Custom and Model Based Detection of Deficiencies Related to Java Multithreading

Jagmit Singh

A Thesis
in
The Department
of
Electrical and Computer Engineering

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

Your file Votre référence
ISBN: 0-494-10251-9
Our file Notre référence
ISBN: 0-494-10251-9

Abstract

Custom and Model Based Detection of Deficiencies Related to Java Multithreading

Jagmit Singh

Multithreading (MT) has been extensively used for developing Graphical User Interface (GUI) and server side applications, because of the multiple benefits, both concerning program organization and efficiency, offered by it. On the other hand, multithreaded programming is difficult and error prone. It is easy to make a mistake in programming, which could lead to errors and these errors are difficult to detect. Thus automatic bug detection techniques, such as runtime analysis, static analysis, model checking and theorem proving are applied.

Runtime analysis is based on the idea of concluding properties of interest from a single run of the program. We implement and compare two runtime analysis approaches, an ad-hoc custom based approach and model checker based approach to detect common bug patterns. Hyades, a plugin of Eclipse Integrated Development Environment (IDE) supplemented with bytecode instrumentation tool - JTrek is used for trace collection. All the relevant events required for analysis are collected using this trace collection approach. The state-of-art model checker Spin is used for trace verification. The comparison of the approaches is based on experiments we performed on three different Java multithreaded applications, and the results indicate that the custom based approach performs better than the model-based approach.

Acknowledgements

First I would like to express deep appreciation to both my supervisors, Prof. Alexandre Petrenko (Centre de recherche informatique de Montreal, CRIM) and Prof. Otmane Ait Mohammed (Concordia University) for their continuous support, help, and guidance. Another person I would like to give deep thanks is my mentor Dr. Serge Boroday, in fact, without his constant guidance, support, and help this work would have never converged.

I also owe my thanks to faculty and staff members in the department of Electrical and Computer Engineering and Centre de recherche informatique de Montreal, CRIM, especially Dr. A.K Elhakeem, Dr. A. En-Nouarry, Hesham Hallal and Ms. Diane Moffat for their help.

My gratitude always goes to my parents, Harjit Kaur and Upkar Singh, for their endless love, care, and patience.

I would like also to acknowledge the financial support for my master studies, provided mainly by SAP labs, CRIM, and Concordia University.

Table of Contents

List of Tables

List of Figures

List of Acronyms

MT	Multi Threading
IDE	Integrated Development Environment
JVM	Java Virtual Machine
PROMELA	Process Meta Language
JMPAX	Java Multi Path Explorer
JPAX	Java Path Explorer
BCEL	Bytecode Engineering Library
FSM	Finite State Machine
EFSM	Extended Finite State Machine
JAXB	Java Architecture for XML Binding
XML	Extensible Markup Language
LTL	Linear Temporal Logic

Chapter 1. Introduction

Multithreading has been extensively used in web related, server side and graphical user interface and many more applications because of its multiple benefits, both concerning, program organization and efficiency. Some of these benefits include increased performance, multitasking, parallelism, increased responsiveness and others. At the same time, because of the inherent non-determinism and unpredictable scheduling of the threads in multithreaded applications, they are prone to various concurrency related problems such as deadlocks, livelocks, and dataraces. Here we concern analysis of Java multithreaded applications, because Java language is one of the most popular and successful programming languages with extensive multithreading support. Also we consider detection of those bug-patterns or antipatterns which reoccur in various Java multithreaded applications. Manual detection of such antipatterns is difficult, time consuming and inefficient. Thus the automatic bug detection techniques are applied. Broadly these techniques can be divided into static and runtime analysis. Here experiments are performed for antipattern detection using runtime analysis.

The runtime analysis is based on the idea of concluding properties of interest such as correctness, performance, patterns etc, from a single run of the program [25]. A few popular runtime analysis tools are JPAX, JMPAX and JavaMac [27], [36], [49]. The runtime analysis can be broadly divided into offline and online analysis. In the offline analysis (also known as post mortem analysis), a trace obtained from the execution of the target application is analyzed for bug patterns or antipatterns. On the other hand, with online analysis, antipattern analysis is conducted while the target application is executed. The online analysis is more beneficial than offline analysis for antipattern detection (especially those antipatterns related to real-time) in large-scale applications, because the size of the execution trace of large applications is large and thus cumbersome to analyze by offline analysis. The offline analysis is applied to antipattern detection in small scale to medium size applications whose execution traces are manageable enough to analyze. Mostly small and medium scale applications are targeted. Here we propose and implement two offline runtime analysis approaches. We develop offline analysis approaches because of the reduced overhead and simpler analysis.

We implement two runtime analysis approaches namely custom based detection and model checker based detection. Both approaches are based on the idea of offline analysis of the execution trace for antipattern detection. In the custom detection the execution trace is verified for antipatterns using detectors coded in Java. In the model checker based approach, a formal model in input language of SPIN model checker is built from the execution trace and then antipatterns are detected using model checking. Both approaches use the same instrumentation and trace collection technique, but the difference between them is in the antipattern detection method.

Both approaches rely on the Eclipse platform for the trace collection. Eclipse platform is an IDE (Integrated Development Environment) built on a mechanism for discovered, integrated, and executable modules called plug-ins [34]. Hyades, an Eclipse plugin for testing, tracing and monitoring software systems is used in our approaches for trace collection.

Finally a comparison is made between these two approaches (namely custom based detection and model checker based detection) in terms of the quality of analysis, resource consumption, time usage, complexity, ease of usage and scalability.

1.1 Motivation

Multithreading has been extensively and widely used to perform complex computations and operations in multiprocessor and multiplatform networking environment, such as Internet. In particular, it is essentially used in developing GUI and server side applications. Also most of the recent programming languages provide support for multithreading.

Developers and designers are more adapted to thinking and programming in a sequential manner. As a result they commit a lot of mistakes when programming multithreaded applications, which are concurrent in nature rather than sequential. In other words, multithreaded errors are difficult to avoid; thus, for the detection of these errors, automatic bug detection techniques are usually applied. Runtime analysis is one of the automatic bug detection techniques. Runtime analysis has been successfully applied in the detection of "difficult to detect errors" in many projects [49], [27], [36]. The errors in multithreaded applications often manifest only on certain executions, which makes their detection more difficult. However, sophisticated analysis methods are able to predict problems that do not appear on the observed executions [50]. Unfortunately, few of these methods are based on solid formal basis; lack of the formal basis often results in numerous false alarms. Thus there is a need to develop methods for predictive trace analysis, based on formal verification techniques, such as model checking.

The runtime analysis of Java multithreaded (MT) programs written in Eclipse environment was part of the joint work between SAP labs and CRIM.

1.2 Objectives

The objectives of this work are as follows:

1. To develop a lightweight instrumentation approach to extract the information necessary for detecting antipatterns, without causing significant overhead to the target program.
2. To develop runtime analysis approaches for multithreaded antipattern detection in Java multithreaded applications, namely custom based detection (a semiformal approach) and model checker based detection (a formal approach), and to implement them.
3. To make a detailed comparison between these two runtime analysis approaches by performing experiments.

1.3 Related Work

The runtime analysis has gained increased importance in recent years because of the increased use of critical and business related software. The continuous and smooth functioning of software in these critical applications is very important. Even a slight interruption can lead to a huge loss of life or property. These critical applications are used in space missions, nuclear power plants and medical related devices and instruments. Considering the criticality of these applications various bug detection testing techniques are applied. Runtime analysis is one such popular bug detection technique.

Runtime analysis has been extensively researched in industry and academics, because of its increased importance. It has been studied and used in organizations such as NASA Ames Research Centre, University of Pennsilvania, University of Urbana

Champaign, Digital Corporation and Compaq. These studies have led to many analysis tools such as JMPAX, JPAX, JavaMac, Verisoft, and JProbe Threadalyzer [36] [27] [49] [56] [41]. The tools JMPAX and JPAX are developed to detect bugs in the software used in NASA space missions. JMPAX (University of Urbana Champaign) has the ability to predict the errors in all possible executions (of a multithreaded trace) only by observing the single run of the program. The possible executions are those executions which do not violate the observed casual dependencies on state updates events. This ability to perform comprehensive analysis is achieved with the help of the vectors clocks inserted into the bytecode to capture the relevant causality of the events and thus helps to predict the potential errors from the single run of the program.

The JPAX, a tool for monitoring the execution of Java programs, is developed at NASA Ames research centre [27]. The tool facilitates the automated instrumentation of Java bytecode, which when executed emits relevant events of interest to the observer. The observer performs the verification of the properties of interest based on the information extracted from these events. JPAX uses a variant of Eraser [50] algorithm to predict data races by analyzing a single execution of the monitored program [27].

The JavaMac - a research prototype tool developed at University of Pennsilvania is a version of the existing MAC (Monitoring and Checking) framework applied for Java programs. The salient aspect of the JavaMac architecture is the use of formal requirement specification to check run-time executions of the Java programs. The formal specification is specified as Primitive Event Definition Language (PEDL) and the Meta Event Definition Language (MEDL) [49].

The JProbe Threadalyzer, developed by Quest software, can identify deadlocks (even potential deadlocks), data races, and stalled threads. JProbe is one of the most successful commercial tools for thread analysis. JProbe Threadalyzer uses two analyzers to detect potential deadlock conditions. The first analyzer detects deviations in the order of lock acquisition, which often indicate potential deadlocks. Lock order violations can occur when concurrent threads need to hold two locks at the same time. The second analyzer detects the so-called "hold-while-waiting" condition that occurs when a thread holds a lock while waiting for notification from another thread [41]. Also JProbe Threadalyzer uses two different methods to predict possible data races. The first method is based on the happen-before relation [41] and the second method is based on lock cover analysis. The lock cover analyzer watches all access to shared variables, and tracks the lock cover - the set of locks held by all threads that access a shared variable.

The multithreaded problems are explained in detail in Chapter 2, and relevant tools and techniques for the analysis of Java multithreaded applications are reviewed in Chapter 3.

1.4 Research Contribution

The main research contributions of this thesis can be stated as follows:

- **Lightweight Instrumentation of Java Applications**

The Java tracing is based on Hyades, which is probably one of the most promising platforms for development of various quality assurance applications. Unfortunately, the non-intrusive tracer of Hyades is not able to collect all the events required for the detections of multithreaded problems. Thus, the bytecode is instrumented to collect such events.

Most of the instrumentation tools cause overhead to the target application, which can alter real time properties. The lightweight instrumentation approach developed in this thesis complements an existing, tracing tool (Hyades) for collecting all the relevant events of interest from the target application without causing significant overhead to the target application. Our instrumentation is lightweight because only empty methods are added.

- **Multithreaded Java Trace Analysis Based on Model Checking**

A model checking verification technology is applied for detection of multithreaded antipatterns in the traces of Java programs. While the application of existing model checkers for trace analysis is not new, our model of trace reflects thread concurrency and enables predictive analysis, which could predict possible faults or problems.

- **Custom Multithreaded Java Trace Analysis**

It is shown that certain simple multithreading related antipatterns do not need sophisticated detection techniques, and could be detected more efficiently with custom detectors. The custom analyzers for the execution traces of Java multithreaded applications, using same Java-XML technology (JAXB) are built.

- **Comparative Analysis**

A comparative analysis of both tools (custom analyzer and model based detector) is performed. Our experiments reveal that for medium size applications, the custom analysis tool is faster (almost 3 times) than model based detection. Although the analysis time for both tools is of the same order. The model checking time is negligible compared to the time consumed in model building, compilation, linking and other auxiliary steps.

1.5 Organization of the Thesis

The thesis is organized as follows. Java multithreading problems are explained in detail in Chapter 2. Chapter 3 describes relevant tools and techniques for the analysis of Java multithreaded applications. In Chapter 4 we suggest an instrumentation method build on existing approaches. Chapter 5 describes the custom based detection approach and developed experimental prototype tool. Chapter 6 describes in detail the model checker based detection approach, while in Chapter 7 a detailed comparison of two approaches, namely custom based detection with model checker based detection is provided. Finally Chapter 8 contains conclusion and a description of future work.

Chapter 2. Java Multithreading

2.1 Introduction to Java Multithreading

The Java multithreading can be defined as a way of building Java based applications with multiple threads. In a multithreaded application each thread is a different stream of control that can execute its instructions independently, which allows a multithreaded application to perform numerous tasks concurrently. For example, the first thread can run the GUI, while the second thread performs some I/0 and third performs some calculation [43]. Thus, multithreading enables concurrent execution of several threads within the same application. Multithreading is a convenient way to decompose large application into relatively independent smaller tasks and thus increases the overall efficiency [43]. Multithreading is almost a necessity for all but the most trivial programs.

However analysis of multithreaded programs can be a challenging task, because of the complications involved in characterizing the effect of the interactions between threads. The solution therefore is developing efficient abstractions and analysis techniques that capture the effect of each thread's actions on other parallel threads.

2.2 Benefits of Multithreading

In spite of all the challenges of multithreading, it has many benefits, some of which are listed here. Because of the benefits offered by multithreading, it has been supported in most of the recent programming languages. It has been extensively used in Java programming for developing server side, user interface and web related applications.

- **Performance gains from multiprocessing hardware/parallelism**

  Computers with more than one processor (multiprocessor computing) offer the potential for enormous application speedup and thus higher performance gains [43]. Multithreading (MT) is an efficient way to exploit the parallelism of the hardware. Different threads can run on different processors simultaneously with no special input from the user.

- **Increased application throughput**

  In a single threaded program, when a request for service is made, it must wait till the service is complete, which makes CPU idle [43]. In such a situation the multithreaded program can utilize the CPU idle time by utilizing second thread to service another request. For example, the second thread can handle I/O operation. Thus multithreading helps in effective utilization of time and hence increases the overall application throughput.

- **Increased application responsiveness**

  In the case of single threaded application, a single thread performs most of the operation. If one part of that single thread operation is stopped then the whole operation administered by that thread is stopped [43]. Such a blocking situation decreases the user responsiveness. To prevent such a blocking situation, multithreaded program proves useful, that is even if one thread is stopped other threads can still continue their operation.

- **Replacing process-to-process communication**

  In an application where multiple processes are used for communication purpose, multiple threads can replace those processes to accomplish the same task. In the traditional multiprocessor environment the communication is done through sockets, pipes etc, and the same communication be performed more efficiently by multiple threads through shared variables.

2.3 Terms Related to Multithreading

Multithreading

A form of parallelism where multiple execution threads run concurrently and communicate via shared memory.

Locks

Multithreaded application use locks to synchronize and communicate their behaviour to one another. The locks around shared variable allow the threads to easily synchronize and communicate. The thread that holds the lock on an object knows that and will not allow other threads to access this object. Even if the thread holding the lock is pre-empted another threads cannot access the object, until the original thread wakes up, finishes its work and releases the lock. Thread that acquires the lock in use goes to sleep until the thread holding the lock releases it, when the lock is released the sleeping threads wake up and move to the ready-to-run queue.

Monitor

As defined by C. A. R. Hoare in 1974, "a *monitor* is a concurrency construct that encapsulates data and functionality for allocating and releasing shared resources (such as network connections, memory buffers, printers and so on)" [31]. To achieve resource allocation or release, a thread calls a *monitor entry* (a special function that indicates as an entry point into a monitor). If there is no other thread executing code within the monitor, the calling thread is allowed to enter the monitor and execute the monitor entry's code. But if a thread is already inside of the monitor, the monitor makes the calling thread wait outside of the monitor until the other thread leaves the monitor. The monitor then allows the waiting thread to enter. Because synchronization is guaranteed, problems such as data being lost or scrambled are avoided.

2.4 Java Implementation of Multithreading

Threads functionality is implemented in Java using the class `java.lang.Threads` and there are two approaches to create a new thread of execution. One approach is to declare a subclass of Thread class. This sub class should override the run method of Thread class. An instance of the subclass can then be allocated and started. Another approach is that another thread can be created, which implements the `runnable` interface. That class then implements the run method (sample of this Java implementation is shown in Figure 2.1). One can then create a thread object with this `runnable` as the argument and call `start ()` on the thread object.

```
Public MyRunnable implements Runnable {
Public void run ( ) {
doWork ( );
}
}
Runnable r = new MyRunnable ()
Thread t = new Thread (r);
t.start ( );
```

Figure 2.1: Code Sample to Implement a Runnable in a Thread

2.4.1 Thread Synchronization in Java

Here we explain how the thread synchronization is implemented in Java using locks. In Java programming, each object has a lock; a thread can acquire the lock using the `synchronized` keyword [7]. With this keyword, certain method on blocks of code could be declared as synchronized on an object. For example, let *o* be an object, when entering a section that is synchronized on *o*, the current thread tries to acquire the lock ("enters the monitor") for *o*. If lock is granted, no other thread could access a section or a method, synchronized on *o*, unless the lock owning thread quit the synchronized section, or invokes `o.wait`. By calling `o.wait` the current thread temporarily releases the locks it holds on *o* and thread is added to the wait set of *o*. It is suspended until another thread calls `o.notify`, or when an optional specified amount of time has elapsed. `Wait` can be useful if the current thread is waiting for a certain condition that can only be met by another thread that needs access to the monitor. When the waiting thread resumes execution, the locks are automatically reacquired. By calling `o.notifyAll` all threads in the wait set of *o* wake up. `NotifyAll` is used when it cannot be guaranteed that each thread in the wait set of *o* can continue execution [7].

Yet another method of Java, `Thread.join ()`, could be used for different type of synchronization. Whenever one thread calls the `join ()` method of a second thread, it ensures that all the events of the callee thread have been executed before the events following the `join ()` call in the caller thread. In many cases one could use `join ()` instead of the `wait ()/ notify ()` pairs.

2.5 Multithreaded Problems

Multithreading can cause various problems, such as incorrect application behaviour or deadlocked conditions. Here we will discuss the common multithreaded problems

2.5.1 Deadlock

The misuse of locks could cause many problems, mostly synchronization related and deadlock is one of most severe among them.

There are several definitions for deadlock condition. It can be summarized as; when a deadlock occurs in a program, the whole program or a part of does not progress anymore [22]. In a multithreaded Java application, deadlock occurs when each of two or multiple threads is waiting for availability of a lock that will not become available because it is held by some of these threads [22]. We illustrate the deadlock conditions with a small example [5].

```
 Thread 1
Synchronized (A) {
Synchronized (B) { }
}
Thread 2
Synchronized (B) {
Synchronized (C) {}
}
Thread 3
Synchronized (C) {
Synchronized (A) {}
}
```

Figure 2.2: A Simple Deadlock Example

As shown above in Figure 2.2, if all three threads hold one lock each, none of them can continue, because the second lock they need is already taken [5]. Such a cyclic acquiring will create a deadlock condition [5].

2.5.2 Race conditions

Races occur when several threads access the same resource simultaneously without proper coordination [50]. As a result the program might end up producing output far different from the desired one. For example, a race condition occurs when two concurrent threads access a shared variable and when, a least one access is write, and the threads use no mechanism to prevent the simultaneous access.

The following conditions for two events could lead to data race condition in an object oriented multithreaded program [11]:

- The two events access the same memory location (same field/element in a class/object).
- The two events are executed by different thread objects.
- The two events are not guarded by same synchronization object.
- There is no execution ordering enforced between the two events by thread creation or termination.

2.5.3 Livelocks

A livelock occurs when one thread takes control (e.g., locks an object of a shared resource) and enters an endless cycle. In other words, a livelock is a condition in which two or more threads continuously change their state in response to change in the other thread(s) without doing any useful work [5].

A livelock is similar to a deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.

Due to the similarity between a deadlock and a livelock the task of identifying and detecting livelocks in a program becomes complex as well. An example of a livelock is the famous dining philosopher problem [16]. Consider, in a dining philosophers program, the scenario where all the philosophers pick up the fork on their right at the same time. Then, they all put the fork back simultaneously. By repeating this endlessly the program enters in a livelock where all the philosophers are active but none is eating. The justification is that all the philosophers were trying to avoid a deadlock (when they all take the fork to the right and do not release it). However, they ended up with a livelock.

2.5.4 Efficiency and Quality Problems

The main factor affecting the efficiency of MT applications is synchronization. As much as it is needed in MT programs, synchronization causes a significant overhead that usually accounts to 5-10% of the total execution time in some cases [1]. This results from the fact that managing synchronization in Java MT applications requires the Java Virtual Machine (JVM) to perform some internal tasks (writing any modified memory locations back to main memory) that could impair the efficiency of the application.

Another aspect that affects the efficiency of MT applications is the use of `notify ()` method instead of the `notifyAll ()` method (whenever it is possible). The `notifyAll ()` method is more expensive.

2.6 Multithreaded Antipatterns

The concept of patterns has been widely used in software design and development. A pattern is "a consistent, characteristic form, style, or method" [17]. General characteristics of patterns are [57]:

- When developers write a code, they usually follow some pattern. The pattern followed is derived from their previous experience.
- Some developers follow the same pattern.
- Some patterns could lead to success and some bad patterns could lead to failure.
- Usually patterns are localized within a small amount of time and space.
- Patterns instance are recognizable.

