot Set	Rule	Threads	Ending Slot
{0}	1	A; D_1; B	1
	1	A; $D2$; B	1
{1}	2	A; D_1; B	1
	2	A; $D2$; B	1
{1}	3	$E1$; C	0
	3	$E2$; C	0

Using labels with relevant variables, the user is expected to specify the Pomset automaton as follows.

Start Slot	Rule	Threads	Ending Slot
{0}	1	{Buyer. currentOrder} ;{ Seller_1. receivedOrder}; {Buyer. purchaseOrder}	1
	1	{Buyer. currentOrder} ;{ Seller_2. receivedOrder}; {Buyer. purchaseOrder}	1
{1}	2	{Buyer. currentOrder}; {Seller_1. receivedOrder}; {Buyer. purchaseOrder}	1
	2	{Buyer. currentOrder}; {Seller_2. receivedOrder}; {Buyer. purchaseOrder}	1
{1}	3	{Seller_1. payment}; {Buyer. purchased}	0
	3	{Seller_2. payment}; {Buyer. purchased}	0

3.2 Correctness of an atom

3.2.1 The model

The key emphasis in our approach is a clear distinction of two separate concerns: the causal requirement among atoms, and the correctness in computation arising from each atom. With this separation of concern, we do not have an exponential number of states with which correctness must be checked. Instead, the requirement involves only a number of states equal to the number of atoms in the specification.

Assuming that the ordering between atoms is satisfied separately as modeled in Section 3.1, the computational correctness of an atom can be specified using a predicate on the state reached upon execution of each atom, without taking into account the 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.

3.2.2 Specification language

We specify the computational correctness through predicates, which involve getting the value of the predicate variables (relevant variables that define predicates). Therefore, the specification language involves the user input to specify the predicates, and the user input to provide the set of predicate variables in the code space. It should be noted that predicate variables are different from atom variables used in identified atoms. In fact, predicate variables may be modified outside relevant atoms in the Pomset automaton.

In the implementation, a predicate to be satisfied upon completion of an atom is a Java expression to be evaluated at the (minimal) global state associated with that atom in the run. As an illustration, consider atoms B and D_i in the product buyer example. The corresponding predicates are listed below:

Atom	Predicate
D_i {Seller_i. receivedOrder}	Seller_i. product = = Buyer. product
B {Buyer. purchaseOrder}	Buyer. price = = min (Sell_1.price, Seller_2.price)
E_i {Seller_i. payment}	Seller_i. price = = Buyer. price

Chapter 4 System Design and Algorithm Design

The tool we have developed is for monitoring and checking program codes during their runtime execution. This chapter contains the details of the algorithms and system design of the tool for the purpose of monitoring and checking. Given a Pomset automaton as the ordering requirement and a set of predicates written in Java expressions as computational requirements, the tool checks if a run satisfies both requirements.

We start the description of the high-level design by introducing several assumptions in our design. First, we assume that programs under test are in JADE. Second, atoms are detected at runtime by user defined atom variables. Third, ordering requirements are specified by user defined automaton. Last, the automaton is scalable and allows nesting of protocol sessions. That means our automaton can specify dynamically growing systems rather than statically determined set of agents.

Based on the above assumptions, the tool is composed of two parts: the distributed monitor module, and the global checker module. The system architecture is shown in Figure 20.

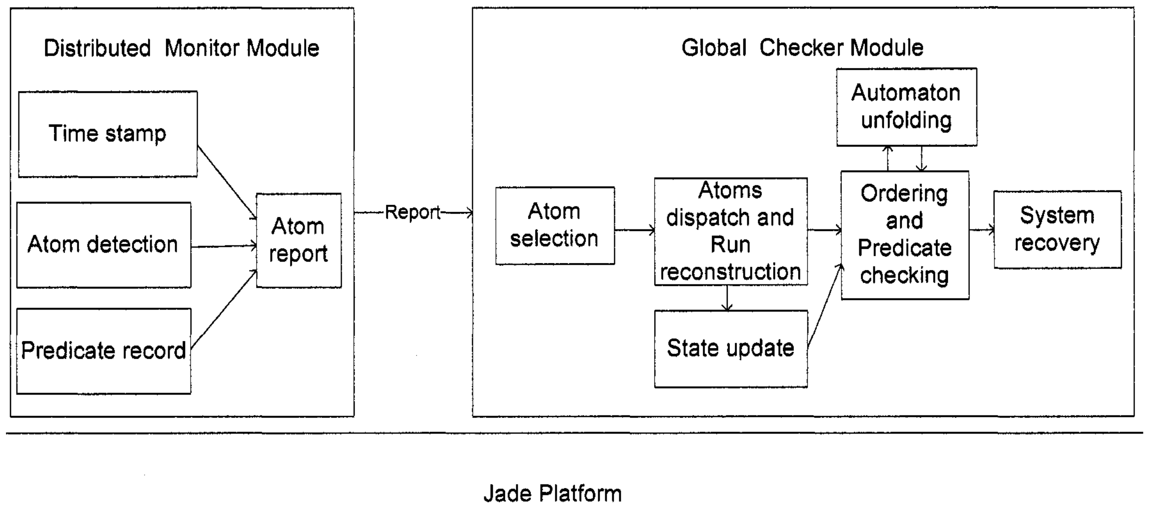


Figure 20: System Architecture

The Distributed Monitor is located in each agent object, and maintains a global time stamp. It is responsible for the detection of atoms and the recording of predicate variables. It reports to the global checker the detected atoms with the values of updated variables. The Distributed Monitor is designed to be event driven. Relevant events include time-stamp events, message tagging events, predicate events, atom start/end events, and atom and states report events. Time-stamp events modify the global timestamp of each atom, which involve the receive events and the first atom variable modification events in the atom. Message tagging events tag the time stamp to each atom, which involve the first send event in an atom. In the case of no send event in an atom, the first receive event of the next atom would be the message tagging event. Predicate events are the events that change the values of the predicate variables, which involve the execution of predicate statements. Atom start/end events are the receive events and the send events. Atom and states report events report the relevant atoms with their time stamp and the values of predicate variables to the global checker, which involve the first atom variable modification events in the atom.

As shown in Figure 20, the Distributed Monitor is composed of the time stamp component, the atom detection component, the predicate record component, and the atom report component. The time stamp component is responsible for maintaining the global time stamp and tagging them to the atoms. The atom detection component is responsible for detecting the start and the end of the atoms and the identification of those atoms using atom variables. The predicate record component is responsible for recording the values of the predicate variables. Taking the information from the time stamp component, the atom detection component, and the predicate record component, the atom report component is responsible for reporting the atoms with the time stamp, and the values of predicate variables to the global checker.

The Global Checker receives reports from the distributed monitors, performs the checking for both ordering and predicates on-the-fly. It is composed of the atom selection component, the automaton unfolding component, the atom dispatch and run reconstruction component, the state update component, the ordering and predicate checking component, and system recovery component. The atom selection component is responsible for choosing the ready atoms from the received atoms and sending them to the atom dispatch and run reconstruction component. We assume our automaton can have arbitrary number of starters. The atom dispatch and run reconstruction component is responsible for dispatching the ready atoms to their corresponding starters and reconstructs the run for each starter. Upon the receiving of the atoms, the state update component updates the values of predicate variables with the corresponding time stamp in the maintained database; the ordering and predicate checking component selects the atom set to be checked along with the unfolded the automaton to run the ordering checking. If the ordering requirement is satisfied, the automaton unfolding component unfolds the automaton by the allowable atoms and selects the enabled transition rules; the ordering and predicate checking component runs the predicate checking using the updated values of predicate variables. Whether there is an atom check failure or a predicate check failure, the system recovery component is responsible for taking recovery actions to avoid crashes of the system.

4.1 Design of the distributed monitor

4.1.1 Distributed monitor module architecture

The distributed monitor maintains a global time stamp locally, and is responsible for atoms detecting and variables recording. Its architecture is shown in Figure 21.

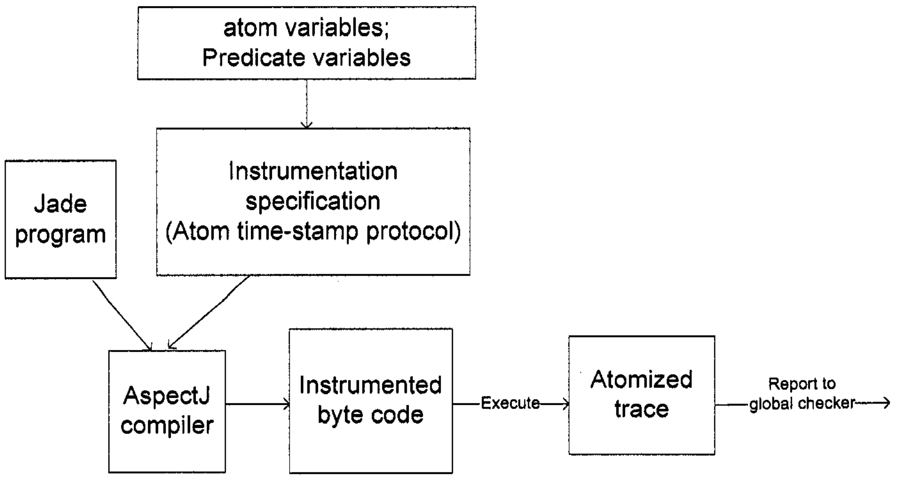


Figure 21: Distributed Monitor Module

The user provides the atom variables and predicate variables along with the source code (Jade program) for the local monitors. An atom time-stamp protocol is designed for the purpose of identifying all the atoms according to the atom variables provided by the user, and detecting the time stamp of the atoms as well as the values of predicate variables with their respective time stamps. The monitors take the atom time-stamp protocol as the instrumentation specification to be implemented in an AspectJ class. Then with the help of AspectJ compiler the AspectJ class will be weaved into the source codes and the instrumented byte codes are generated. After the execution of the instrumented byte codes, the local monitors report the atomized trace to the global checker.

4.1.2 Atom time stamp protocol

The critical part of the distributed monitor module design is the design of the atom time-stamp protocol as the instrumentation specification, which is implemented in the

AspectJ class. Algorithm 1 involves an atom time stamp protocol with the detection of relevant atoms and causal relationship among detected atoms. The input of the algorithm is the source code of the program with the user defined atom variables and predicate variables. The output is the atoms labeled by a set of atom variables, their time stamps, and the value of predicate variables with their respective time stamps. The time stamp is defined for the purpose of identifying the potential causality among the detected atoms and identifying the minimal state of each atom.

To identify the minimal state of an atom, the atom should be reported together with the values of the predicate variables that it owns with the time stamp. So in Algorithm 1, we report the previous atom, not the current one, when a new atom is detected. Therefore at the atom state, it records any changed value of the predicate variables. In the algorithm 1, the global time stamp `T` is designed as a two-tuple so that `n` in `<t1,…, tn>` is a variable that can dynamically grow.

4.1.3 Implementation assumptions

User provides the class name and the variable name of each atom variable in the atom variables input file, and the class name and the variable name of each predicate variable in the predicate variables input file. In the current implementation, we assume that these relevant variables only could be data members of a class and their types could be primitive variables and reference variables. If the relevant variables are local variables of methods, user needs to make them global.

Algorithm 1 Atom time-stamp protocol with detection of a relevant atom.

```
Process i:

Initialization:
            T = <t1,..., ti,.., tn> = <0,0,…0>;
            new_atom = true;
Repeat:
before a current (caught) event is performed do case of

atom event:
            {
            record variable;
            if new_atom then
                  {
                  <t1,.., ti, … tn> = <t1, .. ti + 1, … tn>;
                  new_atom = false;
                  report T* and values of relevant variables;
                  }
            }
predicate event:
            {record variable;}
send(m):
            {
            if T is never tagged on a message via this
            channel then send (m, T) instead of just send(m);
            if new_atom = false then T* = T;
            new_atom = true;
            }
receive(m, T'):
            {
            if new_atom = false then T* = T;
            new_atom = true;
            T = max(T, T');
            }
until termination of process;
report T* and values of relevant variables;
```

In the above T* is the timestamp of the previous relevant atom (not current one). A relevant atom is reported when either a new relevant atom is detected or when the process has terminated.

Another constraint is that the assignment statements of the relevant variables should be outside of the field declaration and the constructor of Behavior class because before the Behavior constructor is executed the Behavior object cannot get the reference of its agent which calls the behavior.

4.2 Design of the checker

The global checker runs the checking algorithm upon receiving the atoms, and reports any errors in either ordering or predicates. The checking algorithm involves two parts of checking: ordering checking and predicate checking.

4.2.1 Ordering Checking

4.2.1.1 Ordering checking algorithm

Ordering checking is to check a run of a program against its Pomset automaton. If the run Pomset contains more ordering than the requirement Pomset, the run is admissible. Otherwise, the run is not admissible. Algorithm 2 shows the ordering checking algorithm. The input of the algorithm is a run Pomset and a Pomset automaton. The output of the algorithm is an ordering error which is reported if the run Pomset is not admissible. We use this checker algorithm to unfold the automaton by keeping track of all activated slots and transitions rules. Notice that with the semantics chosen for the automaton, each enabled start slot set can unfold exactly one of its transition rules.

Algorithm 2 ordering checking algorithm.

```
Let I be the run pomset and P be the given automaton.
Let $I^0$ be the current set of enabled atoms in I.
Let $P^0$ be the current set of enabled atoms (that can evolve from the automaton P, according to
the pomset fragment/slot semantics).

repeat
        if $I^0$ is contained in $P^0$
            then prune $I^0$ from I and advance $I^0$ in P
            else report error and terminate
until    I = null or error has been reported
```

4.2.1.2 Some difficulties and their solutions

i. Scaling and nesting of protocol sessions

The scalability of an automaton means that the automaton can specify programs in which the interactions can scale up in agents, including the nesting of protocol sessions. In other words, we have dynamically growing systems rather than statically determined set of agents. The dynamic changing of agents requires the automaton to be scalable as well.

For example, when the previous product-buyer application (which involves a buyer and two seller agents) is improved to be a dynamically growing program, it will require our automaton to be scalable. In other words, the buyer agent calls for proposal from a set of seller agents (Seller$i$, $i \leq n$) instead of two, and accepting one of these before completing the rest of the protocol. The scaling has to do with the fact that the set of broker agents may grow (and is decided only at runtime), and the automaton has to be able to specify it.

Hence, we can not specify the automaton by specific agents and slots; instead, we incorporate index variables with slots so that a slot can scale up to allow for a variable degree of concurrency in the specification. These index variables will get their values during the run. And then using these runtime-determined information, we can unfold the automaton properly and do the checking.

For example, in Section 3.1.5, the Pomset automaton of product-buyer application has been specified in the form of a list of decomposed threads. However, this automaton is not scalable since the involved agents are statically determined (Buyer, Seller1, and Seller2).

To improve the automaton to be dynamically scalable, we incorporate an index variable (i) to be associated with the end slot in Rule 1. As a result, Rule 1 is specified as follows:

Start Slot Set	Rule	Threads	Ending Slot
0	1	$A; D_i; B$	1. (i)

The above Rule 1 represents undetermined number of transition rules. The index (i) will be determined by a set of seller agents when the program is executed. And each seller agent will conduct an instance of Rule 1. Correspondingly, Rule 2 and Rule 3 are specified as follows (where "+" is used to represent a conjunctive slot):

Start Slot Set	Rule	Threads	Ending Slot
1. (i). +	2	$A; D_i; B$	1. (i)
1. (i). +	3	$E_i; C$	0

Moreover, we assume that the scaling has to do with multiple levels of nesting of protocols too. For example, when the brokers receive the call for proposal from the client agent, they may propose different sale prices and the client agent accepts one of these prices before completing the rest of the protocol; or they call for proposals from other wholesalers through separate protocols. Then, after accepting one of the proposed prices offered by the wholesalers, the brokers propose their prices to the client agent, and the client agent accepts one of these prices (from brokers) before completing the rest of the protocol. In this case, the program involves two levels of nesting of protocol sessions: one is between the client agent and its brokers, and the other is between a broker agent and its wholesalers. By incorporating indexing of slots, we can solve this nesting of protocol sessions problem as well. As a result, protocols can be bound with their nested protocol in a specification too. A more complex example to illustrate it is given in Chapter 5.