Recently in the software verification and validation domain, the concept of predefined error description (known as antipatterns or bug patterns) has been introduced to help reduce the effort spent in verification or debugging the software. *Design Patterns* are often used in software design and development particularly in the object-oriented design and development; they offer elegant solutions to common problems in software design. Their usage helps in saving the software development and maintenance cost.

Two types of antipatterns identified so far are design antipatterns and bug patterns, as described below:

2.6.1 Design Antipatterns

Common design patterns which have failed again and again in the software design and development. From this viewpoint, a design antipattern is simply a solution to a problem that does not work correctly, and it can be seen as just another design pattern [9]. The notion of an antipatterns can be stated as "something that looks like a good idea, but which backfires badly when applied" [3].

2.6.2 Errors or Bug Patterns

A bug pattern is a pattern, which leads to errors/faults in software applications. In the multithreaded context we view that bug patterns could lead to MT problems, e.g. deadlocks, livelocks, and race conditions. A bug which frequently repeats themselves in the various Java programs can be classified as a bug pattern or bug antipattern.

2.7 Antipattern Library

At the least 38 different antipatterns (bug patterns) have been identified, which relate to concurrency, synchronization, and other common multithreaded Java problems [22]. These MT antipatterns have been catalogued in library and classified in their corresponding groups [22].

2.7.1 Classification of Antipatterns

Antipatterns identified can be classified into the following categories [22]. This classification is based on the MT problems (listed below) the antipatterns address.

1. Deadlocks
2. Livelocks
3. Race Conditions
4. Efficiency Problems
5. Quality and Style Problems
6. Problems with unpredictable consequences.

2.7.2 Antipattern Template

To archive the antipatterns in a library, the template as shown in Table 2.1 is defined [22]. The template was proposed to represent antipatterns of problematic situations in the MT Java code. Each template provides information about a particular antipattern including the definition (name, description, and category), an example of occurrence (when possible), the re-factoring solution, and potential conflicts of applying the solution, possible detection technique, and some comments.

Table 2.1: Antipattern Template

Name	A concise definition of the problem.
Description	The situations in which this problem could appear. The effects it has on the code and the application.
Category	Deadlock, Livelock, Race Condition, Efficiency problem, Quality and Style Problem, Problem with unpredictable consequences.
Example	If available, sample code where the problem is illustrated.
Detection	How to detect the problem in the Java code. A high level description of the proposed algorithm to be used in the detection process.
Re-Factoring	Solution: How to solve the problem once detected in the program. Conflicts: Sometimes solving one problem of a certain class can cause another problem of a different class. For example, Blob threads and over synchronization.
Comments	The source of this pattern. Any comments that could be helpful in the detection or re-factoring.

The information provided in the template helps both the developers of MT applications and professionals building tools to detect the antipatterns in MT applications. The template is easy to be understood by programmers. It contains useful information for using antipatterns in programming practice, as most of the fields are directly related to programming practice. It can be used to teach programmers how to avoid writing buggy programs. Next we give example of an antipattern “premature `join ()` call” [22] whose detection we will discuss later.

Table 2.2: Antipattern- Premature Join () Call

Name	Premature `join()` call
Description	.A call to the `join ()` method of a thread is premature if this thread has not been started at the time of the call. In Java such calls are simply ignored, but their presence is alarming because they may indicate a fault in the program logic or non-optimal code.
Category	Quality and style problem.
Detection	A data flow analysis is needed, but dynamic analysis could be more efficient. Detectable by Flavers.
Re-Factoring	Solution: Rethink the logic of the program to make sure that the `join ()` method of a thread is not called until it is already started. Conflicts: None.
Comments	Source: http://cis.poly.edu/gnaumovi/papers/flavers-java.pdf

2.7.3 Dynamic Java Antipatterns and Relevant Events

As the focus of this research is mostly on dynamic analysis, only dynamic MT antipatterns are listed. Dynamic analysis approach is detailed later in Section 3.2.

Antipattern: Locked but not used object

Description: A thread locks but never uses an object.

Detection: In each synchronized block, check if the synchronized object is being actually used.

Events required for its detection:

- monitor enter/exit,
- object creation.

Antipattern: Overthreading

Description: It is when a "large" number of threads are defined and created.

Detection: This antipattern can be detected by computing the number of the threads creation and destruction events that are logged. The threshold of simultaneously running threads is user defined, though some sources [22] suggest that it should be set to the value of three.

Events required for detection:

- thread creation,
- thread blocking.

Antipattern: Blob thread

Description: A thread that takes the whole or a large part of the activity of the system.

Detection: This antipattern can be detected by calculating the ratios (duration of the program)/ (duration of the thread) for all the threads and then comparing the ratio for the blob thread with other threads. In this way, we can calculate the duration of the global program execution and those of the threads and thus detect the presence of the blob thread.

Events required for detection:

- method entry,

- monitor enter.

Antipattern: Complex computation within an AWI/Swing thread

Description: The listeners in a Java's Abstract Windows Interface (AWI) thread:

- are too long (temporary irresponsive interface),
- never end, e.g., due to a cycle with a continuous true condition (irresponsiveness),
- share many objects with the main thread (may affect responsiveness).

Events required for detection:

- method entry/exit,
- object access.

Antipattern: Misuse of `notifyAll ()`

Description: The `notify()` method is more efficient than the `notifyall()` method, especially when not too many objects are shared between threads. In addition, there is a possible performance problem with the overuse of `notifyall()` known as "Thundering Herd". When `notifyall()` is called, all threads (that are waiting for the same object-lock), will receive a signal but only one gets the lock. On the other hand, when there are just two threads operating on this object, the use of `notifyall()` is not needed [22].

Detection: Collect all objects used in each thread and then search for objects that are used by only two threads.

Events required for the detection:

- `notifyall()` calls.

Antipattern: Unnecessary notification

Description: Notification issued when no threads are waiting. For example for each thread that may call `notify()` or `notifyall()` methods, check if another thread could call `wait()` method on the same object.

Events required for detection:

- method call (calls of `notify/wait`).

Antipattern: Waiting Forever

Description: Thread executes the `wait` for the lock object of monitor, but is never notified and thus, never resumes its execution.

Detection: A pure dynamic approximation would be to detect long waits (waits that exceed a user specified long period of time, e.g., one sec). Detection of such a property from a trace is straightforward.

Events required for detection:

- method entry/exit.

Antipattern/Problem: High Level Data Race

Description: "High-level data races occur when different activities, executed in parallel, do access shared resources, but with different atomicity views" [6].

Detection: It can be performed using a run-time analysis algorithm, which searches inconsistencies in the views that different activities have on shared resources. The algorithm works by analyzing a single randomly chosen execution trace for operations that take and release locks and for operations that access shared resources [6]. From this information it can be concluded whether all the activities have consistent views. Inconsistent views typically arise if at least one activity "does it right". This antipattern covers a wide range of bugs and is sometimes referred to as a special subclass of datarace problem.

Events required for detection:

- monitor enter/exit,
- field variable access.

Antipattern: Dead Interaction

Description: Call of the thread that has already terminated.

Events required for detection:

- method enter/exit,
- exceptions throw.

Antipattern: `Wait` Stall

Description: The thread should not `wait` (after calling the wait method) for more than user specified amount of time

Events required for detection:

- method entry/exit,
- thread start/end.

Antipattern: Premature `join()` Call

Description: A call to `join()` is premature if this thread has not yet started at the time of the call [22].

Events required for detection:

- method entry/exit,
- thread start/end.

Antipattern: `Join()` with "Immortal" thread

Description: A call to `join()` with a thread that never ends (e.g. daemon or main thread). Since thread immortality is not always detectable dynamically, the antipattern should be approximated.

Events required for detection:

- method entry/exit.

Antipattern: Double call of the `start ()` method of a thread

Description: The `start ()` method is not supposed to be called more than once for the same thread [22].

Events required for detection:

- method entry,
- thread start/end,
- monitor enter/exit.

Antipattern/Problem: Divergence

Description: A divergence (in the form of livelock or stall) occurs when a process does not attempt to communicate with the rest of the system for more than a given (user-specified) amount of time [56].

Events required for detection:

- object access (with timestamps).

Antipattern/Problem: Resource Deadlock

Description: A resource deadlock can occur when two or more threads block each other in a cycle while trying to access synchronization locks (held by other threads) needed to continue their activities [24].

Events required for detection:

- object access (with timestamps),
- monitor exit/enter or deadlock contention.

Chapter 3. MT Java Analysis Approaches

3.1 Introduction

In this chapter, techniques and tools for antipattern detection are described. These techniques and tools can be broadly classified by three types namely; dynamic checkers, static checkers and formal techniques such as model checking and theorem proving.

Dynamic analysis requires the execution of the program and then analysis of the execution trace for the property verification. In static analysis, one does not run the program, but performs analysis of the code (either source code or bytecode). The analysis is normally independent of the input order or thread scheduling since the code is analyzed without execution.

3.2 Runtime Analysis

Runtime Analysis – is based on the idea of concluding properties of interest from the single run of a program [25]. "The purpose of runtime analysis is to cover an area not covered by formal verification and testing" [42]. The purpose of formal verification and testing is to assure the correctness of all possible executions of a target application, whereas runtime analysis assures correctness of the current execution of a program. We tried to detect the classified antipatterns with both static analysis and runtime analysis. Most of the antipatterns identified could be detected by both static and runtime analysis except for few. But runtime analysis gives significant edge over static analysis in detection of certain antipatterns. In detection of these antipatterns, runtime analysis could detect false positives generated by static analysis.

3.2.1 Benefits of Runtime Analysis

The advantages of runtime analysis:

1. The possibility of detecting errors, which have actually happened (on specific data, platform, and JVM).
2. The ability to detect errors which are impossible or too difficult to detect statically.
3. Source code is not required.

3.2.2 Challenges of Runtime Analysis

Implementing the runtime analysis approach faces a number of challenges, among which are:

1. Observation of a program behavior requires special efforts (instrumentation).
2. Instrumentation at points of observation may cause side effects on behavior and timing characteristics.
3. The runtime analysis can assure correctness of the current execution taken during the observation period and not much can be said about other possible executions. Thus it cannot prove the system correct as a whole.
4. Similarly, the results are valid only for a thread scheduling performed while the program was executed. Thus the results are not consistent.
5. Errors are often difficult to reproduce.

3.2.3 Tools for Runtime Analysis

Below we provide summary of a few most popular commercial and research runtime analysis tools.

3.2.3.1 Tool: JavaMac

JavaMac was developed at Department of Computer Science, University of Pennsylvania; main authors are Moonjoo Kim, S. Kannan and M Viswanathan. JavaMac is a prototype implementation of monitoring and checking (MAC) architecture for Java programs [49].

A salient aspect of the JavaMac is the use of a formal requirement specification to check run-time execution of the target program. For specifying formal requirement the Primitive Event Definition Language (PEDL) and the Meta Event Definition Language (MEDL) are used. Its architecture is modular, separates monitoring implementation-dependent low-level behaviour and checking high-level behaviour with regard to formal requirement specification. JavaMac's architecture can be divided in three main modules namely Information Extraction, Monitor/Checker and Formal Requirement Specification. In addition architecture instruments the target program and analyzes the execution the execution of the target program automatically based on the formal requirement specifications.

3.2.3.2 Tool: Java PathExplorer

Java PathExplorer (JPaX) was developed at the Automated Software Engineering Group, NASA Ames Research Center; main authors are Klaus Havelund and Grigore Rosu [27].

Java PathExplorer is a tool for monitoring the temporal behavior and finding concurrency related errors (such as deadlocks and dataraces) detection particularly in multithreaded applications. The tool performs script driven instrumentation of the program's bytecode, which emits events to an observer during its execution. The observer checks the emitted events against user specified high-level requirement specifications, for example, temporal logic formulae, and against lower level error detection procedures, usually concurrency related, such as deadlock and data race algorithms.

3.2.3.3 Tool: Java MultiPathExplorer

Java MultiPathExplorer (JMPaX) analyzes a multithreaded program against the safety properties expressed using temporal logic [52]. The tool is developed within Formal Systems Laboratory at the University of Illinois at Urbana-Champaign; main authors are Koushik Sen, Grigore Rosu and Gul Agha. In fact, the limitations of JPaX motivated this development.

JMPaX is a prototype tool for runtime safety analysis of multithreaded programs. It can predict violations of safety properties expressed in temporal logic from executions of multithreaded programs [52].

The user of JMPaX specifies the safety properties of interest, using a past time temporal logic, regarding the global state of the multithreaded program (assumed to be in compiled form). Then, JMPaX calls an instrumentation script which automatically instruments the executable multithreaded program to emit relevant state update events to an external observer, and finally runs the program on any JVM and analyzes the safety violation messages reported by the observer [52]. An appealing aspect of this approach is that a single execution, or interleaving, of a multithreaded program is observed, a comprehensive analysis of all possible executions is performed; a possible execution is any execution which does not violate the observed causal dependency partial order on state update events. The tool JMPaX built on this approach has the ability to predict safety violation errors in multithreaded programs by observing successful executions.

3.3 Static Analysis

Static Analysis detects runtime errors and unpredictable code constructs without executing code. In other words, it is based on the analysis of code (source code or bytecode) and is normally independent of input order or thread scheduling since the code is analyzed without execution. Static analysis tools of various types, including formal analysis tools, are being developed, which can detect faults in the multi-threaded application [47] [22]. Common static analysis techniques include data flow analysis, control flow analysis, type checking as performed by modern programming language compilers, abstract interpretation and type and effects analysis.

3.3.1 Benefit of Static Analysis

Verification can begin earlier in the Software Life Cycle resulting in early detection of problems and thus reduction in development cost [47].

3.3.2 Challenges of Static Analysis

The biggest challenge for Static Analysis is generation of false positives sometimes due to over approximation [47]. However, in some cases the number of false positives may be very large and subsequent errors can be the result of an initial or upstream error. Thus correcting these errors can eliminate some false positives.

For example, assume that the analysis only tracks the sign of some integer variables. If positive and negative values are added, the algorithm cannot tell the sign of the result and will consider both alternatives as error on the safe side. One of them may lead to error that corresponds to no actual feasible execution of the real program [47].

3.3.3 Tool for Static Analysis

Here a brief description is given for JLint, a typical static analysis tool.

3.3.3.1 Tool: Jlint

Jlint checks Java class files for loops in the lock dependency graph. This graph includes both static and runtime methods. It also makes sure the programs follow certain consistency rules when using the wait method in Java. Race conditions are found by building the transitive closure of methods, which can be executed concurrently, and the methods they call. Then, all field accessed by such methods which fulfill certain conditions are reported as possible race conditions in data access. Jlint is rather conservative at reporting errors, since it does not allow annotations, which could eliminate false positives [5].

3.4 Model Checking

Model checking is a formal technique for verifying finite state concurrent systems against required specification of the system. The tasks involved in model checking are as follows [51]:

1. A formal model of the system is build in terms of a state transition system. The state transition system is a tuple $M = (S, S0, R, L)$ where:
    - S is the finite set of states.
    - $S0$ a subset of S is the set of initial states from which system can start its execution.
    - $R \in S \times S$ is a total relation, describing the possible transitions from one state to another state of the system, and;
    - L: $S \rightarrow P(AP)$ is a labelling function, stating the atomic propositions (AP) that hold in a given state, where $P(AP)$ is the powerset of the set AP.
2. The properties that the model must satisfy are stated as a specification. The specification is usually given as input in some logical formalism. The commonly used formalisms are temporal logics.
3. After expressing the model and the formal specification, the verification task involves checking the conformance of the model to the given specification. In case of a negative result, a counter-example is generated. This process is completely automatic.

In model checking, all possible computations of the systems are analyzed. So the verification is rigorous and complete. Model checking discovers a bug if it is present in the system. Theoretically, model checking is very efficient. However, in practice model checking may require the entire state space of the system to be stored before bug can be detected, this result in *state space explosion* problem. In sequential programs variables may have many possible values leading to a large number of possible states. If the total number of possible states of the system is large, model checking becomes intractable which makes this technique not scalable.

3.4.1 Spin Model Checker

Spin is a widely used model-checker that supports the formal verification of distributed systems. This model checker was developed at Bell Laboratories in the Formal Methods and Verification group.

Spin has also been used to detect logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signalling protocols, etc. The tool checks the logical consistency of a specification. It then reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

3.4.1.1 Language of Spin

PROMELA is input language for Spin Model-checker. PROMELA (Process Meta Language) is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and Hoare's language CSP. It contains the primitives for specifying asynchronous (buffered) message passing via channels, with arbitrary numbers of message parameters. It also allows for the specification of synchronous message passing systems (rendezvous). Mixed systems, using both synchronous and asynchronous communications, are also supported [32].

The language can model dynamically expanding and shrinking systems: new processes and message channels can be created and deleted on the fly. Message channel identifiers can be passed from one process to another in messages.

Correctness properties can be specified as standard system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of next-time free LTL, or indirectly as Buchi Automata (expressed in PROMELA syntax as Never Claims).

3.4.1.2 Features of Spin

Spin can be used in three basic modes [47]:

- As a simulator, allowing for rapid prototyping with random, guided, or interactive simulations.
- As an exhaustive state space analyzer, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search).
- As a bit-state space analyzer that can validate even very large protocol systems with maximal coverage of the state space (an approximation technique).

3.4.1.3 Documentation

The tool is well documented with tutorial, user manual, several research papers and books. The most recent and comprehensive reference is book of Gerard J. Holzmann, The Spin Model Checker "Primer and Reference Manual".

3.4.2 Benefits of Model Checking

The benefits of model checking are [47]:

1. It is a fast, automated method for exploring all relevant execution paths of non-deterministic systems. This is very important because it is virtually impossible for humans to conceive every test scenario required to verify a non-deterministic system in a plausible time frame for software development.
2. Model checker can possibly backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between tests required in traditional scenario based testing.
3. Detects problems in the early stages of software development lifecycle; thereby greatly reducing overall development costs.

3.4.3 Challenges of Model Checking

There are two challenges associated with model checking [47]:

- Models must be translated into model checking language like PROMELA (for Spin).
- State space explosion – Because of the complexity of software components interaction and because of wide range of data structures values, it is common for a model checker to run out of memory before exploring the entire state space.
- Very few model checkers provide a direct support to verification of Java applications.

3.4.4 Model Checking Based Tools for Java

Here we provide summary of few popular Java analysis tools based on model checking

3.4.4.1 Tool: Bandera

Bandera has been developed by SANTOS group at Kansas State University. Bandera takes as input Java source code and generates a program model in the input language of one of several existing verification tool; Bandera also maps verifier output back to the original source code. It enables the automatic extraction of safe, compact finite-state models from the program source code [13]. Bandera tries to bridge the gap between software source code and an abstract representation of it.

3.4.4.2 Tool: JPF

The *Java PathFinder* has been developed at the Automated Software Engineering (ASE) department at NASA. The Java PathFinder, JPF, is a translator from a subset of Java 1.0 to PROMELA, the programming language of the Spin model checker. The purpose of JPF is to establish a framework for verification and debugging of Java programs based on model checking. The system is especially suited for analyzing multi-threaded Java applications, where normal testing usually falls short [37]. The system can find deadlocks and violations of Boolean assertions stated by the programmer in a special assertion language.

3.5 Theorem Proving

Theorem proving – It is a rigorous formal technique, based on logical induction, in which system requirements are translated into complex mathematical equations and solved with verification experts. Solving these equations proves that the system is accurate [47].

3.5.1 Benefit of Theorem Proving

It can use the full power of mathematical logic to analyze and prove properties of any design.

3.5.2 Challenge of Theorem Proving

Theorem proving requires significant efforts and expertise, and is mostly appropriate for analysis of small-scale systems.

3.5.3 Theorem Proving Based Tool

3.5.3.1 Tool: ESC/Java (Extended Static Checker for Java)

ESC/Java is a programming tool for finding errors in Java programs. ESC/Java detects at compile time, common programming errors that are not detected until runtime and sometimes not even then; for example null dereference errors, array bounds errors, type cast errors, deadlocks, and race conditions [18].

ECS/Java uses program verification technology (automatic theorem prover), but feels to a programmer more like a type checker. It tries to detect certain kind of errors only, but does not prove the program's correctness and also this technique is more automatic than full program verification. ESC/Java performs *modular checking*, that is, ESC/Java verifies each class separately. This means that ESC/Java can be applied to code that calls libraries even if the code for the libraries is not available. It also means that ESC/Java can be applied to library code whose clients or subclasses have not yet been written.

Chapter 4. Java Trace Collection

4.1 Introduction

The runtime analysis requires the execution of a program. Then the analysis (either online or offline) of the execution trace for antipattern detection. An execution trace contains relevant events, which are analyzed for the users' specified MT antipatterns. In order to extract these relevant events, the target program is instrumented. Instrumentation may be performed either at bytecode, source code, or JVM level. When the instrumented program is executed, the relevant events are emitted and collected in an execution trace (in XML format).

Information provided by these events depends on the instrumentation tool used, the location of the instrumented code, level of instrumentation, and depth of the instrumentation etc. Thus quality of runtime analysis depends on the instrumentation tool used and the approach followed.