In addition, our automaton can have arbitrary number of starters. In other words, the initial slot S_0 is actually a set $< S_{01}, S_{02}, S_{03}, \ldots, S_{0n}>$, which could have as many number of initial slots as a program needs to use. Therefore, the automaton has a scalable set of transition rules initially, and different runs fire different starters and follow separate rules.

ii. Getting the concurrent atom set

Why do we need to get the concurrent set of enables atoms I^0, and check the entire set in

Algorithm 2? This issue concerns the correctness of the checking algorithm. The following simple example will reveal the reason.

Suppose the run contains two atoms, $A \parallel B$, and the specification automaton contains two ordered atoms, [A; B]. If the checker selects A from the run and checks it against the automaton, it will not report an error and proceed to check B without reporting any error. Obviously, the run does not satisfy the specification; but the checker fails to report the error. The correct checker algorithm should identify the set of enabled atoms in the run. In the example, it will be $\{A, B\}$ and check if the entire set is enabled in the specification (without doing so one at a time). Then it will reveal that A but not B is enabled in the specification and report the failure of the run.

This algorithm is implemented without any problem in a previous off-line version of the checker because the entire concurrent atom set is easy to get using the time stamp since all the atoms of the run are known. However, when we want to do it on-the-fly, we can not ensure that the atom set obtained using time stamps among the reported atoms is an entire set of the concurrent atoms.

For example, the ordering requirement in Figure 22 is: $a \rightarrow b$; $b \rightarrow (c \parallel x)$. In run (a) and in run (b), x' and x'' are reported after b. In scenario (a), c and x' are received by the checker as a concurrent set to be checked. The checker reports no error and the run does satisfy the specification. But in scenario (b), c and x'' are received by the checker as a concurrent set to be checked. The checker reports no error. Obviously, the run does not satisfy the specification. So the checker has failed to report the error. To avoid that, the checker must get all the concurrent set members in the run Pomset at the checking point, and check them together. In scenario (b), the checker does not check *b* until *x*" is received, and then it should check *b* and *x*" together as a set.

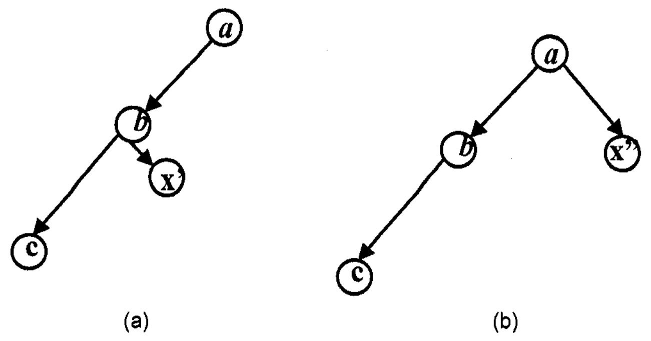


Figure 22: Reported atom set

Algorithms 3 detailed order checking algorithm

```
In each run, while the automaton is unfolded, when multiple rules
are enabled, we get a set of concurrent enabled atoms, set1 =
{atom1, atom2 …}, set2 = {atom3, atom4 …}, set3 = {atom5}.

Get an atom from runAtoms (partial ordered);
If the atom belongs to any of the above set,
Then
     If the set is completely attained,
     Then check the set, and delete the set;
     Else store the atom.
Else
      If the atom concurrent with the stored atoms,
      Then check it;
      Else leave it in the runAtoms
```

The detailed ordering checking algorithm is shown above as Algorithm 3 which ensures to check the complete set of the concurrent atoms of the run Pomset.

iii. Unfolding the automaton

Algorithm 4 is a more detailed ordering checking algorithm than Algorithm 3. It gives the detailed design to solve the problem of how we prune an atom of the run Pomset and how we advance the transition rules. In Algorithm 4, when an atom has been checked to be an enabled atom, the automaton will be updated. As a result, some atoms are identified with its agent id and its sender ids, and some index variables in the start slot may be identified by the agent ids. Algorithm 5 gives the details of the `updateAutomaton ()` method. The input of the algorithm is an atom which has been checked to be an enabled atom, and the output of the algorithm is an updated automaton.

iv. Merging of atoms in a run

In some situations we need to break an atom into a sequence of atoms to specify the automaton generally; for example, to specify the choices. But an actual run may have more ordering than the specification. The checker must be able to do the checking correctly under such scenarios. For example, let the specification be $\{x\} \rightarrow \{z\}$ weaved with $\{x\} \rightarrow \{y\}$. The actual run $\{x, y\} \rightarrow \{z\}$ satisfies the specification in this case and the checker should not report error. In order to do this correctly when checking $\{x, y\}$, if $\{x\}$ is enabled, then decompose the atom $\{x, y\}$ into two atoms $\{x\}$ and $\{y\}$ and check them.

Algorithm 4 detailed ordering checking algorithm with the unfolding of the automaton

```
Repeat:
Get an atom from the atom set;
isEnabledAtom = false;
Prune one atom from runAtoms;
If the atom has predicate variables, record it with the vector clock;
if the atom is not a choice atom
then {
        get enabled atoms from the enabled Rules;
        if the atom is one of the enabled atoms
        then{
                isEnabledAtom = true;
                updateAutomaton(atom);
                if the atom is the last atom in the thread of enable rule
                then {
                        remove the enabled rule;
                        get the enabled slot;
                        if the enabled slot is a conjunctive slot
                        then {
                                record the enabled slot;
                                if any conjunctive start slot set is satisfied
                                then get enabled rule;
                                }
                        else get enabled rule from the enabled slot;
                        }
        else {
                    remove the atom from the thread of the enable rule;
                    specify the next atom's agentID and senderID;
                        }
        }
else {
        get enabled atoms from the enabled choice rules;
        if the atom is the only atom in the enabled rule
        then {
                if the atom is one of the enabled atoms
                then{
                        isEnabledAtom=true;
                        updateAutomaton(atom);
                        if the end slot contains wildcard, replace it;
                        add the enabled the rule from the end slot;
                        remove the choice slot set;
                        break;
                    }
            }
        else {
                if the atom is one of the enabled atoms
                then{
                        isEnabledAtom=true;
                        updateAutomaton(atom);
                        prune the atom from the rule;
                        specify the next atom's agentID and senderID;
                        add the enabled choice rule to the enabled rules;
                        remove the choice slot set;
                        break;
                    }
                }
       }
if(isEnabledAtom)
then{
        checkPredicate(atom);
        updateEnabledRules(atom);
        }
else {
        report ordering error;
        throw an ordering error exception;
       }
```

Algorithm 5 automaton update algorithm upon checking of an enabled atom

```
If the atom is the last atom in the rule
Then
      If multiple receivers
      Then {
            Specify the next atom's senderID;
            Specify the next atom's agentID;
            Specify all the wildcard in the start slot;
            Specify this atom's receiverIDs;
            }
      Else {
            Specify the next atom's senderID;
            Specify the next atom's agentID;
            Specify this atom's receiverID;
                                  }
Else
      Specify this atom's receiverID;
      Specify next atom's agentID and senderID.
```

4.2.2 Predicate checking

For an atom to be pruned, the predicate associated with it (specified in the input file) should be checked with the recorded values of the relevant variables. The predicate is specified by Java Boolean expressions. Therefore, our strategy for predicate checking is simple cut and paste. In practice, these predicates take the form of method calls on the corresponding atoms and are accomplished via the Java reflection mechanism. The method calls take the predicate variables as their parameters.

4.2.2.1 Predicate checking algorithm

Algorithm 6 is the predicate checking algorithm. The input of the algorithm is a run Pomset and a predicate input file. And the output of the algorithm is predicate error.

Algorithm 6 Predicate checking algorithm

```
Produce the Methods class from predicate input file
Get values of the predicate variables
Pass the values and invoke the corresponding method
If the method call returns false, report error.
```

4.2.2.2 Some difficulties and their solutions

The main difficulty lies in the correct requirement of predicate checking. This involves two issues. One is how to select the proper relevant atoms among the atoms with the same ID, in the case of a scalable automaton (in other words, how to determine the causal cone of the concerned atom). The other issue is how to collect most recent values of the predicate variables. The solution to the first issue lies in the semantics of the model. As we know the scalability of the automaton is attained by producing different instances of a rule. The correspondence between ordering checking and predicate checking is that they follow the same transition rule. Thus the correct requirement to choose the proper predicate variable is to choose the one which is in the same rule as the checked atom. To solve the second issue, we maintain a database associating each agent process with its predicate variables' value and the corresponding time stamps.

4.3 Complexity of the checker

The complexity of our checking algorithm is composed of three parts: the complexity of the atom time-stamp protocol, the complexity of predicates checking and the complexity of the reachability checking. Suppose N is the number of the atoms, m is number of relevant variables which are modified inside or outside of each atom, and E is the number of edges in the complexity graph. The complexity of the atom time-stamp protocol would be $O(mN+E)$. Suppose p is the complexity of the predicate of each atom. The complexity of the predicates checking would be $O(pN)$. During the unfolding of the automaton, we visit every node (atom) once and visit every edge once in the partial order behavior tree in order to check the ordering. Therefore, the complexity of the reachability checking is $O(N+E)$. As a result, the complexity of our checking algorithm is $O((m+p)N + E)$.

Chapter 5 A sample use of the tool

This chapter shows a sample use of our tool through the example E-market application. By showing a well selected snapshot of some parts of the code, we explain how to choose the atom variables for the purpose of identifying the atom in design space. We also explain how to specify the predicates to be checked using predicate variables. Finally, we give two sample screenshots of the output to show the information users can get from the result of checking.

In order to use the tool, the user needs to specify the distributed computation as a Pomset automaton and the properties as predicates. In order to do that, the user has to identify the atom variables and predicate variables. Specifically; first of all, the user has to derive the design specification of a program in terms of the roles and their relationships; second, identify all the atoms in the program; third, specify the Pomset automaton; fourth, identify the atom variables to label the atoms; finally, identify the predicate variables and specify the predicates.

5.1 Deriving Design Specification

The example E-market application has already been described in Section 3.1.1. According to the design specification, Table 1 summarizes the roles and describes their responsibilities and objectives, and Figure 11 is the role diagram of the application.

5.2 Atomization

The atoms that can be executed by an application can be identified by constructing the atomized interaction diagram based on the design specification.

Figure 23 shows the atomized interaction protocol diagram for the E-market application.

Rectangles are used to depict atoms; each atom has a corresponding label.

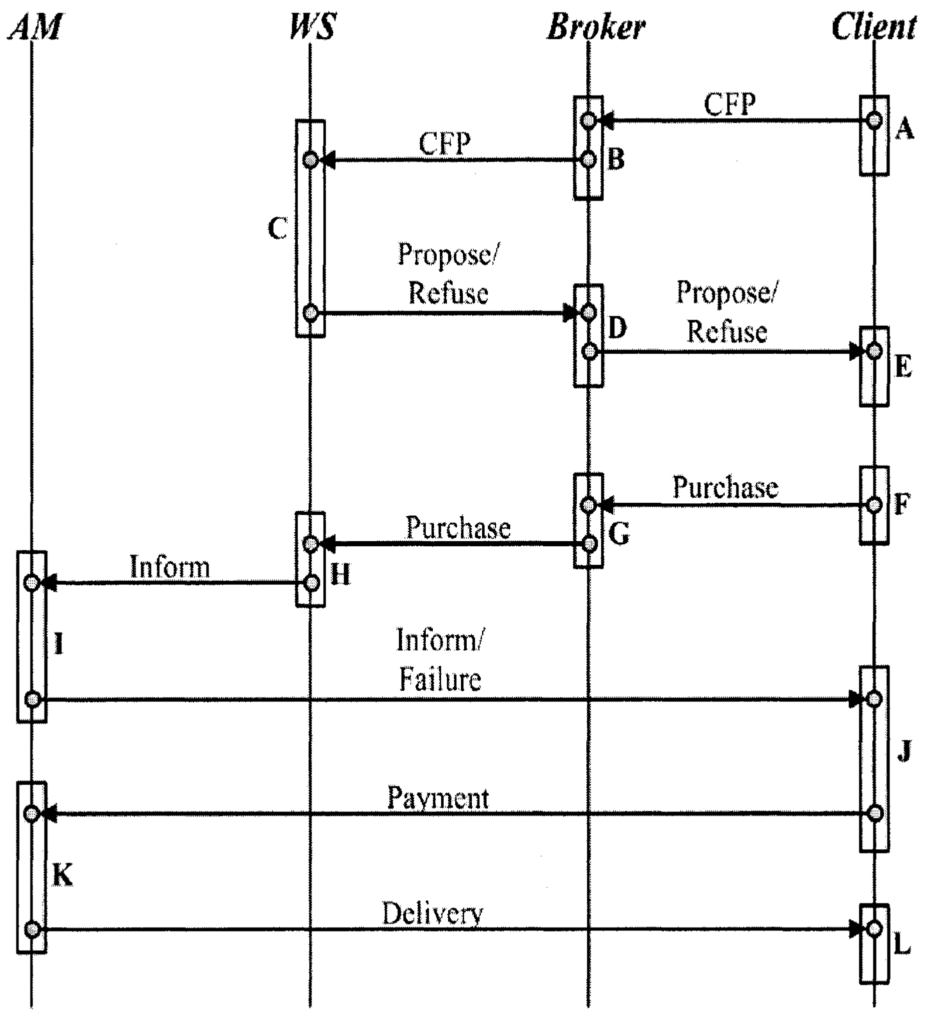


Figure 23: The atomized interaction protocols diagram of the E-market application.

Once this is done, we have the set of all atoms that can be executed by the application. Table 2 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.

Table 2: The set of atoms executed by the E-market application

Atom	**Agent**	**Description**
A	Client	Sending a call for proposal to brokers.

B_i	Broker	The i^{th} Broker receives the client call for proposal and sends a call for proposal to wholesalers.
C_{ij}	Wholesaler	The j^{th} wholesaler of the i^{th} broker receives the broker request; if the requested product is available, the wholesaler will send a proposal for the broker, otherwise he will send him a refusal message.
D_i	Broker	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.
E	Client	The client receives all the proposal/refusal messages from the brokers and selects the best proposal if any.
F	Client	The client sends a purchase message to the broker who has provided the best proposal.
G	Broker	The broker receives a purchase order from the client and forwards it to the corresponding wholesaler.
H	Wholesaler	The wholesaler receives a purchase order from the broker; if the requested product is available he will approve the purchase order, otherwise he will reject it; in any case he will forward the result to the account manager
I	Account	The account manager will receive the purchase order; if it

	Manager (AM)	is approved, he will send an invoice to the client, otherwise he will send a Failure message to the client.
J	Client	The client receives a message from the AM, this message can be an inform message or a failure message. If it is a failure message then the client will make another call for proposal 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)
K	AM	The AM receives the proof of payment and delivers the confirmation to the client.
L	Client	The client receives the confirmation.

Figure 24 shows an example of an atomized run of the E-market application. In this run we have assumed that there are two brokers B1 and B2 known to client C. Brokers B1 and B2 can contact wholesalers W1 and W2. 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.