For example, in JMPaX, a runtime analysis tool has the ability to predict safety violation errors in multithreaded programs by observing successful executions. The ability to predict safety violation errors was obtained through the use of "smart" observer and this "smart" observation was possible due to vectors clocks inserted in the program. These vector clocks are inserted by the bytecode instrumentation toolkit-BCEL in order to monitor static member variables [52].

This chapter discusses various instrumentation tools (both commercial and research based) and then describes in detail the instrumentation approach followed in this project. Our instrumentation approach for trace collection is a hybrid one, it combines

Hyades tracing with bytecode instrumentation tool- JTrek. Most of the relevant events required for antipattern detection were collected using our instrumentation approach.

4.2 Instrumentation Review

The instrumentation tools can be divided into four categories based on the level at which instrumentation is performed:

- Operating system,
- Java Virtual Machine (which includes both custom and standard instrumentations such as profiling and debugging services),
- Source code or;
- Compiled code (bytecode).

4.2.1 JVM Level Instrumentation

Java Virtual Machine (JVM) instrumentation consists of modifying the existing JVM to provide the required data collection. An attractive feature of JVM instrumentation is access to information, which is unavailable with internal methods, such as byte and source code instrumentation.

Custom JVM level instrumentation suffers from the following disadvantages [42]:

- Reengineering of a JVM requires deep knowledge of the JVM, which is a complex software component, and could be error prone.
- The JVM uses Just-In-Time (JIT) compilation and Hot Spot dynamic compilation for performance enhancement [4]. When these features are enabled, simple modification of the bytecode interpreter unit is not sufficient [42]. One

also has to modify compilation, inlining, and interpreter units. This increases the complexity of JVM instrumentation.

- JVM has been updated frequently (there have been four major and two minor updates during the last four years – v1.0 to v1.3 being major and v1.4 to v1.5 being minor updates); modification of the JVM for monitoring should be done as frequently.

Custom JVM instrumentation is difficult, thus many tools previously built over custom instrumentation, migrate to standardized instrumentation.

Because of the above limitations, JVM level instrumentation does not seem to be a practical solution. To overcome the above-mentioned difficulties of custom instrumentation, modern JVM are already instrumented with standardized and extensible profiling (JVMPI) and debugging (JPDA) services. With the development of the JVM profiling interface, custom JVM instrumentations have become rare. In fact, some tools, such as JinSight, abandoned them in favor of JVMPI.

The standard debugging and profiling architectures (JVMPI and JPDA) of JVM are discussed in detail in Sections 4.3.2 and 4.3.4.

4.2.1.1 Java Virtual Machine Based Instrumentation Tool

4.2.1.1.1 Tool: JinSight

JinSight developed at IBM AlphaWorks is a tool to visualize and explore a Java program's run-time behavior. It is useful for performance analysis and debugging of Java program. It displays performance bottlenecks, object creation and garbage collection, execution sequences, thread interactions, and object references [40].

JinSight consists of two parts:

1. *Instrumented Java Virtual Machine*, which inserts the instruction and these instructions are executed as Java program runs. As the program runs, it produces a Jinsight trace file with information about the execution sequence and objects of the program. The user can choose options to turn tracing on and off, to limit the type of information recorded, and to mark significant events in the trace file. Filtering and control over level of the detail are particularly useful since tracing every detail of a program's execution will generate large size trace rapidly (30MB/min); the resulting trace quickly running into the hundreds of megabytes and more.
2. *JinSight visualizer*, which reads the trace data and presents graphical views of program execution, recurring method call patterns, object interconnection, call graph etc.

4.2.1.2 Java Profiling Interface

The JVMPI (Java Virtual Machine Profiling Interface) is a bidirectional function call experimental interface between the Java virtual machine and an in-process profiler agent [39]. The virtual machine notifies the profiler agent of various events, e.g. memory allocation, thread-start and lock contention etc. On the other hand, the profiler agent can issue requests for more information through the JVMPI. For example, the profiler agent can turn on/off a specific event notification, request a dump (snapshot) of objects, threads, or lock (monitor) status, based on the needs of the profiler front-end. A proof of concept profiler agent is provided within Sun SDK since version 1.2.

The possible monitored events using the JVMPI are:

- Method enter and exit
- Object alloc, move, and free
- Heap arena create and delete
- Garbage Collection start and finish
- JNI global reference alloc and free
- JNI weak global reference alloc and free
- Compiled method load and unload
- Thread start and end
- Class file data ready for instrumentation
- Class load and unload
- Contended Java monitor wait to enter, entered, and exit
- Contended raw monitor wait to enter, entered, and exit
- Java monitor wait and waited
- Monitor dump
- Heap dump
- Object dump
- Request to dump or reset profiling data
- Java virtual machine initialization and shutdown

4.2.1.2.1 Profiling Tools

Tool: Hyades

Hyades, an Eclipse project provides an open source platform for Automated Software Quality (ASQ) tools, and a range of open source reference implementations of ASQ tooling for testing, tracing and monitoring software systems. Hyades provide an extensible framework and infrastructure that embrace automated testing, trace, profiling, monitoring, and asset management. The goal of the Hyades project is to bring ASQ tools into the Eclipse environment in a consistent way that maximizes integration with tools used in the other processes of the software lifecycle [34].

The Hyades project offers a Java Profiling Agent that collects the following events:

- trace start/end
- method call/return/entry/exit
- thread start/end
- exception throw
- object allocation/free/move
- JVM initialization/shutdown
- garbage collection start/end

Lock contention events are not supported. The collected events are stored in XML compliant files.

4.2.1.3 Java Debugging Architecture

JPDA (Java Platform Debugger Architecture) is a three-tiered debugging architecture that allows tool developers to easily create remote debugger applications, which run portably. The architecture is standardized and supported by most JVM implementations [38].

Certain functionalities of JPDA and JVMPI overlap. For example it allows more control and interaction, namely, to manipulate (suspend, resume, stop, ...) threads, add/remove breakpoints, get/set the value of a local variable, watch field access, and change memory allocation scheme, as well as line by line execution.

The possible observed events are:

- method entry and exit
- field access and modification
- thread end and start
- class load, unload and preparation
- death and initialization of virtual machine
- single step execution and breakpoint events

However local variables and arrays are not observed. To make them observable source code modification is recommended to transform arrays and local variables into observable entities.

4.2.1.4 Java Platform Profiling Architecture of J2SE 5.0

The J2SE 5.0 (Java 2, Platform Standard Edition), currently the latest version of Java release, provides comprehensive monitoring and management support: instrumentation to observe the Java virtual machine, Java Management Extensions (JMX) framework and remote access protocols [35].

The JVM Monitoring & Management API specifies a comprehensive set of instrumentation of JVM internals to allow a running JVM to monitor. This information is accessed through JMX (JSR-003) MBeans and can accessed be locally within the Java address space or remotely using the JMX remote interface.

J2SE 5.0 provides the following APIs for monitoring and management [35]:

1. *Java Virtual Machine Monitoring and Management API*: The `java.lang.management` API enables monitoring and managing the Java

virtual machine and the underlying operating system. The API enables applications to monitor themselves and enables JMX-compliant tools to monitor and manage a virtual machine locally and remotely.

2. *Sun Management Platform Extension*: The `com.sun.management` package contains Sun Microsystems' platform extension to the `java.lang.management` API and the management interface for some other components of the platform.
3. *Logging Monitoring and Management Interface*: The `java.util.logging.LoggingMXBean` interface enables us to retrieve and set logging information.
4. *Java Management Extensions (JMX)*: The JMX API defines the architecture, design patterns, interfaces, and services for application and network management and monitoring in Java. The APIs are based on the JMX Specification.

4.2.2 Source Code Level Instrumentation

Source code instrumentation is to adds extra source code (for example instructions, packages etc) called probes to report the events in the program to be analyzed [10]. Advantages of source code instrumentation are:

- Source code is more naturally understood and thus allows a custom instrumentation.
- Source code instrumentation eliminates the need for understanding the JVM and the actions of the compiler.
- Source code instrumentation is portable over platforms and machines.

One major disadvantage is that source code is required, and is not always available.

4.2.2.1 Source-Code Level Instrumentation Tool

Tool: JavaScope

JavaScope was developed at Sun Microsystems. JavaScope is a set of software programs to determine how well a Java program or one or more Java source files are tested (test coverage measurement). It provides a tool to instrument the application and a browser to view resulting data. It can instrument everything, but offers no control over instrumentation techniques, location, or ability to add probes.

4.2.3 Bytecode Instrumentation

The Java compiler converts the Java source code to the class file format [44]. Instead of modifying source code (which is source code instrumentation), the resulting Java bytecode is modified. An executable Java program consists of a set of classfiles, a classfile contains definition of one class. A Java classfile is loaded into running Java virtual machine at run-time and these classfiles are dynamically linked at run-time. In order to link classfiles dynamically, a classfile contains symbolic information such as string constants, class names, field names, method names, local variable names, and other constants that are referred to within the classfile. This information in a classfile helps in instrumentation.

An instrumentor is a program which perform instrumentation, instrumentor takes two inputs namely, Java classfile (*.class) and instrumentation specification, which contain information such as variables/methods to be monitored and thus instrumented. Based on these two inputs, the instrumentor inserts instructions in the target classfile.

Bytecode instrumentation is of two types: dynamic and static instrumentation. Dynamic instrumentation is performed during program execution, while static instrumentation is performed prior to execution.

4.2.3.1 Java Bytecode Format

The understanding of the Java Bytecode format is important for the bytecode instrumentation. Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C, C++, or other compiled programs. Knowing the assembler instructions that are generated by the compiler for the source code we write, helps to know how to code differently to achieve memory or performance goals [21].

The content of a Java class file starts with a header containing a "magic number" `(0xCAFEBABE)` and the version number, followed by the *constant pool*, which can be roughly thought of as the text segment of an executable, the *access rights* of the class encoded by a bit mask, a list of interfaces implemented by the class, lists containing the fields and methods of the class, and finally the *class attributes* [14]. Attributes are a way of putting additional, user-defined information into class file data structures.

The Bytecode translation of a well-known statement `"System.out.println ("Hello World")"` is:

```
getstatic     java.lang.System.out
Idc            "Hello World"
invokeVirtual     java.io.printstream.Println
```

The first instruction loads the contents of the field `out` of class `java.lang.System` onto the operand stack. This is an instance of the class `java.io.PrintStream`. The `ldc` (Load constant) pushes a reference to the string

"Hello world" on the stack. The next instruction invokes the instance method `println` which takes both values as parameters (Instance methods always implicitly take an instance reference as their first argument).

4.2.3.2 Advantages of Bytecode Instrumentation

Bytecode instrumentation offers numerous advantages [42]:

1. A class file, the unit of Java bytecode contains the rich symbolic information about the system such as method names, global variable names and local variables that is useful for automatic instrumentation.
2. Many of the issues of interest for run-time monitoring such as actual access to variables and power consumption of instructions are revealed precisely at the bytecode level.
3. Java bytecode prohibits pointer arithmetic, which enables the detection of the updating of variables and also it is strongly typed.
4. Bytecode instrumentation adds the least overhead to a Java programs execution.
5. Java Bytecode is platform independent.
6. Many high level languages like Ada and Lisp compile their source code to Java Bytecode. Thus techniques and tools developed for Java could apply to Ada and Lisp [42].
7. The tool support for byte code instrumentation is better than source code or JVM level instrumentation support.

The main disadvantage in developing of bytecode instrumentation tool seems to be a need for deep knowledge of Java bytecode language.

4.2.3.3 Java Bytecode Instrumentation Tools/Toolkits

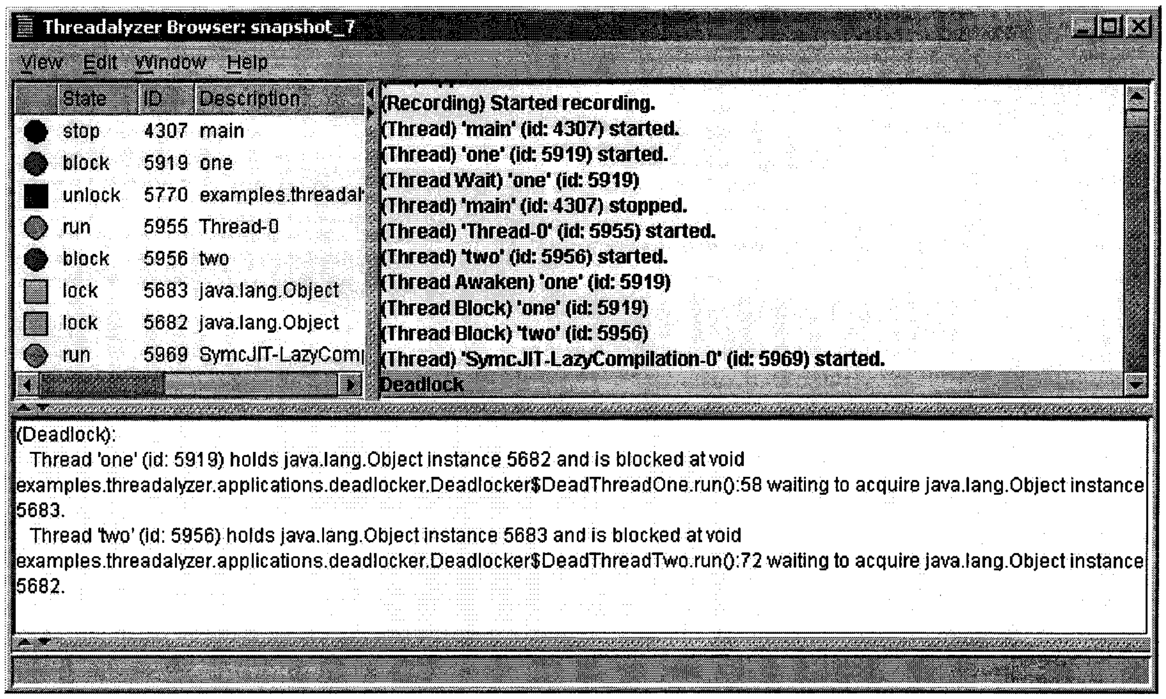
The bytecode instrumentation tools such as JTrek and BCEL were being successfully used in several runtime analysis projects such as JavaMac, JPaX and JMPaX respectively. In JMPaX project, BCEL was chosen for instrumentation purpose above JTrek, because the vector clocks can be easily inserted using BCEL than JTrek. Some of the bytecode level instrumentation tools are listed here.

4.2.3.3.1 Tool: JProbe Threadalyzer

JProbe Threadalyzer detects thread problems that can threaten the application performance [41]. It analyzes the Java code to:

- pinpoint the cause of stalls, deadlocks and race conditions;
- predict deadlocks with advanced lock analysis;
- visualize the status of all running threads;
- view precise source location, where problems occur;
- avoid data corruption due to race conditions.

The event collection is performed by instrumentation. The events relevant to the property under analysis are logged into "snapshot" files, and could be visualized (see Figure 4.1) or converted into text or XML formats.



	State	ID	Description
●	stop	4307	main
●	block	5919	one
■	unlock	5770	examples.threadal
●	run	5955	Thread-0
●	block	5956	two
▨	lock	5683	java.lang.Object
▨	lock	5682	java.lang.Object
●	run	5969	SymcJIT-LazyComi

```
(Recording) Started recording.
(Thread) 'main' (id: 4307) started.
(Thread) 'one' (id: 5919) started.
(Thread Wait) 'one' (id: 5919)
(Thread) 'main' (id: 4307) stopped.
(Thread) 'Thread-0' (id: 5955) started.
(Thread) 'two' (id: 5956) started.
(Thread Awaken) 'one' (id: 5919)
(Thread Block) 'one' (id: 5919)
(Thread Block) 'two' (id: 5956)
(Thread) 'SymcJIT-LazyCompilation-0' (id: 5969) started.
Deadlock
```

```
(Deadlock):
  Thread 'one' (id: 5919) holds java.lang.Object instance 5682 and is blocked at void
examples.threadalyzer.applications.deadlocker.Deadlocker$DeadThreadOne.run():58 waiting to acquire java.lang.Object instance
5683.
  Thread 'two' (id: 5956) holds java.lang.Object instance 5683 and is blocked at void
examples.threadalyzer.applications.deadlocker.Deadlocker$DeadThreadTwo.run():72 waiting to acquire java.lang.Object instance
5682.
```

Figure 4.1: A Trace Snapshot from JProbe

Threadalyzer flags thread stalls as potential problems in the analyzed programs. Threadalyzer leaves it to the programmer to control how long a thread must be inactive before being flagged as stalled.

Threadalyzer is capable of identifying and flagging any data races it encounters while running the program. The data race detection process is usually resource intensive and may lower down Threadalyzer performance. Powerful lock analyzers help in identifying thread problems before they happen.

4.2.3.3.2 Tool: JTrek

JTrek was developed at Digital Corporation (Digital has now merged with Compaq and Hewlett-Packard). JTrek is a platform independent advanced technology written in Java for troubleshooting Java applications. JTrek consists of the *Trek* class library, which enables Java developers to write Java applications that analyze and modify Java class files.

It is used as an instrumentation tool in Java PathExplorer (JPaX), a Run-time Verification Tool. JTreK reads Java classfiles, traverses them as abstract syntax trees while examining their contents and inserts new code. The inserted code can access the contents of the method call-time stack at run time. JTreK is also used as an instrumentation tool in Java-Mac "A run-time assurance tool for Java Programs" developed at University of Pennsylvania, U.S.A.

4.2.3.3.3 ToolKit: Byte Code Engineering Library (BCEL)

The *Byte Code Engineering Library* (formerly known as JavaClass) is a toolkit for the static analysis and dynamic transformation of Java class files [14]. It enables developers to implement the desired features on a high level of abstraction without handling the internal details of the Java class files. It is intended to give users a convenient possibility to analyze, create, and manipulate Java class files.

BCEL was designed to model bytecode in an object-oriented way by mapping each part of a class file to a corresponding object. Particular bytecode instructions may be inserted or deleted by using instruction lists and applying changes to existing class files. Efficient bytecode transformations can be done by using *compound instructions* as a substitute for a whole set of instructions of the same category. For example, an artificial push instruction can be used to push arbitrary integer values to the operand stack. With the aid of run-time reflection, i.e., meta-level programming, the bytecode of a method can be reloaded at run time.

Java MultiPathExplorer (JMPaX), a trace verification tool, uses the BCEL Java library to modify Java class files for collecting data access events and maintaining a vector clock, which identifies a partial order among events.

4.3 Our Trace Collection and Instrumentation Approach

4.3.1 Introduction

In our runtime analysis approach, for instrumentation purpose we developed an integrated instrumentation approach, which combines two approaches:

- profile interface based approach – Hyades tracing,
- Bytecode instrumentation tool – JTrek.

By using this integrated instrumentation approach we could reap the benefits of both the approaches. The Hyades framework consists of Java profiling interface which provides non-intrusive trace collection, for events such as method entry/exit, trace start/end, exception throw, etc. These events are emitted as XML fragments during execution of target application. The target application is executed with Java profiling agent attached to JVM to capture and record the Java application behaviour.

The limitation of Hyades tracing is that events such as; monitor enter/exit, variable write and read could not be collected. These events are required for detection of concurrency errors, such as deadlocks and data races. To collect these additional events the Hyades tracing is supplemented with customized bytecode instrumentation tool - JTreK.

For example, to log additional events such as `monitorenter` and `monitorexit` in the trace, JTreK instruments the target classfile with empty methods

`Object.lockentry` and `Object.lockexit` before the start and end of synchronized block respectively. Similarly to log variable updates events in the trace, JTrek instruments the target class file with empty methods `variable_write` and `variable_read` before the monitored variable write or read operation.

When this instrumented classfile is executed on Hyades Platform, the additional events such as; monitor enter/exit and variable updates are logged in execution trace.

4.3.2 Hyades Tracing

The Hyades profiling and tracing tool consists of the Profiling and Logging Perspective. It enables to profile the application, to interact with the application when profiling, and to examine the application for concurrency or memory related problems. The Profiling Tool collects events related to the Java program's run-time behaviour. Events collected from a profiling session are saved to an external file in XML format for later analysis.

4.3.2.1 Event Structure and Attributes

The data output of the Java Profiling Agent (Hyades Tracing) is a set of XML elements, which are either emitted as fragments within a non-XML trace stream or as part of a valid XML document [34]. Here we briefly discuss the event structure (in XML format) of the execution trace of Java multithreaded appliocations.

The event structure consists of the following elements [34]:

- IDs
- Common attributes
- Structural elements
- Trace behaviour elements

- Class elements
- Object elements
- Method elements
- Line elements
- Memory management elements
- Exception elements
- JVM elements
- Monitor elements
- All trace elements

4.3.2.1.1 IDs

Attributes of the elements have various kinds of IDs. Threads, classes, methods, and objects each have unique IDs, represented by threadId, classId, methodIs and objectId respectively. Each ID has a defining element and an undefining element. A defining element provides the information related to an ID, for example, the defining element for a thread ID contains the name of the thread.

An ID is valid until its undefining element arrives. An undefining element invalidates the ID, whose value may be reused later as a different kind of ID. The value of a thread ID, for example, may be redefined as a method ID after the thread ends.