5.3 Pomset Automaton Specification

The atomized interaction protocol of the E-market application is captured by the Pomset automaton shown in Figure 25. We assume that there are i Brokers known to the client, and for each broker knows j wholesalers. There are 8 transitions in this Pomset automaton. The three vertical dots appearing in some of the transitions indicate 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 also involves atoms $\{B_2, \ldots, B_{i-1}\}$. Moreover, there is an edge between atom A and each atom in $\{B_2, \ldots, B_{i-1}\}$, and there is an edge from each atom in $\{B_2, \ldots, B_{i-1}\}$ to its corresponding slot, as is the case for the two atoms B_1 and B_i that are already depicted in transition T_1.

The Pomset automaton of Figure 25 can be expressed equivalent in the form of a list of decomposed threads, as shown below:

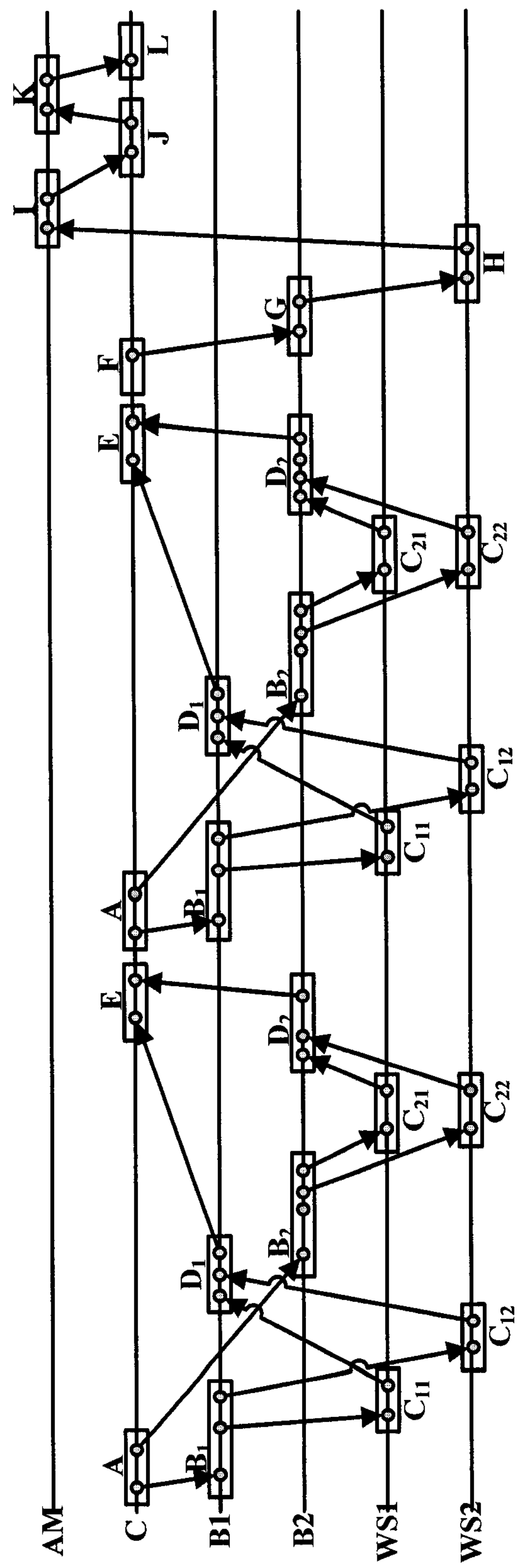


Figure 24: An example of an atomized run of the E-market application.

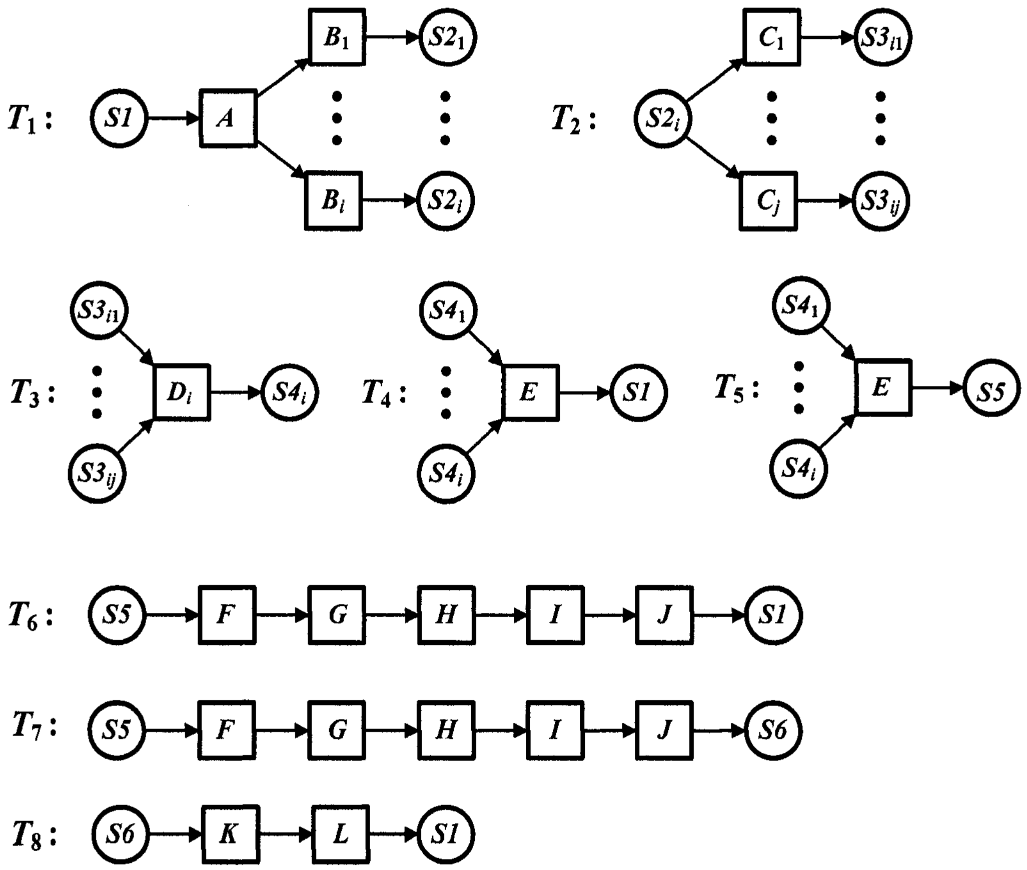


Figure 25: The POMSET automaton of the E-market application.

Start Slot Set	Rule	Threads	Ending Slot
1	1	A; B_i;	2. (i)
2. (i)	2	C_ij	3. (i). (j)
3. (i). (j).+	3	D_i	4. (i)
4. (i).+	4	E	5
5	5		1
5	6	F; G; H; I; J;	6
6	7		1
6	8	K; L;	1

5.4 Atom labeling

Each atom in

Figure 23 includes a label corresponding to the subset of atom variables that trigger relevant events in the atom. One of the difficulties in the use of our tool is choosing these atom variables properly so that the atoms labeled by the atom variables can be detected and checked during the run.

For the purpose of identifying the atom variables, we use the following selected snapshots of some parts of the source code to explain how to relate the code to the atoms in the specification. Atom A is responsible for sending a call for proposal to the brokers. It corresponds to the following code block in the `Client.RequestPerformer` class.

```
case 0:
  // Send the cfp to all brokers
  try {
  ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
  currentOrder=new Order(myAgent.getAID(),currentOrderID++);
  currentOrder.setOrderItems(Product);
  marker_currentOrder_items=1;
  cfp.setContentObject(currentOrder);
  for (int i = 0; i < BrokerAgents.length; ++i) {
    cfp.addReceiver(BrokerAgents[i]);
  }
  cfp.setConversationId("Product-trade");
  cfp.setReplyWith("cfp"+System.currentTimeMillis());
  myAgent.send(cfp);
 }
  catch (IOException ex) {
      ex.printStackTrace();
  }
  mt = MessageTemplate.or(MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
          MessageTemplate.MatchPerformative(ACLMessage.REFUSE));
  step = 1;
  break;
```

There are several variables involved (`cfp, currentOrder, mt, step`) in the code block. Any single variable or combination of them could be chosen as atom variables. However, there are some guidelines that can help the user to make a correct and efficient choice: (1) According to our definition of maxi atom, an atom may receive information from other processes, followed by relevant events, before sending information to other processes. In other words, the potential relevant statements should be between the receive event and the send event. Therefore, variables `mt` and `step` are not suitable to be the atom variables. (2) Consider class data members prior to local variables, so that the user saves the effort to make them global. Therefore, since `cfp` is a local variable in the try block, it is not the first choice. (3) Choose the variable set that can identify the atom and make the set as small as possible. At first, the user can try to choose one variable to identify the atom as long as this variable can identify this atom. This attempt may fail, when any single variable between the receive event and the send event in the atom is also modified in other atoms. Therefore, the user can not use it alone, but choose another variable together with it to identify the atom. In the worst case, when all variables in the atom can not identify the atom, the user has to use auxiliary variables.