4.3.2.1.2 Common attributes

Many event elements share the same attributes. The following attributes appear on more than one element:

Time

The time, at which the event starts, the format of the time attribute is "*utc.fff*" where *utc* is the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time, according to the system clock. It is expressed as an unsigned 32-bit value formatted as a string.

fff is the fraction of seconds to the highest precision that can be retrieved.

ThreadId/ThreadIdRef

ThreadId defines and threadIdRef refers to the thread in which the element occurred. ThreadId's are unique within the scope of a trace regardless of how many threads are started and ended. It is expressed as an unsigned 32-bit value formatted as a string.

MethodId/MethodIdRef

MethodId defines and methodIdRef refers to the method that the element is associated with. It is expressed as a 32-bit unsigned value in string format.

ObjId/ObjIdRef

ObjId defines and objIdRef refers to the object associated with the event. It is expressed as a 32-bit unsigned value in string format.

ClassId/ClassIdRef

ClassId defines and classIdRef refers to the class associated with the event. It is expressed as a 32-bit unsigned value in string format.

TraceId/TraceIdRef

TraceId defines and traceIdRef refers to a UUID (Universal unique identifier) that uniquely identifies the trace instance.

4.3.2.1.3 Structural elements

When emitted as part of a valid XML document, the trace information is contained under a root TRACE element.

```
<TRACE>
<node/>
<processCreate/>
<agentCreate/>
    ...
 all other events
    ...
<agentDestroy/>
</TRACE>
```

Trace Behaviour Elements

The following elements provide information about the trace as a whole:

- Node
- ProcessCreate
- AgentCreate
- AgentDestroy
- TraceStart
- TraceEnd
- ProcessSuspend
- ProcessResume
- Option
- Filter

Thread elements

The following elements provide information about threads. Other elements will point to a Thread element's thread_id to identify the thread they are running in;

ThreadEnd,
ThreadStart.

Class Elements

ClassDef;
MethodDef,

Although technically part of the classDef event, the method element is broken out into a separate element so that it can be optionally output only when referenced.

Object Elements

The element objAlloc traces storage allocation. It has its own section because it also holds identity information for an object, which can be referred to by method events associated with the object, such as a methodEntry event.

Method elements

The following elements provide information about methods:

- MethodEntry
- MethodExit
- MethodCall *(Deprecated)*
- MethodReturn *(Deprecated)*
- InvocationContext
- ObjDef
- Value
- MethodCount

MethodEntry and methodExit are output when a method is entered, and when the method returns respectively. MethodCall and methodReturn are output when a method is about to be called, and after a method returns.

The InvocationContext element holds identity information so that a methodEntry can determine who invoked the method regardless of location. InvocationContext information will identify either a methodCall or methodEntry of a remote agent for distributed invocations.

The objDef element holds identity information for an object, which can be referred to by elements associated with the object, such as the value element.

The value element is used to reference a data value, either for parameter values in a methodCall, or for the return value of a methodReturn.

MethodCount tracks the number of times a particular method has been invoked. This element is designed to aid in collecting code coverage information. A methodCount element is produced for every method for every class loaded by the application.

4.3.3 Bytecode Instrumentation Using JTrek

Here the customized bytecode instrumentation using JTrek is described in detail. The JTrek makes easily possible to process Java class files, examine their contents and insert new code. The inserted code access the contents of various runtime data structures, such as the call-time stack, and when executed on Hyades platform emit events carrying this extracted information to the trace or console output [27].

The instrumentation works as: JTrek iterates through the bytecode instructions of the target program and uses callbacks to perform user-specific instrumentation. Iteration may be at the level of Java statements or individual bytecode instructions.

JTrek allows insertion of certain types of code, but does not allow definition of new local variables, fields, or methods. At each byte code instruction, JTrek calls a method `void at (Instruction instr)`, which we override, and to which the current instruction is passed as parameter. JTrek provides a large variety of classes for instrumentation such as Instruction, Code, Class, Statement, and Method, each targeting a particular Java construct that can be accessed from an instruction. Each method class contain various kinds of information. For example, in the Instruction class, the method `Statement getStatement()` returns an object of the class `Statement`, representing the statement in which the instruction occurs. The `Statement` class in turn contains a method, `Method getMethod()`, returning the method in which the statement occurs. For example, the method in which an `instruction instr` occurs can be obtained by the expression: `instr.getStatement().getMethod()`.

The `void at (Instruction instr)` overridden method is inserted before the monitored variable/method; a switch-statement branches out depending on the opcode of the instruction. In case an instruction is for instrumentation, JTrek inserts the call of a method either before or after the instruction.

4.3.3.1 Additional Events Logged Using the Bytecode Instrumentation

Here the additional events logged in the execution trace using the bytecode instrumentation are listed. The instrumentation procedure followed to log these events will be explained below.

4.3.3.1.1 Variable Updates

For data race analysis, information need to be extracted, such as when and which variables are accessed, their updated values and threads referencing these variables.

For example, to monitor a *class variable x* of `int` datatype and log the events such as; `<methodEntry variableWrite>`, `<methodEntry VariableRead>`, and the updated values of the variables in an execution trace and console output respectively, JTrek iterates through bytecode instructions and searchs for `putstatic` instruction (an instruction updating a class variable). This `putstatic` instruction has the class variable name, the type of the variable, and the type of the parent class as operand. The iteration is performed using the `dotrek` method of Trek class. The empty methods such as `variable_write ()` or `variable_read ()` are inserted after the `putstatic` instruction. In the `variable_write ()` or `variable_read ()` methods, parameters such as variable type and its values are passed as operand. When this instrumented bytecode is executed on Hyades platform, the events such as `<method Entry variable_write>` or `<method Entry variable_read>` are logged in the execution trace and the variables' parameters such as variable type and variable values are printed at the console output.

For data race analysis and other concurrency errors analysis, we combine both the traces namely the Hyades trace and the trace obtained at the console output are combined.

The primitive field variables, local variables, and `monitorentry/exit` are monitored. Below instrumentation details of these two variables types are provided:

Primitive Field Variables

A field variable is either a class variable or an instance variable. A class variable is updated by `putstatic` instruction. An instance variable is updated by `putfield` instruction. Both instructions (`putstatic` and `putfield`) have variable name and a parent class name as parameters. Both instructions take top stack operand value as the updated value of the variable.

Local Variables

Local variable values are updated by instructions such as `<T>store`, `<T>store_n` and `iinc` where `<T>` stands for primitive type and `<n>` $\in$ `{0, 1, 2, 3}`. These instructions contain an index to a local variable as an operand.

4.3.3.1.2 Monitor Enter and Exit

Detection of concurrency errors, such as deadlocks and data races, requires information, such as when locks are acquired and released. At the JVM level, a lock is obtained when a synchronized method - `monitorenter` instruction is executed.

For example, a `monitorenter` instruction indicates that a thread acquires a lock when entering a synchronized statement, similarly `monitorexit` instruction, indicates that a thread releases a lock. Detection of concurrency errors, such as deadlocks and dataraces, requires extraction of information such as which object is locked and which thread does it. Thus instrumenting all `monitorenter` and `monitorexit` instructions correctly tracks the number of locks held by a thread on an object relating to the synchronized statements. When this instrumented bytecode is executed on the Hyades platform, the events such as `<method entry LockAcquire>` and `<method exit LockRelease>` are logged in the execution trace.

Thus instrumenting all `monitorenter` and `monitorexit` instructions correctly tracks the number of locks held by a thread on an object relating to the synchronized statements. Figure 4.2 shows the original target source code and its compiled bytecode (Figure 4.3), which is then instrumented (Figure 4.4) to log events such as `<methodentry lockAcquire>` and `<methodexit LockRelease>` in the execution trace.

```
private void forksAvailable(int i) {
synchronized (convey[i]) {
convey[i].notify();
 }
     }
```

Figure 4.2: Sample of the Source Code

```
private void forksAvailable(int i)
{
//     0     0:aload_0
//     1     1:getfield        #36  <Field Object[] convey>
//     2     4:iload_1
//     3     5:aaload
//     4     6:dup
//     5     7:astore_2
//     6     8:monitorenter
// try 9 31 handler(s) 31
//     7     9:aload_0
//     8    10:getfield        #36  <Field Object[] convey>
//     9    13:iload_1
//    10    14:aaload
//    11    15:invokevirtual   #100 <Method void
Object.notify()>
//    12    18:getstatic       #47  <Field PrintStream
System.out>
//    13    21:ldc1            #102 <String "notify method is
entered here">
//    14    23:invokevirtual   #65  <Method void
PrintStream.println(String)>
//    15    26:aload_2
//    16    27:monitorexit
//    17    28:goto              34
// finally
//    18    31:aload_2
//    19    32:monitorexit
//    20    33:athrow
//    21    34:return
      }
```

Figure 4.3: Sample of Uninstrumented Bytecode

```
private void forksAvailable(int i)
{
//     0     0:aload_0
//     1     1:getfield          #36  <Field Object[] convey>
//     2     4:iload_1
//     3     5:aaload
//     4     6:dup
//     5     7:astore_2
//     6     8:monitorenter
//     7     9:aload_0
//     8    10:getfield          #36  <Field Object[] convey>
//     9    13:iload_1
//    10    14:aaload
//    11    15:invokevirtual     #130 <Method void
Object.lock1Acquired()>
// try 18 49 handler(s) 49
//    12    18:aload_0
//    13    19:getfield          #36  <Field Object[] convey>
//    14    22:iload_1
//    15    23:aaload
//    16    24:invokevirtual     #100 <Method void
Object.notify()>
//    17    27:getstatic         #47  <Field PrintStream
System.out>
//    18    30:ldc1              #102 <String "notify method is
entered here">
//    19    32:invokevirtual     #65  <Method void
PrintStream.println(String)>
//    20    35:aload_2
//    21    36:monitorexit
//    22    37:aload_0
//    23    38:getfield          #36  <Field Object[] convey>
//    24    41:iload_1
//    25    42:aaload
//    26    43:invokevirtual    #133 <Method void
Object.lock1Release()>
//    27    46:goto             52
// finally
//    28    49:aload_2
//    29    50:monitorexit
//    30    51:athrow
//    31    52:return
}
```

Figure 4.4: Sample of Instrumented Bytecode

The sample of the instrumentor code (as shown in Figure 4.5) inserts the method `Object.lock1Acquire` after the `monitorenter` instruction. This instrumentor code inserts methods (in the target bytecode) such as `aload_0, getfield <Field Object [] convey>, iload_1, aaload, and invokevirtual <Method void Object.lock1Acquired ()>`.

For example, to insert `aload_0` (a bytecode instruction), the instrumentor instruction `code.append (42)` is written, which inserts the bytecode `aload_0`, the opcode "42" refers to `aload_0`. Similarly the bytecode instruction `getfield <Field Object [] convey >` is inserted by the instrumentor instruction `code.append (180, filterLock)` where the opcode "180" refers to instruction "`getfield`" and filterLock refers to `<Field Object [] convey>`. Similarly the bytecode instruction `invokevirtual <Method void Object.lock1Acquire>` is inserted by the instrumentor instruction; `code.append (182, monitorenter2)` where the opcode "182" refers to instruction "`invokevirtual`" and monitorenter2 refers to `< Method void Object.lock1Acquire>`.

```
Protected final void monitorEnterAfter 2(Instruction
instruction)

{

Code code = null;
if(instruction.next() != null)
code = Code.addAt(null, instruction.next());
else
code = Code.addAfter(null, instruction.getStatement());
code.append(42);
code.append(180,filterLock);
code.append(182,monitorenter2);
code.done();

 }
```

Figure 4.5: Sample of Instrumentor Code

The sample of the collected execution trace (Figure 4.6) is obtained by executing the instrumented bytecode (Figure 4.4) on the Hyades platform. The execution trace shows that the event lock1Acquire methodentry (`<lockAcquire methodentry>`) is logged before the notify methodentry. Both these method entries refer to the same `objectIdRef` "8605".

```
<! -- The additional events obtained by instrumentation --

!>

<methodDef name="lock1Acquired" signature="()V"
startLineNumber="247" endLineNumber="248" methodId="107"
classIdRef="120"/>

<methodEntry threadIdRef="7" time="1093561048.943165800"
methodIdRef="107" objIdRef="8605" classIdRef="120"
ticket="994" stackDepth="5"/>

<methodDef name="notify" signature="()V" methodId="103"
classIdRef="120"/>

<methodEntry threadIdRef="7" time="1093561048.957960600"
methodIdRef="103" objIdRef="8605" classIdRef="120"
ticket="1162" stackDepth="5"/>
```

```
<methodDef name="lock1Release" signature="()V"
startLineNumber="262" endLineNumber="264" methodId="105"
classIdRef="120"/>
<methodEntry threadIdRef="6" time="1093561048.974705000"
methodIdRef="105" objIdRef="8605" classIdRef="120"
ticket="880" stackDepth="4"/>
```

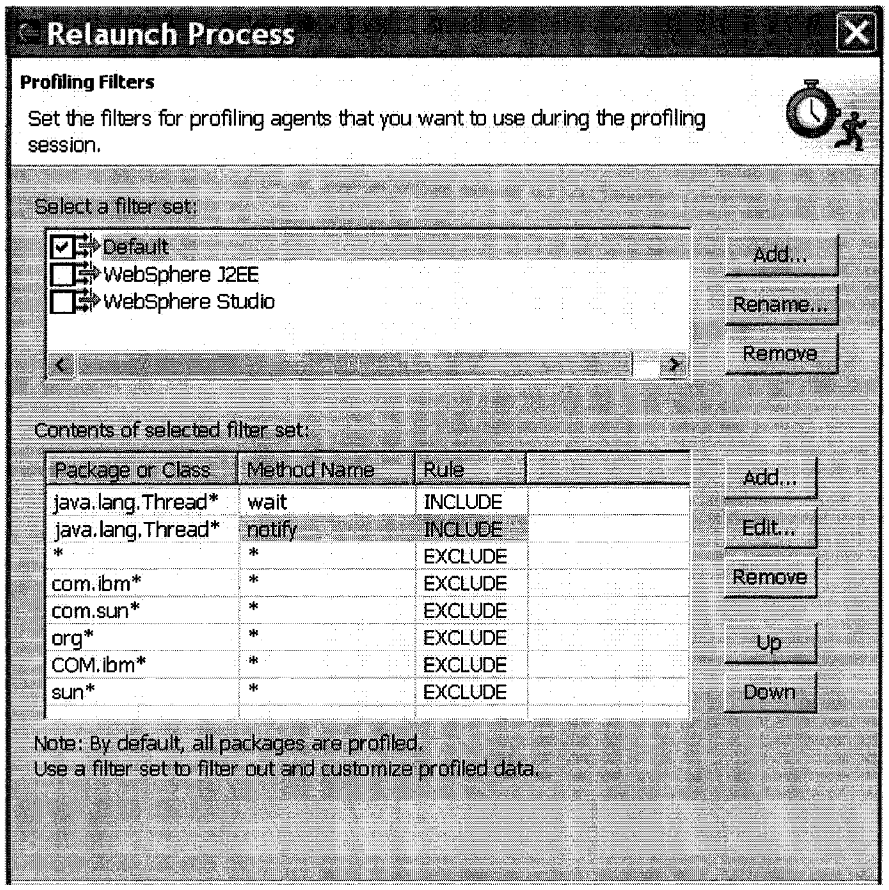
Figure 4.6: Trace Sample of the Instrumented program

4.3.4 Trace Reduction

The execution trace obtained by the Hyades framework is large, usually in the order of 6-20 MB. It becomes difficult to unmarshall, handle, and verify the properties on such a large size of trace. The large size of the trace is caused by the filtering in most of the irrelevant events (These events are irrelevant for the verification of properties of our interest).

To reduce the trace size, fine-tuned filters of Hyades framework are used to filter-in the relevant events and filter-out the irrelevant one. For example, for deadlock detection only the `wait` and `notify` `methodentries` events are required in the execution trace. To filter-in these events only, filter setup such as "`Java.lang.Thread.* Wait Include`", "`Java.lang.Thread.* Notify Include`" and "`* * Exclude`" is written. This filter setup will only include wait and notify methodentries in the trace, excluding most other `methodentries`, with the exception of a few. A snapshot of this Hyades filter is shown below (Figure 4.7).

Using these filters, a smaller trace size was obtained of the order of 63-200 KB. The size of the reduced trace depends upon the type of filters used, number of the filters used, size of the original program, etc. The reduction in the trace size reduces the property verification time.



Relaunch Process

Profiling Filters

Set the filters for profiling agents that you want to use during the profiling session.

Select a filter set:

- [✓] Default
- [ ] WebSphere J2EE
- [ ] WebSphere Studio

Contents of selected filter set:

Package or Class	Method Name	Rule
java.lang.Thread*	wait	INCLUDE
java.lang.Thread*	notify	INCLUDE
*	*	EXCLUDE
com.ibm*	*	EXCLUDE
com.sun*	*	EXCLUDE
org*	*	EXCLUDE
COM.ibm*	*	EXCLUDE
sun*	*	EXCLUDE

Note: By default, all packages are profiled.
Use a filter set to filter out and customize profiled data.

Figure 4.7: Snapshot of Hyades Filter

4.3.5 Benefits and Limitation of our Instrumentation Approach

By using integrated instrumentation approach, we can reap the benefits of both the approaches, and limit their shortcomings. Here the benefits and limitations of our instrumentation approach are listed.

4.3.5.1 Benefits of our Instrumentation Approach

1. *Availability of Parsers*: The execution trace obtained from our instrumentation approach is in XML format. To analyze this trace for an antipattern, parsing is required. There are many Java-XML parsers available; among the most popular are SAX, DOM, and JAXB.

2. *TimeStamp Information*: The events logged such as `<method entry>` and `<method exit>` contain timestamp attributes. This timestamp information is required in verifying the antipattern "`wait` stall", because the "wait stall" detection requires the timing comparison between the wait methodentry and wait methodexit events for each thread.
3. *Trace Reduction*: The filters, available in Hyades framework, can reduce large trace size to the order of 30 MB. These filter-out the irrelevant events and filter-in the relevant events. The filters reduce the trace size, and thus lessen the trace verification time.
4. *Customized Instrumentation*: The instrumentation can be customized to obtain events and attributes as per the user requirement.
5. Most of the events of interest could be recorded and collected.

4.3.5.2 Limitation of our Instrumentation Approach

Additional Trace File: To monitor the updates of the variable, the values of variables are printed to a second file. Thus additional trace file is created, which adds to the analysis overhead and also consumes more memory.

4.4 Conclusion and Future Work

We presented an integrated instrumentation approach to extract necessary information, namely events and their attributes required for offline analysis. This integrated instrumentation approach helps to reap the benefits of both instrumentation approaches while avoiding their limitations. There are a few immediate tasks which need further work:

1. *Trace Integration*: Integrate the two traces obtained to monitor updates of the variables one obtained at the console output and trace obtained in XML format.
2. *Hyades 3.0 Probekit*: Experiment with trace collection using Hyades 3.0 (a latest version of Hyades), which contains the BCI (Bytecode Instrumentation) kit for bytecode instrumentation.
3. *Overhead Reduction*: Further reduce the overhead caused by instrumentation on the target program execution. A significant overhead can alter real-time properties of interest.
4. *Aspect Based Instrumentation*: Experiment with the aspect-based instrumentation, in which instrumentation specification is coded as an aspect.

Chapter 5. Custom Based Antipattern Detection

5.1 Introduction

The custom-based detection is a semiformal runtime approach to analyze the execution trace of a Java program (post-mortem analysis) against certain MT antipatterns coded as Java detectors. The offline analysis (post-mortem) was used to minimize the execution overhead. The MT antipatterns to be analyzed are first formally specified as FSM or EFSM, as shown in Section 5.3. These antipatterns are then coded as Java detectors, sample detectors are presented in Section 5.4. An execution trace of the program being analyzed is collected in XML format, which is then parsed and unmarshalled using Java-XML technology, JAXB. The execution trace is then analyzed for antipatterns using the Java detectors; the output of this analysis is a (possibly empty) set of property violations printed on console output.

The motivation of this approach can be summarized as:

1. *Develop a custom runtime analysis approach:* This approach is custom because the execution trace is verified for antipatterns using the custom Java detectors.
2. *Reduce overhead:* Reduce the execution overhead caused by the instrumentation. A lightweight and integrated instrumentation approach is used, which combines the bytecode instrumentation (a lightweight approach) with Hyades tracing - a non-intrusive JVMPI based trace collection approach.
3. *Simplify off-line analysis:* Develop a simplified approach, for the analysis of the execution trace of Java applications. The collected trace output is in XML format, therefore trace analysis is simplified since data in XML format is quite

representative (in XML, the data is collected in tags). There are many Java-XML parsers available such as SAX, DOM, and JAXB package which make it easy to read and analyze these execution traces of Java programs. In the particular case of runtime analysis in multiprocessor and multiplatform environment such as the Internet, where XML is used for data exchange, the XML-based trace analysis can prove to be very useful and feasible.

5.2 Workflow

The workflow of custom based detection approach is shown in Figure 5.1. It consists of two inputs: the Java classfile to be monitored (created using a standard Java compiler) and the MT antipatterns to be verified.