Based on the above analysis, the user can choose the class data member, `currentOrder` which is before the send event, as atom variable to identify atom A. Then, the execution of the assignment statement of `currentOrder` is a relevant event which is captured by AspectJ during the run, and the variable name is recorded as the identification of atom A at the same time. The send event is captured by AspectJ too as an indication of the end of the atom. Therefore, abstract atom A in the design space is mapped to the atom variable `Client.RequestPerformer.currentOrder` in the code space. As a result, atom {`Client.RequestPerformer.currentOrder`} is the actual atom which the monitor receives during the checking.

As another example, Atom B is responsible for receiving the client call for proposal and sending a call for proposal to wholesalers. It corresponds to the following code block in the Broker class. The receiving event is in the behavior class `ClientRequestsServer`. If the received message does not equal to null, the behavior `ProcessClientRequest` (in which some variables are assigned followed by a send event) is also involved.

```
/**
   Inner class ClientRequestsServer.
   This is the behaviour used by broker agents
   to serve incoming requests for offer from Client agents.
 */
private class ClientRequestsServer extends CyclicBehaviour {

  public void action() {
      MessageTemplate mt =
          MessageTemplate.MatchPerformative(ACLMessage.CFP);
      ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
        addBehaviour(new ProcessClientRequest(msg));
    }
      else {
        block();
      }
  }
}  // End of inner class ClientRequestsServer
```

```
private class ProcessClientRequest extends Behaviour {
      private AID bestSeller;
      private int bestPrice;
      private int repliesCnt=0;
      private MessageTemplate mt;
      private int step=0;
      private ACLMessage msg1, BestProposal;
      Order receivedOrder;
      ACLMessage replyToClient;

      public ProcessClientRequest(ACLMessage mg){
          msg1 = mg;
      }

      public void action() {
        switch (step) {
        case 0:
          // Send the cfp to all whole salers
          ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
          for (int i = 0; i < WholeSalerAgents.length; ++i) {
            cfp.addReceiver(WholeSalerAgents[i]);
          }
          receivedOrder = new Order(null,0);
          try {
                receivedOrder = (Order) msg1.getContentObject();
              }
              catch (UnreadableException ex1) {
                ex1.printStackTrace();
              }
          try {
              cfp.setContentObject(receivedOrder);
            }
            catch (IOException ex) {
              ex.printStackTrace();
            }
          cfp.setConversationId("Product-trade");
          cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
          myAgent.send(cfp);
          mt = MessageTemplate.and(MessageTemplate.MatchConversationId("Product-trade"),
                                   MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
          step = 1;
          break;
```

Based on the previous guidelines, a good choice is to choose the class data member `receivedOrder` as atom variable to identify atom B. Then, the execution of the assignment statement of `receivedOrder` is a relevant event which is captured by AspectJ during the run; and the variable name is recorded as the identification of atom B at the same time. The send event is captured by AspectJ too as an indication of the end of the atom. Therefore, abstract atom B in the design space is mapped to the atom variable

`Broker.ProcessClientRequest.receivedOrder` in the code space. As a result, atom {`Broker.ProcessClientRequest.receivedOrder`} is the actual atom which the monitor receives during the checking.

The mapping from the abstract atoms of the atomized protocol into the set of atom variables in the code space is shown in Table 3.

Table 3: Atom labeling of E-market application

Atom	**Atom variables**
A	{`Client.RequestPerformer.currentOrder` }
B_i	{`Broker.ProcessClientRequest.receivedOrder`}
C_{ij}	{`WSaler.BrokerRequestsServer.receivedOrder`}
D_i	{`Broker.ProcessClientRequest.replyToClient`}
E	{`Client.terminate`}
F	{`Client.RequestPerformer.PurchaseOrder`}
G	{`Broker.ClientPurchaseOrdersServer.receivedOrder`}
H	{`WSaler.BrokerPurchaseOrdersServer.receivedOrder`}
I	{`AccountsManager.WholeSalerRequestsServer.receivedRequest`}
J	{`Client.RequestPerformer.Payment`}
K	{`AccountsManager.PaymentOrdersServer.Payment`}
L	{`Client.RequestPerformer.Purchased`}

5.5 Predicate specification

From the design specification, we can identify the predicate that captures the correctness of the incremental function performed by each atom.

Suppose the application needs to check, at the end of atom K, when a Client sends a proof of payment to the AM after receiving an inform message from the Account Manager (Atom J), it must be that incrementally, the acceptance item of the Client should be equal to the acceptance item of the Account Manager. And the predicate specification should be as follows:

Atom:

```
Client.RequestPerformer.Payment
```

Predicate variables:

```
Client.RequestPerformer.Payment.items;
AccountsManager.WholeSalerRequestsServer.
   receivedRequest.items;
```

Predicates:

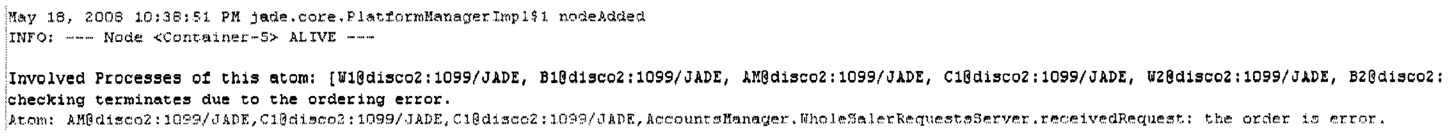
```
Client.RequestPerformer.Payment.items == AccountsManager.
   WholeSalerRequestsServer.receivedRequest.items;
```

5.6 Error reporting by the checker

The run of the application involves two clients; each client has two brokers, and each broker associates with two wholesalers.

5.6.1 A test case with a bug in ordering

We made up the bug by changing the source code. For example, we changed Atom F as follow. The client sends a purchase message to the account manager instead of the broker who has provided the best proposal. Therefore, Atom I will be the next atom in the run. However, it will violate the ordering requirement which specifies that Atom G should be the next atom. The checker reports the error shown below.

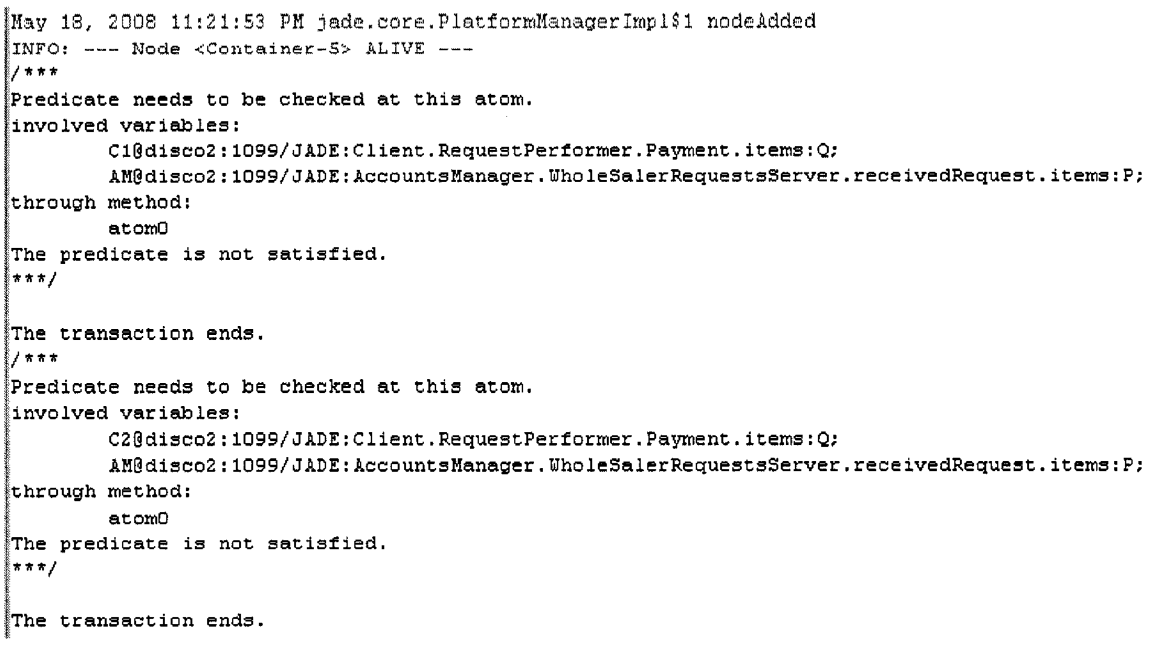


```
May 18, 2008 10:38:51 PM jade.core.PlatformManagerImpl$1 nodeAdded
INFO: --- Node <Container-5> ALIVE ---

Involved Processes of this atom: [W1@disco2:1099/JADE, B1@disco2:1099/JADE, AM@disco2:1099/JADE, C1@disco2:1099/JADE, W2@disco2:1099/JADE, B2@disco2:
checking terminates due to the ordering error.
Atom: AM@disco2:1099/JADE,C1@disco2:1099/JADE,C1@disco2:1099/JADE,AccountsManager.WholeSalerRequestsServer.receivedRequest: the order is error.
```

5.6.2 A test case with a bug in computation

In this test case, we set `Client.RequestPerformer.Payment` with item “Q” in Atom J. Since the original purchase item is “P”, the check reports errors for both transactions shown below.



```
May 18, 2008 11:21:53 PM jade.core.PlatformManagerImpl$1 nodeAdded
INFO: --- Node <Container-5> ALIVE ---
/***
Predicate needs to be checked at this atom.
involved variables:
        C1@disco2:1099/JADE:Client.RequestPerformer.Payment.items:Q;
        AM@disco2:1099/JADE:AccountsManager.WholeSalerRequestsServer.receivedRequest.items:P;
through method:
        atom0
The predicate is not satisfied.
***/

The transaction ends.
/***
Predicate needs to be checked at this atom.
involved variables:
        C2@disco2:1099/JADE:Client.RequestPerformer.Payment.items:Q;
        AM@disco2:1099/JADE:AccountsManager.WholeSalerRequestsServer.receivedRequest.items:P;
through method:
        atom0
The predicate is not satisfied.
***/

The transaction ends.
```

Chapter 6 Conclusions and Future Work

This thesis reported the design and implementation of a runtime verification tool for distributed programs written in the agent-based Jade platform. The ordering requirement of the concurrent program is modeled by a Pomset automaton specifying the correct partial order behaviors of the computation. The correctness of the computation performed by a set of source code statements that are expected to be performed atomically (an atom) is specified by a predicate on variables modified by the atom. The user is required to provide the Pomset automaton and the predicates to the tool using appropriate input files. The user also identifies appropriate atoms and atom variables in the computation so that the tool can observe the atoms using the specified atom variables at run time and check the validity of the predicates at the end of the atoms to verify the correctness of a run on-the-fly. The observation of atoms and variables is done through AspectJ.

Most importantly, the critical difference between our tool versus the other tools using interleaving semantics (e.g. Eagle, Hawk, MOP, and Java Pathfinder), which we have introduced in Chapter 2, is that we allow separate specification of the ordering and the correctness. This intrinsically permits checking them more efficiently without considering all the states during the computation which leads to a lot of problems (like the state explosion). In addition, we relieve the user from specifying the properties by temporal logic.

Another critical difference is that we use the minimal state of each atom for the purpose of identifying the values of predicate variables to be used, and check the predicates upon the completion of an atom. The other tools, in terms of message passing, have to specify the sliced states globally. In that case, each detected local event will update the global states and apply the formulas on them. Obviously, the global states used by the other tools are much more than the minimal states we need to check.

Moreover, we allow an atom to be a single event. This is possible because we are doing the abstraction of the events in the code space versus the event in the abstracted requirement space. Requiring a user to point to exact statement or exact event is a little bit restrictive because we could have the multiple set. In other words, not every occurrence of these statements or methods corresponds to the event, could be a combination of them. If any of the other tools hand-instruments the source code, like Eagle, it can have an event involving several statements or methods. But if the tool instruments the source code automatically, like Hawk and MOP, its event can only have one relevant statement or method. In addition, Hawk and MOP capture all the methods which have the same name as specified in the specification, although some of them may not be relevant to the properties.

To illustrate the differences more clearly, we give parts of the requirements specification of product-buyer application specified in Hawk. For example, a user needs to check the run in Figure 19 against the property that the required product the seller processed is equal to the product the buyer called for proposal.

In this application, send () and receive () methods are the only two methods that can be specified to capture the events. Obviously, these two methods can not identify all the events. Therefore, in order to identify all the events, the user needs to identify the send and receive methods first. As a result, the corresponding consistent cut is shown in Figure 26.

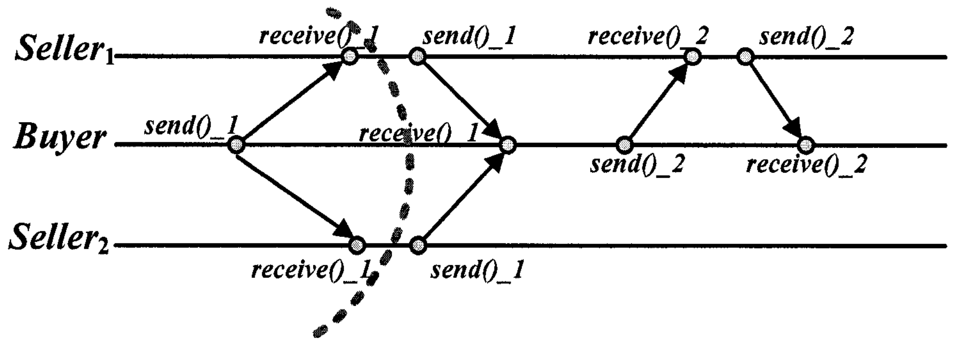


Figure 26 the consistent cut of the checked property

Hawk specification:

```
Var Seller1 s1;

Var Seller2 s2

Var Buyer b;

var ACLMesssage msg_b_snd1;

var ACLMesssage msg_s1_rcv1;

var ACLMesssage msg_s2_rcv1;

mon F =
Always(

   Until ([s1.receive_1() returns msg_s1_rcv1?] false,
   <b.send_1(msg_b_snd1?)> true) (msg_b_snd1 = = msg_s1_rcv1)

/\

   Until ([s2.receive_1() returns msg_s2_rcv1?] false,
   <b.send_1(msg_b_snd1?)> true) (msg_b_snd1 = = msg_s2_rcv1)

)
```

As a runtime verification tool, Eagle and Hawk check a run of the product-buyer application, whose state lattice may be the one shown in Figure 27 or the one shown in Figure 28. While Java Pathfinder can do more, it can simulate non-determinism. As a result, it checks all the possible paths of a program with the help of two capabilities: backtracking and state matching [JFP]. The state lattice of the program that is checked by Java PathFinder is shown in Figure 29.