Custom detection is performed in three steps:

- Event Collection,
- Parsing Step,
- Analysis Step.

5.2.1 Event Collection

Here the relevant events and their attributes are collected in execution trace (in XML format) for MT antipattern analysis. The Java multithreaded application is executed on Hyades platform, which contains the Java Profiling Agent attached to a Java Virtual Machine (JVM) to capture and record the behaviour of a Java application. During the execution of the program, the events and their attributes are emitted as XML fragments from the profiling agent, which are collected in a trace. Most of the events required for antipatterns detection are collected by Hyades tracing with the exception of a few. The remaining events are collected by the custom instrumentation.

5.2.2 Parsing

Here the parsing and unmarshalling procedure is performed using a JAXB package. The JAXB package is a XML-Java technology developed by Sun Microsystems. The unmarshalling procedure is explained in more detail in Section 5.4.

5.2.3 Analysis

The collected events (obtained from the event collection) are analyzed for MT antipatterns. First the MT antipatterns are formally specified as FSM or EFSM, and then these antipatterns are coded as Java detectors. Formal specification of the antipattern (as FSM or EFSM) helps to better understand the antipattern, and thus its correct implementation in Java. The Java detectors are then used for antipatterns detection in the trace. The output of this analysis is a possible set of warnings, printed on a console output.

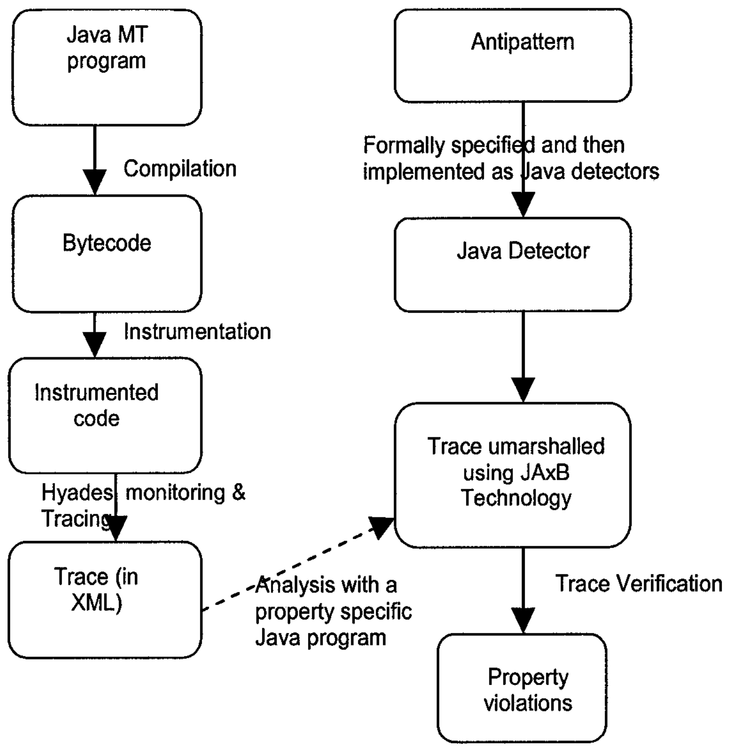

Java MT program
Compilation
Bytecode
Instrumentation
Instrumented code
Hyades monitoring & Tracing
Trace (in XML)
Analysis with a property specific Java program
Antipattern
Formally specified and then implemented as Java detectors
Java Detector
Trace umarshalled using JAxB Technology
Trace Verification
Property violations

Figure 5.1: Workflow of Custom Based Detection Approach

5.3 Antipattern Formalization

5.3.1 Formalization of the "Double Call of Start () Method" Antipattern

In order to correctly and efficiently implement the antipatterns, formal, automata like description as FSM and EFSM are created. The formal specification helps to correctly understand the antipattern and thus its correct implementation. The formal description of the antipattern is a necessary step in model checker based trace verification, and also, we believe, beneficial in custom detector implementation.

The antipattern "double call of the `start ()` method" is formalized as FSM and EFSM.

5.3.1.1 FSM Formalization of "Double Call of Start () Method"

This antipattern can be instantiated in a set of automata (finite state machines), where each automaton corresponds to a thread present in an execution trace, or it can be instantiated in single trace-independent extended automaton. In the first case, the automaton is specified as: the double circle represents the accepting state (state 0 and 1) and the full circle represents unacceptable or violation state (state 2), which indicates that the antipattern is detected. The transition is labeled with a tuple of the form (*o*,*m*,*c*), where *o* is the object owning method *m*, *m* is the method itself, and *c* is the calling thread. For example, in the FSM shown below (Figure 5.2) the transition is labeled as (*o*, start, *), which indicates that the `start` method is called on the thread object *o* (*o*.start), and the wild-card symbol * indicates that calling the thread can take any value. We assume that events that are not explicitly defined in a state are irrelevant to the state and thus discarded in this state. In other words, FSM does not change state when an unspecified stimulus arrives.

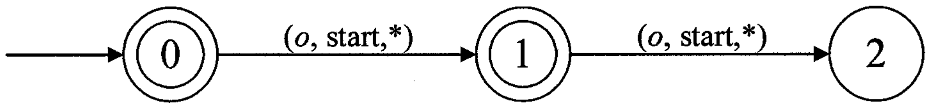


Figure 5.2: FSM Formalization of Double Start ()

Implementation of a "double `start ()`" antipattern detector based on the above automaton involves trace pre-processing to build the list of thread ids (more exactly ids of corresponding objects), and then scanning the trace with evaluation of a set of automata. Similar preprocessing will be required for property verification, based on the model checker, unless the former provides a richer language than the automata. The FSM model can only use constants, thus the above model may only be applied to one particular object (thread). The FSM, extended with variables and operations on these variables is called Extended FSM (EFSM). In the next section the formalization of antipatterns using EFSM is discussed, where *o* is not a constant, but an input variable.

5.3.1.2 EFSM Formalization of "Double Call of Start () Method"

In a single extended machine the antipattern "double call of `start ()` method" can be formalized as shown in Figure 5.3 below. The lists of threads are initially declared empty as (L: = ∅) for EFSM formalization. When the next start call is on the thread object *o* and the thread object *o* is not contained in the list L, the thread object *o* is added to the thread list L.

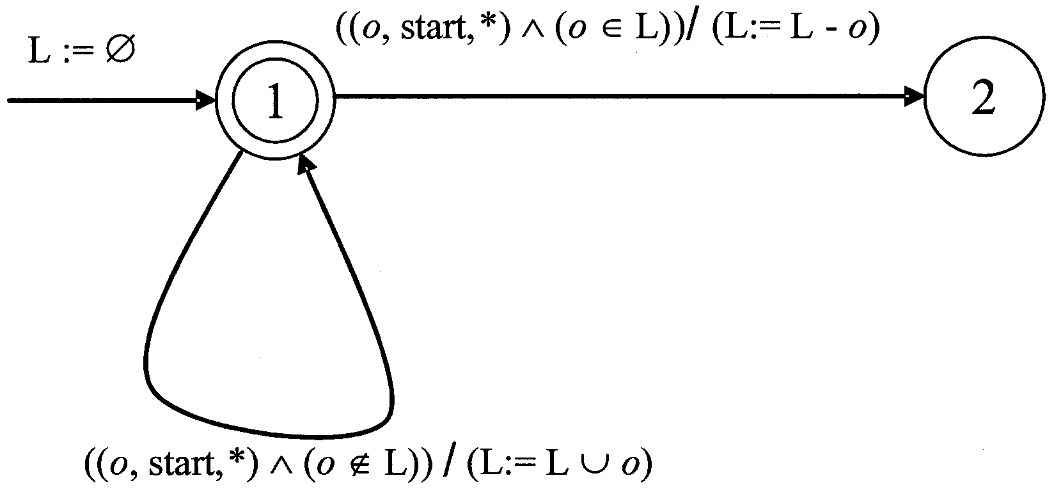

L := ∅
((o, start,*) ∧ (o ∈ L))/ (L:= L - o)
1
2
((o, start,*) ∧ (o ∉ L)) / (L:= L ∪ o)

Figure 5.3: EFSM Formalization of Double Start ()

5.3.2 Formalization of the "Premature Join () Call" Antipattern

The antipattern "premature `join ()` call" is formally specified as FSM and EFSM in a similar fashion.

5.3.2.1 FSM Formalization of "Premature Join () Call" Antipattern

The formal description of the "premature `join ()` call" antipattern can be represented in a set of automata (finite state machines), where each automaton corresponds to a thread (referred by `threadId`) present in an execution trace. The automaton is build, in which the double circle represents the acceptance state and the full circle represents the violation state; entering a violation state indicates that the antipattern is detected. Note that, when events are not explicitly defined in the state, the events are discarded.

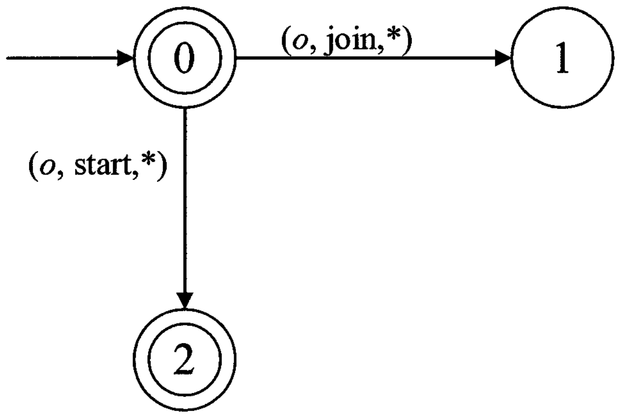


Figure 5.4: FSM Formalization of Premature Join ()

Implementation of the "premature `join ()` call" antipattern detector based on the above automaton involves trace pre-processing to build the list of thread Ids, and then scanning the trace for evaluation of automata. Informally it can be said that first the methodExit of `join` and methodExit of `run` are located for each thread *T*. Then the timestamps of `join`'s methodExit and `run`'s methodexit are compared. If the `run`'s methodexit happens before the `join`'s methodExit on a particular thread *T*, then the message "premature call of `join ()` method" detected is printed.

5.3.2.2 EFSM Formalization of "Premature Join () Call" Antipattern

In a single extended machine (EFSM) the antipattern can be formalized as:

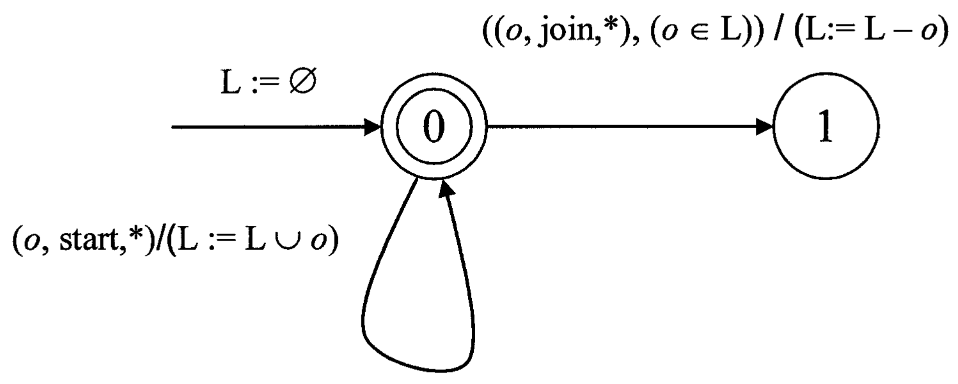

((o, join,*), (o ∈ L)) / (L:= L – o)
L := ∅
0
1
(o, start,*)/(L := L ∪ o)

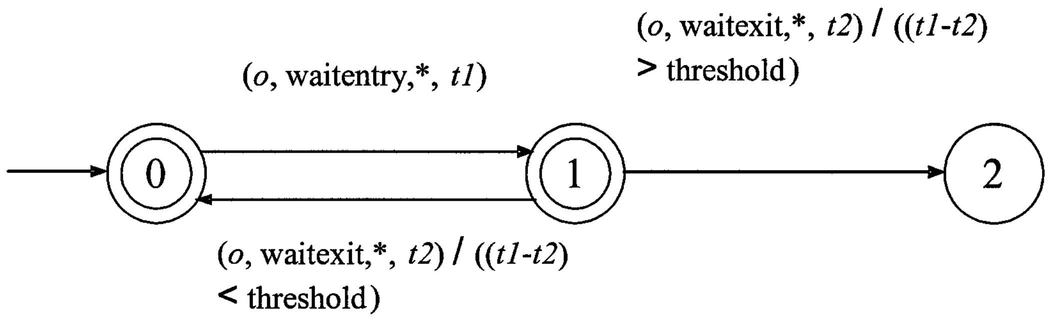
Figure 5.5: EFSM Formalization of Premature Join ()

5.3.3 Formalization of the "Wait () Stall" Antipattern

5.3.3.1 FSM Formalization of Wait () Stall

Here we formalize the "`wait ()` stall" antipattern with an automaton. Informally this antipattern - "`wait ()` stall" can occur when thread executes the `wait ()` method and wait duration exceeds the user specified threshold".

This antipattern can be represented in a set of automata (finite state machines), where each automaton corresponds to a thread (referred by `threadId`) present in an execution trace. The automaton as shown in Figure 5.6 is built: a full circle represents the violation state when reached indicates the `wait ()` stall detection; a double circle represents the accepting state. In the first transition the `wait` methodEntry is assigned the timestamp *t1* and in the second transition the `wait` methodExit is assigned the timestamp *t2*. If the difference ($t1 - t2$) is greater than the user specified threshold, then the message "`wait ()` stall" is detected is printed.

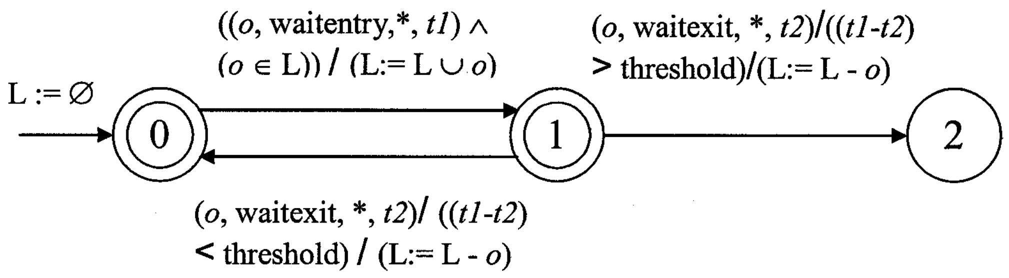


(o, waitentry,*, t1)
(o, waitexit,*, t2) / ((t1-t2) > threshold)
(o, waitexit,*, t2) / ((t1-t2) < threshold)
0
1
2

Figure 5.6: FSM Formalization of Wait () Stall

5.3.3.2 EFSM Formalization of Wait Stall

The "`Wait ()` Stall" antipattern can be formalized as Extended Finite State Machine (EFSM), as shown below:



((o, waitentry,*, t1) ∧ (o ∈ L)) / (L:= L ∪ o)
(o, waitexit, *, t2)/((t1-t2) > threshold)/(L:= L - o)
L := ∅
0
1
2
(o, waitexit, *, t2)/ ((t1-t2) < threshold) / (L:= L - o)

Figure 5.7: EFSM Formalization of Wait () Stall

5.4 Unmarshalling

All the implemented detectors rely on JAXB based unmarshalling of the XML representation of the trace. Unmarshalling is the process of reading an XML document and constructing a *content tree* of Java content objects. Each content object constructed corresponds directly to an instance in the input XML document of the corresponding schema component, and the content tree represents the content and structure of the document as a whole.

The `Unmarshaller` class (`javax.xml.bind.Unmarshaller`) deserialize the XML data into a Java content tree as shown in Figure 5.8, and also validates the XML data as it is unmarshalled, though such validation is optional.

```
// create a JAXBContext capable of handling classes
generated into the foo.jaxb package

JAXBContext jc = JAXBContext.newInstance ("foo.jaxb");

// create an Unmarshaller
Unmarshaller u = jc.createUnmarshaller();

// unmarshal a FooBar instance document into a tree of Java
//content objects composed of classes from the example
//package.

TRACE tr =
(TRACE)u.unmarshal( new FileInputStream
("C:\\jagmit\\profiling_data\\guest\\reduced_nativeagent.xm
l"));

//TRACE tr =
//(TRACE)u.unmarshal( new FileInputStream
("C:\\jagmit\\Trace\\sap_trace\\sap_trace.xml"));
```

Figure 5.8: Sample of Unmarshaller Code

5.5 Custom Detector's Implementation

Three custom detectors, namely "double call of `start ()` method", "premature call of `join ()` method" and "`wait ()` stall" were implemented. This section describes the implementation of these antipattern detectors.

5.5.1 Double Call of Start () Method Detector Implementation

The implementation of "double call of `start ()`" method antipattern detector according to their formal specification as FSM and EFSM is described in this section. A sample of the "double `start ()`"detector is shown in Figure 5.9. Initially the variable `start_count = 0` is defined. Then for each thread (referred by threadId*i*, where *i* is `threadId` number) we iterate and search for "double call for `start ()`" methodEntry. If such an instance is found, the objectIds are compared. If the objectId are found to be the same, the message "double call of `start ()` method" detected for threadId*i* is printed, otherwise the message "double call of `start ()` method" not detected for threadId*i* is printed. In the implementation of "double `start ()`" detector, the trace is traversed and the `objectIds` of the threadStart event are saved in the `thread_objId[]` array. Similarly the `objectIds` of `start` methodentry are saved in the `mentry_objId[]` array.

Then the `objectId` of the threadStart event is compared with the `objectId` of the first and second `start ()` methodentry. This comparison is made in the `thread_objId []` and `mentry_objId []` arrays obtained previously. If the objectIds are the same the message "double call of `start ()`" method detected is printed, otherwise the message "double call of `start ()`" method not detected is printed.

```
for (int k = 0; k<threadId_count; k++){
System.out.println("Checking double start for threadId = "
+ threadId[k]);
start_count =0;
for (int j=0; j<mentry_objId_count;j++)     {

if (thread_objId[k].equals(mentry_objId[j]))
{
start_count++;
}
}

if (start_count==2)
System.out.println ("Double call of the start method
detected for threadId = "+ threadId[k]);
else
System.out.println ("No Double call of start method
detected for thread id = " + threadId[k]);

}
```

Figure 5.9: Sample of Custom Detector for Double Start () Detection

5.5.2 Premature Call of Join () Method Detector Implementation

The definition of the premature call of the `join ()` method states "it consists in the invocation of the `join ()` method to the thread, which is not yet started" [22]. In the trace, the thread should terminate (`run` methodexit) after the corresponding thread `join` (`join`'s methodentry), and before the `join`'s methodexit. The trace verifies this ordering for every `threadId`, and if such an instance where the ordering is found violated, then the message "premature call of `join ( )` method" detected is printed.

As shown in Figure 5.10 for every thread, represented by the threadId*i* (where *i* is an integer of value 2, 5, 6 etc), the timestamp of `run`'s methodexit event is compared with the corresponding `join`'s methodexit event. If the timestamp of `run`'s methodexit is greater than the timestamp of corresponding `join`'s `methodexit`, the message "Premature call of `join ()` method detected for `threadId = i`" is printed, otherwise the message "No Premature call of `join ()` method detected for `threadId = i`" is printed. Also it is checked that `run`'s methodexit event and the corresponding `join's` methodexit should happen on the same object, but on different threads.

```
        // start of first for loop (runexit_count)
    for ( int j = 1; j < event_list.runexit_count; j++)
            {
            compare time = false;
        //  start of second for loop (joinexit count)
            for( int i = 1; i < event_list.joinexit_count
&& compare_time == false; i++)
        {
        if
(event list.runexit_obj_array[j].equals(event_list.joinexit
 obj array[i]))
        {
    //System.out.println("comparing the object Id");
    // starting (if loop) comparing threadid of runexit and
joinexit array
    if(event list.runexit thread array[j].compareTo(event_l
ist.joinexit thr               ead array[i]) < 0 ||
event list.runexit thread array[j].compareTo(event_list.joi
nexit thread array[i]) > 0)
            {
            //System.out.println("comparing the time");
            compare time = true;
            if (event list.runexit time array[j].compareTo
        (event_list.joinexit_time_array[i]) > 0)
            {
    System.out.println("Premature call of join() method
    detected for     threadId = " +
    event list.joinexit_thread_array[i]);
            }
    else if (event list.runexit time array[j].compareTo
            (event_list.joinexit_time_array[i]) < 0 )
            {
    System.out.println("No Premature call of join() method
for threadId =    " +
    event list.joinexit_thread_array[i]);
                       }
            }
    // starting (if loop) comparing threadid of runexit and
joinexit array
    } // end (if loop) comparing objectid of runexit and
joinexit array
    } // end of second for loop (joinexit count)
    } // end of first for loop (runexit_count)
```

Figure 5.10: Sample of Custom Detector for Premature Join () Detection

5.5.3 Wait () Stall Detector Implementation

The `wait ()` stall antipattern occurs when a thread waits (after calling the `wait ()` method) for more than the user specified threshold [22]. The methodentry and methodexit of `wait ()` method for each thread (referred by threadId) is located. Then the time difference between `wait ()` methodentry and its methodexit is calculated and compared with the user specified time period (in this detector it is set as 0.5 seconds). If the calculated time difference is more than the user specified period then the message "`wait ()` stall" detected is printed.

As shown in the Figure 5.11 for each thread represented by the threadId*i* (where *i* is an integer of value 2, 5, 6 etc), the time difference between `wait`'s methodentry and its corresponding `wait`'s methodexit is calculated. If the calculated time difference is more than the user specified period (0.5 seconds) then the message "`wait ()` stall" detected is printed, otherwise the message "No `wait ()` stall" detected is printed.

```
if(mexit_st3.equals(mstring)){

//System.out.println(mentry_st2);
System.out.println("Method Exit time for the wait method" +
mexit_st2);

Float tmp2 = new Float (mexit_st2);
ft2 = tmp2.floatValue();

ft3 = ft2-ft;

System.out.println(ft3);

if ( ft3<0.5)
System.out.println("Wait Stall Detected ");
else
System.out.println(" No wait stall Detected ");

}
```

Figure 5.11: Sample of Custom Detector for Wait () Stall Detection

5.6 Experiments

This section discusses the detection of the antipatterns, double call of `start ()` method, premature call of `join ()` method, and `wait ()` stall.

The experiments were performed on the following hardware and software configuration:

Hardware Configuration:

1.	CPU:	AMD Athlon 900 MHz,
2.	RAM:	512 Mbytes.

Software Used:

1.	Operating System:	WINDOWS 2000,
2.	Compiler:	Java 1.4,
3.	IDE:	Eclipse,
4.	Java - XML Tool:	JAXB (Java Architecture for XML Binding).

5.6.1 Antipattern: Double Call of Start () Method

Description: The `start ()` method is not supposed to be used more than once for the same thread.

Application 1 is a fragment of Java multi-threaded platform Guest [45].

Application 2 is a custom dining philosopher program.

Table 5.1: Analysis Time for Double Start () Detection

Trace size	Execution time	Total time (Execution + Compile Time)
63.6 KB (using fine tuned filters)	4s	27 s
627 KB(using fine tuned filters)	5s	25 s

5.6.1.1 False Positive Detection by Custom Based Detection Approach

This section explores in detail the detection of double `start ()` antipattern by static analysis, and then compares its detection using the custom based detection approach.

Double call of `start ()` method antipattern was detected by static analysis tool Extended-JLin [45].

The message *"another start method call"* is emitted corresponding to the *"double call of the start method of a thread"* antipattern, which is a false positive. The tool detects the antipattern in the following segment of the code. It is signalled for the second `start ()` method call in line [9]. However, this method is not called for the same thread, since the variable *t* received a new thread object in line [8].

```
public void execute() {
[1]     t = new Thread(guestAgent);
[2]     t.start();
[3]     while (agentState != 0) { // stop
[4]     if (agentState==2) { // resume
[5]         agentState = -1;
[6]          if (t!=null)
[7]        t.interrupt();
[8]        t = new Thread(guestAgent);
[9]        t.start();
[10]       }
[11]    //guestAgent.timerMan.execute();
[12]    GuestSystem.pause(1000);
[13]    }
```

Figure 5.12: Sample of Guest Application Code

Based on the information (like methodIdRef, ObjectIdRef) provided by the events methodEntry and threadStart, the false positive given by static analysis for "double call of the `start ()`" method antipattern is detected. The events of the execution trace, which were used for detection, are shown in Figure 5.13.

```
<methodDef name="start" signature="()V" methodId="302"
classIdRef="325"/>
"First methodEntry for start method"
<methodEntry threadIdRef="2" time="1074266683.224861900"
methodIdRef="302" objIdRef="7532" classIdRef="325"
ticket="11188" stackDepth="3"/>
```

```
<methodExit threadIdRef="2" methodIdRef="302"
objIdRef="7532" classIdRef="325" ticket="11188"
time="1074266683.225230500" overhead="0.000021641"/>

"Second methodEntry for start method"

<methodEntry threadIdRef="2" time="1074266683.833227200"
methodIdRef="302" objIdRef="7679" classIdRef="325"
ticket="12388" stackDepth="3"/>

<methodExit threadIdRef="2" methodIdRef="302"
objIdRef="7679" classIdRef="325" ticket="12388"
time="1074266683.833581400" overhead="0.000018102"/>

"ThreadStart event for first methodEntry"

<threadStart threadId="5" time="1074266683.298559400"
threadName="Thread-0" groupName="main" parentName="system"
objIdRef="7532"/>

"ThreadStart event for second methodEntry"

<threadStart threadId="6" time="1074266683.981886600"
threadName="Thread-1" groupName="main" parentName="system"
objIdRef="7679"/>
```

Figure 5.13: Sample of Execution Trace Required for Double Start () Detection

From the XML element <methodDef> the methodId = "302" corresponding to the start method is obtained. The methodEntry event and its attribute objIdRef corresponding to the methodIdRef = "302" (start) are located. The objIdRef's values of the corresponding methodEntry event are compared and found to be different. These objIdRef's values also refer to different threads. We conclude from this analysis that the "double call of the start () method" is not on the same thread. Thus the runtime analysis identified the false positive given by the static analysis.

5.6.2 Antipattern: Premature Call of Join () Method

Description: A call to the join () method of a thread is premature if the thread has not been started at the time of the call [22].

Application 1 is a custom race program [41].

Application 2 is a custom dining philosopher program.

Table 5.2: Analysis Time for Premature Join () Detection

Trace size	Execution time	Total time (Execution + Compile Time)
29.3 KB (fine tuned filters)	4s (approx)	26 seconds
627 KB(using fine tuned filters)	12s (approx)	45 seconds

A Snapshot of the console output for "premature `join ()`" detection based on the custom-based detection approach is shown in Appendix A.

5.6.3 Antipattern: Wait () Stall

Description: The thread should not wait (after calling the wait method) for more than user specified threshold [22].

Detection: Similarly for "double `start ()`" detection in the execution trace, the method Id "109" for the `wait ()` method is obtained from the XML element <`methodDef`>. The `methodEntrys` and `methodExits` of `wait ()` method for a particular thread (referred by `threadId`) are located. The time difference between `methodEntry` and its `methodExit` is calculated and compared with the user specified time period (in this detector it is set as 0.5 seconds). If the calculated time difference is more than the user specified period then the message "`wait ()` stall" is detected is printed.

This antipattern can only be detected by runtime analysis, because of the timestamp information provided by the `methodEntry`, `methodExit`, and `threadstart` events in the trace.

The relevant events required for the "`wait ()` stall" detection are given below:

```
"methodDef" element for wait method

<methodDef name="wait" signature="()V"
startLineNumber="429" endLineNumber="430" methodId="109"
classIdRef="116"/>

"First methodEntry"

<methodEntry threadIdRef="5" time="1074463868.267297000"
methodIdRef="109" objIdRef="5983" classIdRef="116"
ticket="263" stackDepth="4"/>

"First methodExit"

<methodExit threadIdRef="5" methodIdRef="109"
objIdRef="5983" classIdRef="116" ticket="263"
time="1074463869.072965900" overhead="0.000015904"/>
```

Figure 5.14: Sample of Execution Trace Required for Wait Stall Detection

5.7 Advantages and Limitation of Custom Based Detection

Advantages of Custom Based Detection

1. Can detect the false positive given by static analysis, particularly in the detection of the antipattern "double call of `start ()` method".
2. Scalable (i.e. can be used to analyze execution traces of large size applications). However, due to limitations of the XML marshalling tool, the custom detector generates a memory exception when the trace size exceeds 25MB.

Limitation of Custom Based Detection

Custom based detection provides less coverage of a program compared to heavyweight formal approaches such as model checker and theorem provers, because only a single run of the program is analyzed. Thus it cannot prove the correctness of the system as a whole.

5.8 Conclusion

A custom based detection runtime analysis tool was proposed and implemented for detecting three MT antipatterns. This tool gave a significant edge over static analysis, particularly in the detection of the double call of start () method; it could detect the false positive generated by static analysis.

Chapter 6. Model Checker Based Antipattern Detection

6.1 Introduction

In the previous chapters, we discussed a semiformal approach: custom-based detection based on runtime analysis of Java multithreaded applications. In this chapter we will suggest another technique for runtime analysis based on model checking as backend and we will describe this technique and its experimental results. In this approach, the PROMELA model is extracted from the trace (obtained in XML format) of the Java program. This execution trace contains the relevant events. The PROMELA is input language for Spin model checker. The XML to PROMELA translation is done by a Java program based on a translation schema, (see Section 6.3 for details). The extracted PROMELA model is verified against the MT antipatterns using Spin model checker. These antipatterns were formally specified in LTL (Linear Temporal Logic). We choose Spin in this approach for antipattern verification because it is one of the most popular, mature and advanced open-source model checkers. The Spin model checker can automatically determine whether a program satisfies the LTL property, and in case the property does not hold true, a warning is printed.

The motivation of this approach can be summarized as:

1) *Incorporate Formal Techniques*: To develop the model checker based detection approach, which incorporates formal techniques such as model checking for antipattern detection in runtime analysis. As far as we know no work has been done similar to ours in industry or universities. The closest work is Java Pathfinder; a tool developed at NASA Ames Research centre [37] which

combines runtime analysis with model checking. In this tool, the warnings emitted from the runtime analysis are used to guide a model checker [25].

2) *Benefits of Formal Technique*: To evaluate the benefits of formal technique such as Spin model checker in runtime analysis, comparison is made between a formal approach - model checker based detection and custom based detection – the semiformal approach. The comparison is made with respect to characteristics such as:

    - Quality of analysis,
    - Time usage,
    - Resource consumption etc.

3) *Predictive Trace Analysis*: The model checker based approach could possibly be used for the predictive trace analysis, as the model checker could analyze various possible event interleavings.

4) *State Explosion Problem*: Overcoming the state explosion problem of full blown model checking approach. To overcome this limitation, we model check the trace; an execution trace is an abstract representation of the target application. In certain aspects our approach is comparable to abstract model checking.

6.2 Workflow

The workflow of our approach is shown in Figure 6.1. There are two inputs to it: the Java program in byte-code format to be monitored (created using a standard Java compiler) and the properties/antipatterns to be verified. The output is a (possibly empty) set of property violations printed on a console output.

The model checker based approach can be divided into three main steps:

- Instrumentation Step,
- Model Extraction Step,
- Analysis Step.

6.2.1 Instrumentation Step

The program to be analyzed for antipatterns should be instrumented in such a way that when this instrumented program is executed the relevant events are generated for further analysis.

Instrumentation is performed in a program to be analyzed based on the user specified instrumentation specification; the instrumentation specification contains information to be monitored such as classes, methods or instance variables. Based on the information provided, the empty methods such as `lockAcquire`, `lockRelease`, `variable_write` and `variable_read` are inserted in the target classfile. The instrumentation is performed at bytecode level, and for this JTrek, a bytecode engineering tool from Digital [12] is used. This tool reads the Java class files (bytecode), traverses them as abstract syntax tree while examining their contents, and inserts new codes in highly a flexible manner [27]. This instrumented classfile when executed, on the Hyades platform will emit relevant events as XML fragments, which are collected as trace. This execution trace is given as input for model extraction.

6.2.2 Model Extraction Step

The trace is given as input and then the PROMELA model is extracted based on an input translation schema. The extracted PROMELA model is then given as input for antipattern analysis.

6.2.3 Analysis Step

Here MT antipattern verification is performed and property violations (if any) are printed to the console output. The extracted PROMELA model obtained from the previous step is given as input for antipatterns verification. These antipatterns are specified in Linear Temporal Logic (LTL), and this LTL formula translates to never claim. During model checking, it checks for never claim negation in the extracted PROMELA model, and if such a never claim negation is found, it prints the warnings to the console output. Along with these warnings, it prints other verification details, such as depth reached in the model, number of transitions covered, number of matched states, and verification time. A sample of the snapshot of verification output is shown in Appendix B.

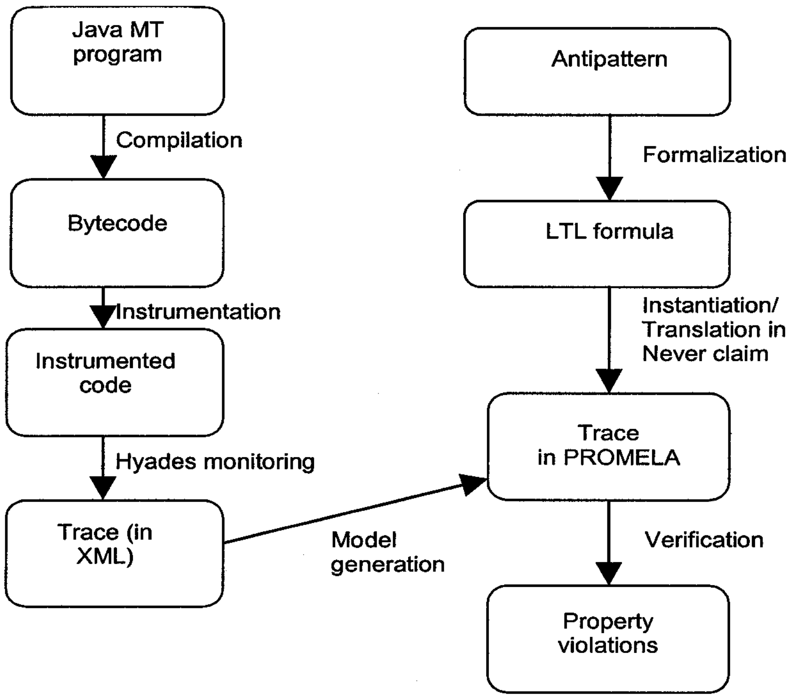


Figure 6.1: Workflow of Model Checker Based Approach

6.3 XML to PROMELA Translation

Each thread is modelled by a PROMELA process. The trace events themselves are translated in a more or less direct way, where each event attribute is modeled by a PROMELA variable. For few instructions, `join ()` and `start ()` we modeled them following their Java semantics [55]. For other thread related Java constructs, a distributed trace approach is followed that assumes that only events of same thread (process) or involved in a communication are ordered [23] [36]. Since threads are controlled with locks we assume that events on the same lock are ordered. Currently, data values and communication via threads are not modeled, since they are not needed for antipattern detection. Note that in Java, data based communication is guarantied to occur if appropriate synchronization constructs are used, otherwise a change of a variable value by one thread may never become visible to other threads [55]. Here the event's attributes and its mapping to the PROMELA model is explained:

1. *Variable/DateType Declaration*: Events attributes such as Reference to Object Identifier (`ObjectIdRef`), Reference to Class Identifier (`ClassIdRef`), and Reference to Method Identifier (`MethodIdRef`) are declared as integer data type in PROMELA.
2. *Process Declaration*: Each thread in execution trace translates to active process in PROMELA. For example, a trace that consists of three threads, namely *thread2, thread5 and thread6,* translates to a PROMELA model that consists of three active processes, namely *process2, process5 and process6,* respectively.
3. *Relevant Events*: Currently `start (), join () wait (), notify (), notifyall (), LockAcquire (), LockRelease ()` method entry and exit and data access events are considered as relevant events of the trace. Other events are not needed for verification itself, though they may be helpful to locate the problem once detected. The model extraction is explained in Section 6.5.
4. *Event Body Translation*: Each relevant event in XML trace translates to `d_step` construct in PROMELA and each event's attribute translates to a variable assignment statement inside the `d_step` construct. The `d_step` insures that each event is atomic and instant.
5. *TimeStamp*: The events in the PROMELA model are assigned the logical timeStamp value, rather than real time value as in the execution trace.

6. *Event Synchronization*: `Start ()` methodentry and corresponding `run` methodentry events are ordered. Events on the lock, related to `lock entry, exit, wait, and notify` are totally ordered.

6.3.1 XML to PROMELA Translation Illustrated with Example

The general idea regarding translation is explained here. The XML trace generally consists of set of tags and declarations and these tags provide information about the data and its relation to other data tags of the events logged. For example, methodEntry event logged (in XML format) as:

`<methodEntry threadIdRef="7" time="1093561048.957960600" methodIdRef="103" objIdRef="8605" classIdRef="120" ticket="1162" stackDepth="5"/>`.

The attributes of this event are; `threadIdRef, time, methodIdRef, objIdRef, classIdRef, tickets` and `stackDepth` etc. The data type of this event's attribute can be either strings or integers.

The XML trace translated to PROMELA model based on a translation schema is shown in Figure 6.2. The translation schema is explained as follows:

1. Initially before the start of event body in PROMELA model, a comment statement is written indicating the event type, threadId and methodId. For example, a comment statement is written as /* wait methodentry in threadId 6 for methodId 113 */.
2. After the comment statement, name of the event type is written as "`name = wait_methodEntry`", and the `methodIdref "113"` refers to `wait ()` methodentry.

3. Other attributes of the events in the XML trace such as `threadIdRef = 6, classIdRef = 116, objIdRef = 8606, methodIdRef = 113` are copied from the XML trace and mapped one-to-one to the PROMELA model.
4. In the XML trace, the event's timestamp is assigned the absolute value such as `time = "1091230513.794350600"` but in the PROMELA model, the event's timestamp is assigned logical value such as `"time = 7"`, which signifies that this particular event happened seventh (7th) in the sequence.
5. The event's attributes in the trace such as; `methodIdRef, objIdRef, classIdRef and threadIdRed`, which are assigned `String` datatype in the XML trace are assigned `int` datatype in the PROMELA model as shown in the Figure 6.3.
6. The event's names such as notify `methodentry/exit, wait methodentry/exit, notifyAll methodentry/exit` are assigned `mtype` datatype in PROMELA model as shown in the Figure 6.3.
7. The event's attributes such as `ticket, stackDepth` which are not relevant from analysis viewpoint are not translated to PROMELA model.

```
     XML Trace                           PROMELA Translation
<methodEntry                                    d_step
threadIdRef="6"                             {
time="1091230513.794350600"             /* first wait methodentry in
                                        threadId6 for methodId 113*/
methodIdRef="113"                           name = wait_methodentry;
objIdRef="8606"                             threadIdRef = 6;
classIdRef="116"                            methodIdRef =  113;
ticket="1074"                               objIdRef = 8606;
stackDepth="4"                              classIdRef =  116;
/>                                          time = 7; (logical time)
                                            }
```

Figure 6.2: XML to PROMELA Translation

```
#define N    120 /* nr of rendevous channels */
#define L    10  /* size of buffer */
mtype =  { methodDef, Notify_MethodEntry,Wait_MethodEntry,
threadstart,Wait_MethodExit,
Notify_MethodExit,lockAcquire_MethodEntry,
lockRelease_MethodEntry,lockAcquire_MethodExit,
lockRelease_MethodExit, Start_MethodEntry,
Start_MethodExit, Join_MethodEntry, Join_MethodExit,
Run_MethodEntry, Run_MethodExit, NotifyAll_MethodEntry,
NotifyAll_MethodExit };


mtype = {message};
mtype   Name;
int  methodId,
MethodIdRef,ClassIdRef,ThreadIdRef,ObjectIdRef,TimeStamp;
chan Q[N] = [L] of {mtype};
```

Figure 6.3: Spin: Some Declarations of PROMELA Model

6.4 Modeling Synchronization

A race condition between two (or more) threads occurs when they modify a member variable of an object simultaneously. Races could lead to data corruption and other various problems.

To avoid data races, a programmer can force fragments of code running on different threads to execute in a certain order by adding synchronization operations. Java offers several constructs that enforce synchronization:

- `start` and `join` which operate on Thread objects,
- locked objects (synchronized blocks and methods),
- `wait` and `notify(All)`.

With `join()` it is feasible to model semantics of these Java constructs very closely, predicting new executions rather then only possible linearization of partial order. In the case of `wait` and `notify`, which provides value driven controls over thread executions, attempts to mimic construct following their Java meaning, will likely result in imprecise model, at least with our level of trace detail. Thus, when it concerns operations on locks, we just enforce order on events that relates to the same lock or thread. This approach is detailed below.

6.4.1 Synchronization in Java

Three main types of MT synchronization events are modeled in PROMELA:

1. Thread `start ()` (StartEntry) and `run` methodentry (RunEntry) (the former mostly precedes the latter).
2. Thread termination (RunExit) and thread JoinEntry and Exit.
3. The events on the same thread are modeled by totally ordered events of a PROMELA process. If the immediately preceding events happen on the same object but on different threads the order is enforced. To enforce this ordering in PROMELA a message is exchanged between events such as `wait, notify, notifyAll entry/exit` and `lock entry/exit`.

6.4.2 Modeling Synchronization in PROMELA

The synchronization is implemented in PROMELA model, using two approaches namely "variable/flag" or "message passing" based approach. In some PROMELA models both approaches are combined.

6.4.2.1 Message Passing Approach

6.4.2.1.1 Modeling Synchronization with Message Passing

Message based approach is used to enforce order for `wait-notify (All)`, `lockacquire/lockrelease` and `start-run` (Entry) events. When a thread invokes wait on an object, the execution of the thread is halted until another thread executes `notify` or `notifyAll` on that very same object. However, a thread is only allowed to invoke `wait` or `notify` on an object if that thread owns the lock of that object.