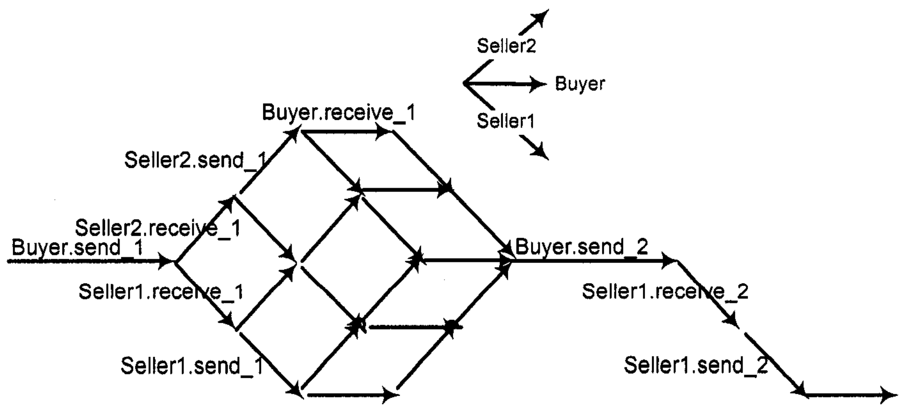


Figure 27: Atom state lattice of one run of product buyer application

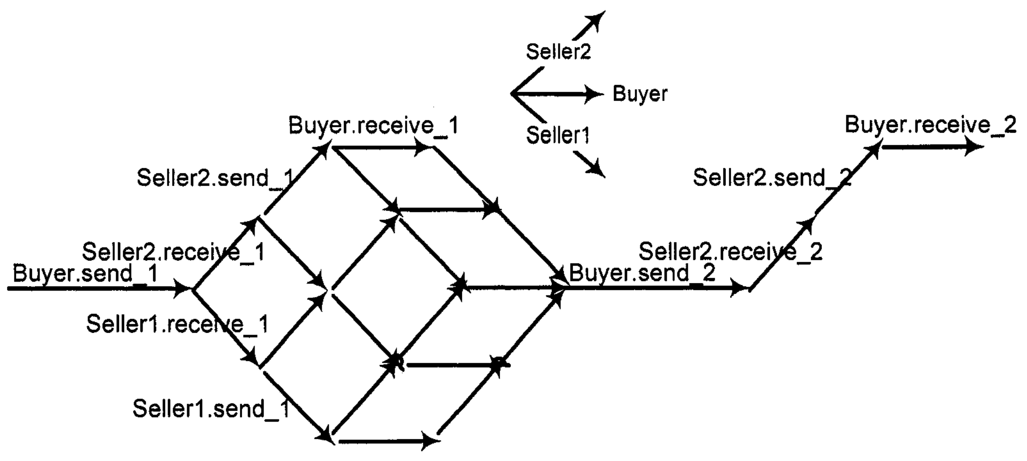


Figure 28: Atom state lattice of another run of product buyer application

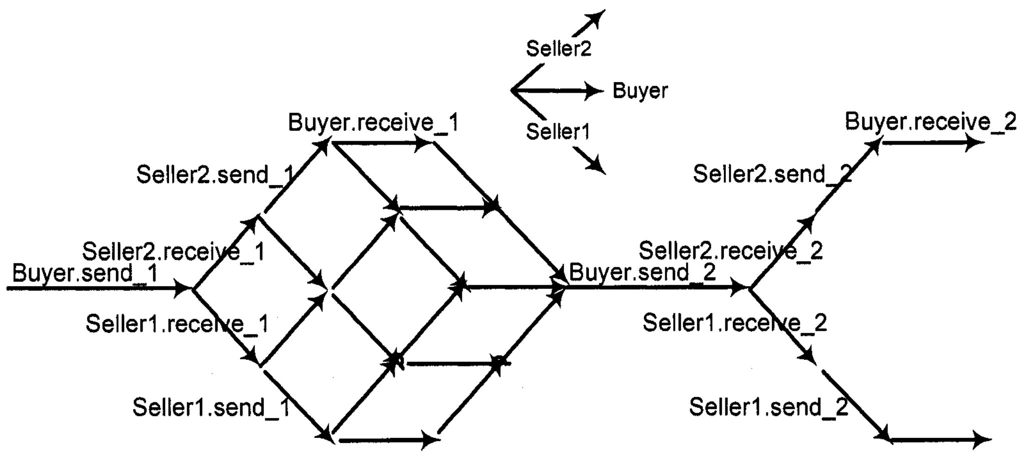


Figure 29: Atom state lattice checked by Java PathFinder

By integrating AspectJ, we can insert the recovery code into the original system. The recovery code is executed when the captured events violate the specified properties. So, the system can take recovery actions to correct the execution at runtime to avoid crashes of the system. In such a way, our tool improves software reliability via recovery, which is highly desirable for the critical systems. On the other hand, the gap between dynamic events for monitoring and static monitor integration based on AspectJ can lead to some limitations. The same problem exists in Hawk, JavaMOP, and our tool. Ideally, for variable update events, these tools should instrument all the updates of involved variables. But, statically locating all such updates requires precise alias analysis. In addition, static instrumentation may cause extra performance penalty in monitoring.

As a future work, the approach used in this thesis can be extended to general Java programs. In this case, the source code of the given Java program should be hand-instrumented to report the values of predicate variables to the monitor which can check the predicates at run time.

Bibliography

[AspectJ] AspectJ crosscutting objects for better modularity. Retrieved May 18, 2008, from The AspectJ project. Web Site: http://www.eclipse.org/aspectj/

[AH05] d'Amorim, M. and Havelund, K. 2005. Event-based runtime verification of java programs. In *Proceedings of the Third international Workshop on Dynamic Analysis* (St. Louis, Missouri, May 17 - 17, 2005). WODA '05. ACM, New York, NY, 1-7. DOI= http://doi.acm.org/10.1145/1083246.1083249

[AUML] The FIPA Agent UML Web Site. Retrieved May 18, 2008, from AUML Web Site. Web Site: http://www.auml.org/

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of the 5th International Conference on Verification, Model Chacking, and Abstract Interpretation (VMCAI'04)*, volume 55(2), 70(4), 89(2) of *LNCS*, Venice, Italy, Jan 2004. Springer.

[CDHR01] J. Corbett, M. Dwyer, J. Hatcli , and Robby. Expressing Checkable Properties of dynamic Systems: the Bandera Speci cation Language. *Technical Report 200104*, Kansas State University, Department of Computing and Information Sciences, 2001.

[CR06] Feng Chen and Grigore Rosu. MOP: Reliable Software Development using Abstract Aspects. *University of Illinois at Urbana-Champaign, Department of*

Computer Science Tec,Technical report UIUCDCS-R-2006-2776, October 2006

[FIPA] Foundation for Intelligent Physical Agents (FIPA). FIPA interaction protocol library specification. *Available from www.fipa.org, 2001.* Document number XC00025D, version 2001/01/29.

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

[JADE] Java Agent DEvelopment Framework. Retrieved May 18, 2008, from Jade - Java Agent DEvelopment Framework. Web Site: http://jade.tilab.com/

[JPF] Java PathFinder. Retrieved May 18, 2008, from Java PathFinder. Web Site: http://javapathfinder.sourceforge.net/

[Kic97] G. Kiczales and et al. Aspect-Oriented Programming. In *ECOOP*, volume 1241. Springer-Verlag, 1997.

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

[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.

[LM07] 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.

[LMG07] Li, H. F., Maghayreh, E. A., and Goswami, D. 2007. Using Atoms to Simplify Distributed Programs Checking. In *Proceedings of the Third IEEE international Symposium on Dependable, Autonomic and Secure Computing* (September 25 - 26, 2007). DASC. IEEE Computer Society, Washington, DC, 75-83. DOI= http://dx.doi.org/10.1109/DASC.2007.30

[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.

[Mu89] T. Murata. Petri Nets: Properties, Analysis and Applications. In: *Proceedings of the IEEE, Vol. 77, No. 4*, pages 541-580. April 1989.

[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.

[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 In- ternational Conference on Computer Aided Verification*, pages 24–437, London, UK, 1993. Springer-Verlag.

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

[Spin] ON-THE-FLY, LTL MODEL CHECKING with SPIN. Retrieved May 18, 2008, from Spin – Formal Verification.

Web Site: http://spinroot.com/spin/whatispin.html