An example of synchronization based on the message based approach is shown in the Figure 6.4. In this example, a message is exchanged between events if they happen on the same objects (`objectIdRef`) but on the different threads (`threadIdRef`). As shown in the Figure 6.4, an event `Start_MethodEntry` happens on the `threadId` 2 and `objectId` 5317 and the consequent event `Run_Methodentry` happens on the `threadId 5` and `objectId 5317`.

To model this synchronization, a message (`Q [1] !` message) is sent from the `Start_methodentry` event and received at the `run_methodentry` event (`Q [1]` ? message).

```
/* Starting of the process 2*/
active proctype thread2()
{

d_step
{
/* Message is send to another object on different thread
(at methodEntry event, at start) */
Q[1]!message->
/* Start MethodEntry for methodId 301 */

Name = Start_MethodEntry;
ThreadIdRef =2;
TimeStamp = 1;
MethodIdRef = 301;
ObjectIdRef = 5317;
ClassIdRef = 324;
}

}/* End of process 2*/


/* Starting of the process 5*/
active proctype thread5()
{

d_step

{

/* Message is received from another event on different
thread (at MethodEntry event, at start) */
Q[1]?message->


/* Run MethodEntry for methodId 311 */

Name = Run_MethodEntry;
ThreadIdRef =5;
TimeStamp = 3;
MethodIdRef = 311;
ObjectIdRef = 5317;
ClassIdRef = 324;
}

}/* End of process 5*/
```

Figure 6.4: Sample of PROMELA Model, Synchronization Based on the Message Based Approach

6.4.2.1.2 Advantages and Disadvantages of Message Based Approach

Message based synchronization approach is used in our early research prototypes, since it is easy to visualize message exchange with MSC (Message Sequence Chart) in Spin.

However this approach is limited in scalability due to Spin limitations on number of channels, which can be declared (approximately 255). Because of this limitation, message based synchronization approach is replaced with variable based synchronization approach.

A sample of the MSC generated based on message based approach is shown in Appendix C.

6.4.2.2 Variable Based Approach

6.4.2.2.1 Modeling Synchronization Using Variable Based Approach

A sample of the PROMELA model generated using the variable based approach to model behaviour of the `join` method is shown in Figure 6.5. The global Boolean variable `ActiveThread`$_i$, where *i* is an integer of value 2, 5, 6... is a thread identifier; this is initially declared false. When the thread is started (i.e. `RunEntry` event) this variable is set to true ("`ActiveThread`$_i$ `= true`"). Similarly when the thread terminates (i.e Run Exit event) this variable is set to false (`ActiveThread`$_i$ `= false`). To enforce order between the `RunExit` and `JoinExit`, where the latter should happen before the former and also on the same object but different threads, `JoinExit` event is executed only when this condition satisfies ("`::ActiveThread`$_i$ `= = false ->`").

```
bool  Activethread2 = false;
bool  Activethread5 = false;
bool  Activethread6 = false;

/* Starting of the process 5*/
active proctype thread5()
{
d_step
{
/* Run MethodExit for methodId 285 */
Activethread5 = false;
Name = Run_MethodExit;
ThreadIdRef =5;
TimeStamp = 6;
MethodIdRef = 285;
ObjectIdRef = 4379;
ClassIdRef = 298;
}
}/* End of process 5*/
/* Starting of the process 6*/
active proctype thread6()
{
:: (Activethread5 = = false) ->
d_step
{
/* Join MethodExit for methodId 283 */
Name = Join_MethodExit;
ThreadIdRef =6;
TimeStamp = 9;
MethodIdRef = 283;
ObjectIdRef = 4379;
ClassIdRef = 298;
}
}/* End of process 6*/
```

Figure 6.5: Sample of PROMELA Model, Start()/Join() Synchronization Based on the Variable Based Approach

A sample of the PROMELA model generated to model synchronization between events `lockacquire` and `waitentry` and based on the "variable based" approach is shown in the Figure 6.6. Initially the global variable `Q` is declared as `bit Q [N]`, where `N` is the array index, which indicates the synchronization order. The wait is executed (i.e. `waitentry` event) and the variable `Q` is set equal to 1 ("`Q [N] = 1`"). When the corresponding lock is acquired (i.e. `lockAcquire` event) before notify-entry the variable `Q` is compared to one (1) and if found equal, its value is set equal to zero (0) such as ("`(Q [N] = = 1) -> Q [N] = 0`"). The events (`lockAcquire` and `waitentry`) should happen on the same object but on different threads. Also the latter should happen before the former. Similarly, the variable based approach is used to model synchronization between other events such as `lockrelease`, `notify entry/exit`, `notifyAll entry/exit`, `waitexit`, `start entry` and `run entry`.

```
bit Q[N];

d_step
{
/* The variable is set at receiving end, same objectId
different threadId (at MethodEntry event) */
(Q[21]== 1)->Q[21] = 0;


/* LockAcquire MethodEntry for methodId 107 */

Name = lockAcquire_MethodEntry;
ThreadIdRef =5;
TimeStamp = 90;
MethodIdRef = 107;
ObjectIdRef = 8562;
ClassIdRef = 120;
}

d_step
{
/* The variable is set, case of different objectIds and
same threadIds (at methodEntry event)*/
Q[21]= 1;


/* Wait MethodEntry for methodId 111 */

Name = Wait_MethodEntry;
ThreadIdRef =8;
TimeStamp = 79;
MethodIdRef = 111;
ObjectIdRef = 8562;
ClassIdRef = 120;
}
```

Figure 6.6: Sample of PROMELA Model, Wait ()/Notify() Synchronization Based on the Variable Based Approach

6.4.2.2.2 Advantages and Disadvantages of the Variable Based Approach

The variable based approach is particularly advantageous over the message based approach for modeling large traces. In the message based approach, messages are exchanged between events using channels. In the extracted PROMELA model, an array of channels of size L is declared, the size of L is limited (it is 255). The message based approach fails to model large traces in which the size of channel array required exceeds 255.

On the other hand, the "variable based" approach has a disadvantage over the "message passing based" approach: the MSC (Message Sequence Chart) cannot be obtained in former.

6.5 XML to PROMELA Translation Algorithm and its Implementation

6.5.1 Translation Algorithm

The XML to PROMELA translation algorithm is implemented in Java. The translation algorithm where the synchronization is based on the message based approach is explained below. Please refer to the flowchart in Appendix E for more details.

1. The XML trace is parsed and objectId's and threadId's of relevant events e.g. `waitentry/exit`, `notifyentry/exit`, `lockAcquire entry/exit`, `lockrelease entry/exit` `startentry`, `runentry`, `runexit` and `joinexit` are saved in `objectId []` and `threadId[]` arrays respectively. Then their event types, such as

lockoperation_event, startentry_event, runentry_event and other_event are saved in event_type [] array.

2. The events obtained from the previous step are categorized as follows:
    - The events wait entry/exit, notify entry/exit, lockAcquire entry/exit and lockrelease entry/exit, and are categorized as lockoperation_event.
    - The events runexit and joinexit are categorized, as other_event. The events startentry and runentry, are categorized as startentry and runentry.
3. The next iteration is performed in objectId[], threadId[] and event_type[] arrays. At the same time a search is performed for those objectIds and threadIds in the arrays whose preceding objectIds and events are the same but are on different threadIds. In the start and run synchronization, the event start methodentry should be preceded by a run methodentry event. If such instances are found, the indexes of the objectIds are saved in send_message [] array. The indexes of those objectIds from which they are different are saved in receive_message [] array. These arrays are created to number send and receive messages, which are used to model synchronization in message passing based approach.
4. The events of the XML trace are not written to the PROMELA model in the same order as they are read from the XML trace, but in a different order. For example, all the relevant events of threadId 4 are written first, and then all the relevant events of threadId 5 are written second and so on. Because of the way the model

is generated, the send and receive messages are not numbered according to the order obtained from the `send_message []` and `receive_message[]` arrays.

5. The next iteration is performed through the `objectId []`, `threadId []` and `event type []` arrays in such a manner that the `objectIds` in `objectId []` array is iterated in order corresponding to their `threadId` 2, 3, 4.. .While iterating in this manner, a search is performed for those instances whose consecutive `objectIds` are the same, but corresponding `threadIds` are different. Whenever such instances are found its, index key is saved in the integer *i* and then its relative position *g* is found in the `send_message []` array using the method "`int g = Arrays.binarysearch (send_message, i)`". The send message - `Q[g] !` is then numbered accordingly.
6. The same procedure as just detailed is followed to number the receive messages - `Q[g]?`.
7. The XML trace is iterated again and the `threadId's` of the `<threadStart methodentry>` event are saved in the `threadIdStart []` array.
8. The events from the XML trace are written to the PROMELA model as detailed in step 2. The information from the `threadIdStart []` array obtained in step 5 is used, while writing events to the PROMELA model.
9. While events are written to the PROMELA model, synchronization is enforced based on the message based approach, i.e. send and receive messages are numbered and inserted. The send and receive messages are numbered according to algorithm described in step 2 & 3. Before or after writing any event body to the

PROMELA model, a check is performed to see, whether a send or receive message is required to be inserted and if required, the send and receive messages are numbered and inserted.

6.5.2 Java Implementation of the Algorithm

The XML to PROMELA translation algorithm as implemented in Java consists of six classes namely logical_timestamp, thread_count, list_entry_exit, receive_send message, method_def and main classes. Here the functionality of these classes is explained.

1. ***Logical_timestamp***: This class assigns a logical timestamp to events in the PROMELA model. The timestamp of events in the XML trace are assigned absolute value, `time="1091230513794350600"`. The absolute timestamp is mapped to a logical timestamp in the corresponding translated event in the extracted PROMELA model. The mapped logical timestamp indicates the relative occurrence of the event.
2. ***Thread_count***: This class outputs the number of threads in an execution trace. To get the thread count, the trace is parsed and the thread count is assigned a value based on the number of `<thread_start>` entries.
3. ***List_entry_exit***: This class outputs the `objectId []` and `threadId []` arrays. The trace is parsed, and objectIds and threadIds of the relevant events are saved in the `objectId []` and `threadId []` arrays, respectively.
4. ***Receive_send Message***: This class outputs the `receive_message[]` and `send_message[]` arrays. The `objectId []` and `threadId []` arrays, obtained from previous class are parsed. The index values of those `objectIds`, which are the same but happen on different `threadIds`, are saved in the

`send_message []` array. Similarly the index values of those `objectIds` from which they are different are saved in the `receive_message []` array.

5. ***Main Class***: This class translates the XML trace to the PROMELA model and writes this model to an external .doc file. While writing these events to the PROMELA model, the send and receive messages are numbered and inserted. As discussed before, events are not written in the same order as they are read from the XML trace, but in a different order. For example, all the relevant events of the `threadId` 4 are written, and then the relevant events for the `threadId` 5 are written and so on. Before or after writing each event body to the PROMELA model, it is checked if the send or receive message is required to be inserted or not. If required then the send and receive messages are numbered and inserted.

6.6 Property Specification in PROMELA with LTL

Before discussing the MT antipattern specification in LTL, a brief introduction on LTL is given below.

6.6.1 LTL Overview

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. Pnueli suggested using Linear-Time Propositional Temporal Logic (LTL) for reasoning about concurrent programs. Since then, several researchers have used LTL to state and prove correctness of concurrent programs, protocols, and hardware.

Linear-Time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the propositional logic operators (and (&&), or (||), xor (^), not (!), etc.) there are future-time operators. The following syntax is used in our tool for these four operators [32].

- Always in the future. One can use the English keywords `Always`, or a box ([]) to represent the always operator.
- Sometime in the future. One can use the English keywords `Sometime` or a diamond (<>) to represent a sometime operator.
- Until. One can use the English keywords `Until`, or U to represent the until and since operator.
- Next iteration. One can use the English keywords `Next`, or a cross (X) to represent these operators.

6.6.2 Property Specification

Here we consider formalization of premature `join ()` antipattern in LTL. It consists in the invocation of the `join` method to the thread, which is not yet started [22]. Obviously it is impossible to specify such an antipattern in LTL independently of the number of threads. Consider the instantiation of antipattern for one particular thread T_i (i is an integer 1,2,3..), where `join` to thread T_i is called before the start of T_i. Actually, it is more convenient to formalize the absence of an antipattern, so a model checker can pinpoint the problem with a counterexample to the correctness claim. Obviously, the formalization of antipatterns requires predicates, which indicate invocation of the `join` method; Join (T_i) and thread start, Start (T_i). To formalize the absence of a "premature `join`" to the thread (T_i), a pattern specification system [54] could be used. The most adequate pattern is precedence: S = Start (T_i), precedes P = Join (T_i), which is mapped onto LTL, as! P W S = ! Start (T_i) W Join (T_i), where W is the weak until operator. In a trace that consists of n threads $T_1,\ldots, T_n$ instantiations of antipatterns for each thread could be either checked one by one, for each thread T_i, or at once with all combined in one composite property;

! Start (T_1) W Join (T_1) & ! Start (T_2) W Join (T_2) &...& !Start (T_n) W Join (T_n)

The second approach was followed, which is more convenient, while the first one provides better diagnosis. It is immediately clear, which exact thread is involved in premature `join`. Instantiation of predicates Start(T_i) and Join(T_i) is implementation dependent.

Similarly, formalization of double `start ()` antipattern in LTL is considered. It is informally defined as "the `start ()` method is not supposed to be started more than once for the same thread". Actually, its absence is formalized so that model checker could pinpoint the problem with a counterexample to the correctness claim. The formalization of this antipattern requires the definition of two predicates P and Q, defined as P = Start (Ti) and Q = start (Ti). In terms of these predicates, the double `start` absence is defined as "absence of P after Q". This claim is specified in LTL as `[] (Q -> X [] (! P))`. This means that [] (always) in a thread Ti, predicate Q should [] (always) be preceded (X operator) by the absence of predicate P (! P).

In a trace that consists of multiple threads $T1$, $T2$.....Tn, the absence of the double `start` should be checked on each thread separately.

An example of double `start` antipattern specified as LTL formula for three threads follows: "*[] (a -> X [] (! a)) && [] (b -> X [] (! b)) && [] (c -> X [] (! c))*", where the a, b and c are predicates, which are defined as follows in PROMELA:

```
#define a        (MethodIdRef == 301 && ObjectIdRef == 5317
&& ThreadIdRef == 2)
#define b        (MethodIdRef == 301 && ObjectIdRef == 5317
&& ThreadIdRef == 5)
#define c        (MethodIdRef == 301 && ObjectIdRef == 5317
&& ThreadIdRef == 6)
```

This LTL formula translated to never claim negation as:

```
    /*
     * Formula As Typed: []  (a  ->  X []  ( ! a))   &&  []
(b  -> X []  ( ! b)) &&  []  (c  -> X []  ( ! c))
     * The Never Claim Below Corresponds
     * To The Negated Formula !([]  (a  ->  X []  ( ! a))
&&  []  (b  -> X []  ( ! b)) &&  []  (c  -> X []  ( ! c)))
     * (formalizing violations of the original)
     */

never {    /* !([]  (a  ->  X []  ( ! a))   &&  []  (b  ->
X []  ( ! b)) &&  []  (c  -> X []  ( ! c))) */
T0_init:
    if
    :: ((c)  && (((b))  ||  ((c)))  && (((a))  ||  ((((b))  ||
((c))))))) -> goto accept_S5
    :: ((b)  && (((b))  ||  ((c)))  && (((a))  ||  ((((b))  ||
((c))))))) -> goto accept_S9
    :: ((a)  && (((a))  ||  ((((b))  ||  ((c)))))))  -> goto
accept_S13
    :: (1) -> goto T0_init
    fi;
accept_S5:
    if
    :: ((c)) -> goto accept_all
    :: (1) -> goto T0_S5
    fi;
accept_S9:
    if
    :: ((b)) -> goto accept_all
    :: (1) -> goto T0_S9
    fi;
accept_S13:
    if
    :: ((a)) -> goto accept_all
    :: (1) -> goto T0_S13
    fi;
T0_S5:
    if
    :: ((c)) -> goto accept_all
    :: (1) -> goto T0_S5
    fi;
T0_S9:
    if
    :: ((b)) -> goto accept_all
    :: (1) -> goto T0_S9
    fi;
T0_S13:
    if
    :: ((a)) -> goto accept_all
    :: (1) -> goto T0_S13
    fi;
accept_all:
    skip
}
```

To verify the LTL in a trace which consists of *N* threads (*T*1, *T*2, *T*3....*Tn*), this LTL formula is extended accordingly for *N* number of Threads.

6.7 Experiments

6.7.1 Steps of Verification

The verification of MT antipatterns using the model checker based approach is performed in the following steps.

1. ***Never Claim and Source Code Generation***: The command prompt instruction "`spin - a -N c:\double_start.ltl c:\double_start`" generates never claim negation from the LTL formula and source code. This is then compiled and run to perform verification. The `-a` option generates source code that can be compiled and run to perform various types of verification of a PROMELA model. The output is written as a set of C files named pan. [cbhmt], that must be compiled to

produce an executable. The `-N` option allows the user to specify a never claim file that the Spin parser will include as part of the model.

2. ***Compilation***: Before verification, the executable is generated from the compiled C program (pan.c). The compilation command "`gcc -DVECTORSZ = 2048 -o pan.exe pan.c -Ic:\include -L c:\lib`" generates an executable (pan.exe). This compilation instruction is executed with "`-DVECTORSZ = 2048`", which is a command-line argument. The default size for the state vector is 1024 bytes, which is usually insufficient for large model, will prompt for the recompilation with a higher value. In this verification, the default size is specified as `-DVECTORSZ = 2048`. The include instruction "`-Ic:\include`" will include the required header files (with .h extension).
3. ***Verification***: After compilation, the executable (pan.exe) is executed by command option "`pan - a`", which performs the verification. When the verification terminates, the verification result appears as shown in Appendix B. The top line says: "property violated", and further down it states: "errors: 1". If there are no errors then "errors: 0" is printed.

The message sequence chart (as shown in the Appendix C) is explained here. A message is sent (message! 1) before the occurrence of event at thread6, second is sent (message! 2) before the occurrence of event at thread7 to the corresponding event in thread6. The sent messages are labeled as message! 1, 2, 3, 4 … and receive messages are labeled as message? 1, 2, 3, 4... A message is exchanged between threads if the consecutive events in the PROMELA model happens on the same object (same `ObjectId`) but on the different threads.

6.7.2 Antipatterns Detection

Experiments were performed to detect two antipatterns namely: "double call of `start ()` method" and "premature call of `join ()` method" using model checker based detection approach. The experiments were performed on the following hardware and software configuration:

Hardware Configuration:

1. CPU:	AMD Athlon 900 MHz
2. RAM:	512 Mbytes

Software used:

1. Operating System:	WINDOWS 2000
2. Model Checker:	Spin 4.1
3. Compiler:	gcc (C compiler)
4. Java - XML Tool:	JAXB (Java Architecture for XML Binding)

6.7.2.1 Double Call of Start () Method Detection

Description: The `start ()` method is not supposed to be used more than once for the same thread [22].

Application 1 is a fragment of Java multi-threaded platform Guest [45].

Application 2 is a custom dining philosopher program.

Before verification, the PROMELA model needs to be extracted, Table 6.1 lists the total time (execution + compilation), required to build the model for the double `start ()` detection. Table 6.2 lists the verification and the compilation time.

Table 6.1: Model Extraction Time for Double Start () Detection

Trace size	PROMELA model size	Execution time	Total Time(Execution + Compile Time)
63.6 KB	3.25 KB	9s	34 s
627 KB	64 KB	18s	38 s

Table 6.2: Verification & Compilation Time for Double Start () Detection

PROMELA model size	Pan.c built time	Pan.exe built time	Verification time
3.25 KB	1s	2s	1s
64 KB	1s	3s	2s

Verification Data

State Vector Size:	1592 bytes
Depth Reached:	15
Number of transitions:	8 (stored + matched)
Number of matched states:	0
Number of states stored:	8
Number of errors:	1

6.7.2.2 Premature Call of Join () Method Detection

Description: A call to the `join ()` method of a thread is premature if this thread has not been started at the time of the call [22].

Application 1 is a custom race program [41].

Application 2 is a custom dining philosopher program.

Table 6.3 lists the total time (execution + compilation) required to build the model for the premature `join ()` detection. Table 6.4 lists the verification and the compilation time.

Table 6.3: Model Extraction Time for Premature Join () Detection

Trace size	PROMELA model size	Execution time	Total Time (Execution + Compile Time)
29.3 Kb	2.23 Kb	5s	27s
627 Kb	64 Kb	18s	38 s

Table 6.4: Verification & Compilation Time for Premature Join () Detection

PROMELA model size	Pan.c built time	Pan.exe built time	Verification time
2.23 Kb	1s	2.5s	1s
64 Kb	1s	3s	2s

Verification Data

State Vector Size:	1592 bytes
Depth Reached:	21
Number of transitions:	17 (visited + matched)
Number of matched states:	1
Number of states visited:	16
Number of errors:	1

6.8 Open Problems and Alternatives for Trace Modeling

Here is the list of possible future works or open problems in the model checker based detection approach

6.8.1 Open Problems

1. *Antipattern Specification using Embedded C*: As described before, the generated PROMELA model was verified for MT antipatterns specified in LTL. The LTL formula translates to never claim and the model checker searches the state space for never claim negation. An alternative to this approach is to implement the MT antipattern detectors in C. The Spin version 4.0 or later supports embedded C

inclusion in the PROMELA model through the use of five new primitives. These primitives are `c_expr, c_code, c_decl, c_state, c_track`. Using these primitives the antipatterns coded in C language are embedded at certain locations in the model. During the model verification, these embedded detectors provide guidance to the precise location of errors. Another advantages of using embed C is that it could be used to specify trace independent properties.

2. *Aspect Oriented Paradigm (AOP)*: Aspect oriented paradigm (AOP) an alternative to object-oriented paradigm (OOP), could possibly be used for PROMELA model extraction from trace. AOP based model extraction will be particularly useful when the execution trace is of large size with many interleavings. To model large traces, using AOP the specifications of the model are divided as concerns and then these concerns are coded independently. Aspect J is a popular AOP based tool.
3. Extend the model checker based detection approach to verify other multithreaded antipatterns.
4. The current model checker based detection approach, which is mainly targeted for error detection can be extended to error correction. To extend it, warnings emitted (antipattern violations) by the model checker, can be read and given as feedback to the target program for error correction.

6.8.2 Alternatives

There are few alternatives to trace modelling and analysis, listed here some of them.

1. Possibly use mathematical techniques such as logical induction for model extraction from the execution trace, and then verify the extracted model for the

MT antipatterns using the theorem proving techniques. The advantage of using theorem proving is that this technique is not limited by the state space size.

2. Experiment with other verification engines such as Maude for MT antipattern detection. Maude is a modularized specification and verification system that efficiently implements rewriting logic [46].
3. Causality Based Analysis: In the multithreaded programs, threads communicate via a set of shared variables and these variables are recorded in the execution trace. It is observed that some variable updates can causally depend on others. This causality based analysis is used to predict errors that can potentially occur in a possible run of multithreaded programs.

Chapter 7. Comparison

7.1 Introduction

In this chapter we make a detailed comparison between two approaches, namely model checker based detection and custom based detection. The comparison will be made on the characteristics such as time usage, complexity of analysis, scalability and quality of analysis.

Our motivation to conduct this comparative study is to assess the applicability of formal approaches in runtime analysis for antipattern detection.

7.2 Experimental Results

We used Java detectors to analyze execution traces of large Java programs, such as SAP Vending Machine Server. The size of an available SAP vending machine Server trace is about 80 megabytes. Analyzing such a large trace directly using a custom based detection approach caused memory overflow exception. To overcome the memory overflow problem, we used filters to reduce the trace size. We could possibly use more scalable parsing tools such as SAX parser.

As discussed earlier, the model checker based detection approach is based either on a variable based approach or a message passing based approach or both. Modeling the trace based on message passing approach does not scale to very large execution trace size, because of the certain inherent limitations of the PROMELA language. Because of this reason we completely replaced the message passing based approach with shared variables for scalability purposes.

For comparison we performed experiments on three applications and two antipatterns using both the custom and model checker based detection approaches. The first application is a fragment of Java multi-threaded platform Guest [45]. The second is a toy demo program (borrowed from JProbe); both with injected faults and the third one is SAP vending machine server. The experiments are performed on AMD Athlon 900 MHz system with 500MB of RAM and Windows 2000 operating system.

Table 7.1: Experimental Results

	Antipattern	Trace Size (using filters)	PROMELA Model Size	Custom Analyzer	Model Building and Verification
App1	Double start	64 K	3.25 KB	4 s	13 s
App2	Premature join	29.3 KB	2.3 KB	4 s	10 s
App3	Double start	667 KB	22.0 KB	6 s	20 s

7.3 Evaluation of Main Characteristics of Custom and Model Based Detection

The comparison between these two tools is made based on the characteristics such as time usage, complexity of analysis, scalability and quality of analysis.

Time Consumption

The results from the experiments have shown that custom detectors are faster (approximately 3 times); however most of the time is consumed not by model checking itself, which takes less than a second, but by auxiliary steps, such as building PROMELA model, compiling PROMELA into executable, etc.

Complexity of Analysis

The model checker based detection is more cumbersome and complex than the custom based detection because in the former the PROMELA model of the trace is required to be generated, which adds to complexity whereas in the latter we directly analyze the XML trace. While model checkers are known to suffer from the state explosion problem, we did not experience this problem in our experiments on small and medium size applications.

Scalability

The custom detection scales better than model based detection; however in our case custom detection is limited by XML processing tool JAXB. Message based versions of our model based detection tool do not scale to large execution traces due to the certain limitations of the model checker of choice, Spin. The variable based version scales better at least up to medium size applications.

Quality of Analysis

Regarding the quality of analysis, model checker based detection is rated better than the custom based detection. A custom tool cannot guarantee the trace correctness because it analyzes the order in which events are recorded. Correctness can be assured by model checker based detection because the model checker based detection allows predictive trace analysis, in the sense that it can analyze several event interleavings. Especially, properties in multiprocessor environment can be more accurately verified using model checker detection than custom based detection. In the multiprocessor environment, multiple threads run on the multiple processors. For example, thread1 runs on processor1 and thread2 runs on processor2 and so on and when executed the trace1 and trace2 are obtained. Our model based tool can perform model checking of parallel composition of these traces (trace 1 & 2). Thus the concurrent properties can be verified. One such concurrent antipattern is premature `join ()` call, it is sensitive to the order, in which events are observed. The concurrency in multiprocessor system, instrumentation, or trace collection setup could possibly distort the event order. For example, if thread `joins` prior to its `start`, the corresponding events could be recorded in the opposite order. Thus the premature `join ()` call antipattern will be missed by custom detector.

Properties that describe antipatterns related to Java multithreading, such as double `start ()` call or `wait ()` stall, which are not truly concurrent, and do not depend on the event interleavings, are detected reliably with custom tool.

Resource (memory) consumption

Regarding the resource consumption (memory), custom detection is rated better than model based detection, because to store the model extracted from the execution trace, an extra memory is required; whereas in the custom based detection we directly analyze the trace.

Chapter 8. Conclusion and Future Work

8.1 Conclusion

We discussed several runtime analysis approaches for dynamic antipattern detection in Java multithreaded applications. While some of the antipatterns depend on the thread scheduling and event interleaving, others are independent of the order in which events occur (e.g. double `start`). Antipatterns such as premature `join` () depend only on the thread-local order of events whereas wait stall antipattern is order-independent. To detect these multithreaded (MT) antipatterns, we developed two runtime analysis tools. The first tool is based on a formal verification method - model checking, and the second tool is based on ad-hoc custom analysis. A model checking tool, Spin, could analyze various possible event interleavings, which improves quality of analysis for order-sensitive patterns. The model checking is known to cause state explosion, but this problem is alleviated since we analyze only one particular distributed execution. In custom antipattern detection tool, each antipattern detector is implemented in Java. The custom based detection benefits from formal antipattern specification in FSM and EFSM, which could be used to define antipattern formally. Then the antipattern detectors are coded manually in Java.

In these two tools, the same instrumentation approach (Hyades tool supplemented by the bytecode instrumentation tool – JTrek) is used for the relevant events extraction from the target program. Hyades – a non-intrusive tracer is complemented with bytecode instrumentation, based on JTrek. The supplementary instrumentation is needed because an analysis of MT applications could require a synchronization events collection such as monitor (lock) entry and exit, a task that is not supported by Hyades. This instrumentation approach could be seen as lightweight, since it instruments Java bytecode with empty methods, which do not perform any functionality. The Hyades tracer collects the invocations of these empty methods. Thus this integrated instrumentation approach results in a small overhead to the target program.

To evaluate these two runtime analysis tools we experimented with antipattern detection on three small and medium size applications. Our comparison study concludes that overall the custom tool is faster, consumes fewer resources, and scales better than a model checker based detection tool. Moreover, custom analysis is equally efficient as a model checker based tool for some simple antipatterns detection. However, a model checking tool could deliver higher quality analysis for concurrent antipatterns. A paper, based on these results appeared in the Proceedings of the Workshop of Dynamic Analysis (WODA-05) [8].

8.2 Future Work

The work could be extended in the following directions:

1. Experiments in antipattern detection using our tools on several realistic large cases studies to compare the results with those obtained from other runtime analysis tools such as JMPAX, JPAX and JavaMac.
2. Detection of other multithreaded antipatterns.
3. Implement GUI for both tools.
4. Improve visualization and traceability.
5. Investigate possibilities for combining static analysis with runtime analysis.

References

[1] Aldrich, J., Chambers, C., Gün Sirer, E., and Eggers, S. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In Proceedings of the 6th International Static Analysis Symposium, Springer-Verlag, LNCS 1694, (Venice, Italy, September 1999), 19-38.

[2] An antipattern book http://www.antipatterns.com/thebook.htm

[3] Antipatterns, Portland Pattern Reposiory Wiki, Ward Cunningham, http://c2.com/cgi/wiki/wiki?AntiPattern

[4] Armstrong, E. Hotspot: A new breed of virtual machine, 1998.

[5] Artho, C. Finding faults in multi-threaded programs. Master Thesis, Institute of Computer System1s, Federal Institute of Technology, (Zurich, Austin. 2001).

[6] Artho, C., Biere, A., and Havelund, K. High-Level Data Races. First International Workshop on Verification and Validation of Enterprise Information Systems-(VVEIS'03), (Angers, France, April 22, 2003).

[7] Baur, M. C. Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs. Diploma Thesis, Computer Systems Institute, Swiss Federal Institute of Technology (Zurich, April 9, 2003).

[8] Boroday, S., Petrenko, A., Singh, J., and Hallal, H. Dynamic Analysis of Java Applications for Multithreaded Antipatterns. In Proceedings of the Third International Workshop on Dynamic Analysis (WODA 2005), (St. Louis, Missouri, USA, May 2005).

[9] Brown, W. J., Malveau, R. C., McCormick, H., and Mowbray, T. J. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, NY1998.

[10] Cain, A., Schneider, J. G., Grant, D., and Chen, T.Y. Run-time Data Analysis for Java Programs. 1st Workshop on ASARTI, (Darmstadt, Germany, July 21, 2003).

[11] Choi, J.D, Loginov, A., Sarkar, V. Static Datarace Analysis for Multithreaded Object-Oriented Programs. IBM Research Report. http://www.research.ibm.com/dejavu/rc22146.pdf

[12] Cohen, S. JTrek, Compaq. http://www.cis.upenn.edu/~rtg/mac/

[13] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., Zheng, H. Bandera: Extracting Finite-state Models from Java Source Code. In Proceedings of the 22nd International Conference on Software Engineering, June, 2000.

[14] Dahm, M. Byte Code Engineering. In JIT proceeding, 1999.

[15] Dahm, M. Byte Code Engineering with the BCEL API. Technical Report B-17-98, Institut f¨ur Informatik, Freie Universit¨at (Berlin, April 3, 2001).

[16] Diljkstra, E.W. Hierarchical Ordering of Sequential Processes. In Operating Systems Techniques, Hoare, C.A.R., and Perrott, R.H., Eds., Academic Press, New York, 1972.

[17] Editors of The American Heritage Dictionaries. The American Heritage Dictionary of the English Language, 4th edition, published by Houghton Mifflin Co, January 2000.

[18] ESC. Extended Static Checking for Java. Compaq, 2002.

http://research.compaq.com/SRC/esc/download.html

[19] Filman, R. E., and Havelund, K. Source-Code Instrumentation and Quantification of Events. In Foundations of Aspect-Oriented Languages (FOAL'02), (Enschede, Netherlands, April 22, 2002).

[20] Goldberg, A., and Havelund, K. Instrumentation of Java Bytecode for Runtime Analysis. Fifth ECOOP Workshop on Formal Techniques for Java-like Programs, (Darmstadt, Germany, July 21, 2003).

[21] Hagger, P. Understanding bytecode makes you a better programmer. Developer Work July 2001.

http://www-106.ibm.com/developerworks/ibm/library/it-haggar_bytecode/

[22] Hallal, H., Alikacem, E., Tunney, P., Boroday, S., and Petrenko, A. Antipattern-Based Detection of Deficiencies in Java Multithreaded Software. Fourth International Conference on (QSIC'04), (Braunshweig, Germany, September 08 - 10, 2004).

[23] Hallal, H., Boroday, S., Ulrich, A. and Petrenko, A. An Automata-based Approach to Property Testing in Event Traces. In Proceedings of the International Conference on Testing of Communicating Systems (TestCom 2003), (Sophia Antipolis, France, May 26-29, 2003), 180-196.

[24] Havelund, K. Dynamic Program Analysis. A talk at NASA Ames Research Center, October 2002.

http://ic.arc.nasa.gov/researchinfusion/materials/JPaX/talk.pdf

[25] Havelund, K. Using Runtime Analysis to Guide Model Checking of Java Programs. The 7th International SPIN Workshop, (Stanford, California, September, 2000).

[26] Havelund, K., and Rosu, G. Efficient Monitoring of Safety Properties. In Software Tools for Technology Transfer, 2004.

[27] Havelund, K., and Rosu, G. Monitoring Java Programs with Java PathExplorer. First Workshop on Runtime Verification (RV'01), (Paris, France, 23 July 2001).

[28] Havelund, K., and Rosu, G. Synthesizing Monitors for Safety Properties. International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), (Grenoble, France, April 14, 2002).

[29] Havelund, K., Artho, C., Drusinsky, D., Goldberg, A., Lowry, M., Pasareanu, C., Rosu G., and Visser, W. Experiments with Test Case Generation and Runtime Analysis. 10th International Workshop on Abstract State Machines, (Taormina, Italy, March, 2003).

[30] Havelund, K., Johnson, S., and Rosu, G. Specification and Error Pattern Based Program Monitoring. European Space Agency Workshop on On-Board Autonomy, (Noordwijk, Holland, October 2001).

[31] Hoare, C.A.R. Monitors: An operating system structuring concept. Communications of the ACM, 17(10), October 1974.

[32] Holzmann, G. J. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.

[33] Hovemeyer, D., and Pugh W. Finding Bugs is Easy. JavaOne Sun's 2004 Worldwide Java Developer Conference.

[34] Hyades Project, Eclipse platform http://www.Eclipse.org/hyades/

[35] Java 2 Platform Standard Edition (J2SE) 5.0 ("Tiger"). Sun Microsystems http://java.sun.com/developer/technicalArticles/releases/j2se15

[36] Java MultiPathExplorer. JMPaX 2.0. http://yangtze.cs.uiuc.edu/~ksen/jmpax/web,2005

[37] Java Pathfinder "A Model Checker for Java Programs" http://ase.arc.nasa.gov/visser/jpf/

[38] Java Platform Debugger Architecture from Sun Microsystems. http://java.sun.com/j2se/1.4.1/docs/guide/jpda/

[39] Java Virtual Machine Profiling Architecture. Sun Microsystems http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

[40] Jinsight, IBM AlphaWorks. July 2001 http://www.alphaworks.ibm.com/tech/jinsight

[41] JProbe. A tool by Quest Software http://www.quest.com/jprobe

[42] Kim, M. Information extraction for run-time formal analysis. Ph.D Thesis, CIS Department, University of Pennsylvania, September 2001.

[43] Lewis, B., and Berg, D. J. Multithreaded Programming with Java Technology. Sun Microsystems Press, 2000.

[44] Lindholm, T., and Yellin, F. Java Virtual Machine Specification, Second Edition.

http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[45] Magnin, L., Pham, V.T., Dury, A., Besson, N., and Thiefaine, A. Our guest agents are welcome to your agent platforms. In Proceedings of the ACM Symposium on Applied Computing (SAC 2002), (Madrid, Spain, March 10-14, 2002), 107-114.

[46] Maude System. http://maude.cs.uiuc.edu/

[47] Nelson, S., and Pecheur, C. V&V OF ADVANCED SYSTEMS AT NASA Technical Report for NASA January25, 2004 http://ase.arc.nasa.gov/vvivhm/reports/FinalNASAReport2.pdf

[48] Roetter, A. Writing Multithreaded Java applications. Developer Works. IBM Resource for Developers. February 2001.

http://www-128.ibm.com/developerworks/library/j-thread.html

[49] Runtime Monitoring and Checking. http://www.cis.upenn.edu/~rtg/mac/

[50] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A Dynamic Data Race Detector for Multithreaded Program. ACM Transactions on Computer Systems, ACM Press, NY, 1997.

[51] Sen, K. Predictive Safety Analysis of Concurrent Programs. Master's thesis, University of Illinois at Urbana-Champaign, May 2003

[52] Sen, K., Rosu, G., and Agha, G. Runtime Safety Analysis of Multithreaded Programs. In Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Helsinki, Finland, 2003), 337-346.

[53] Smith, C. U., and Williams, L. G. Software performance antipatterns. In Proceeding of Workshop on Software and Performance, (Ottawa, 2000), 127-136.

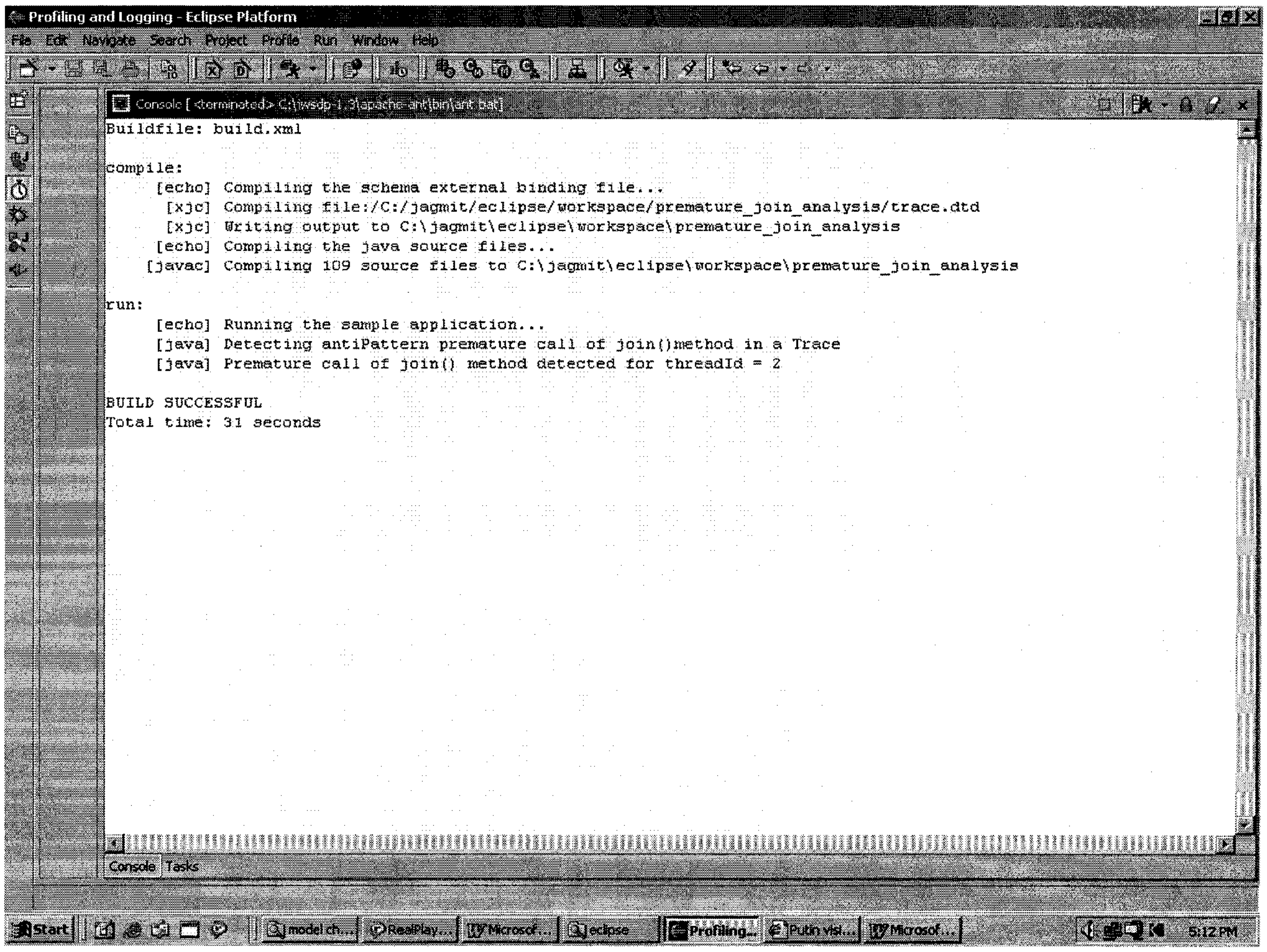
[54] SpecPattern, http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml

[55] Synchronisation in Java, 2001 http://www.usenix.org/publications/library/proceedings/jvm01/full_papers/christiaens /chr-istiaens_html/node3.html

[56] Verisoft "A tool for systematic software testing by Bell Laboratories" http://cm.bell-labs.com/who/god/verisoft/

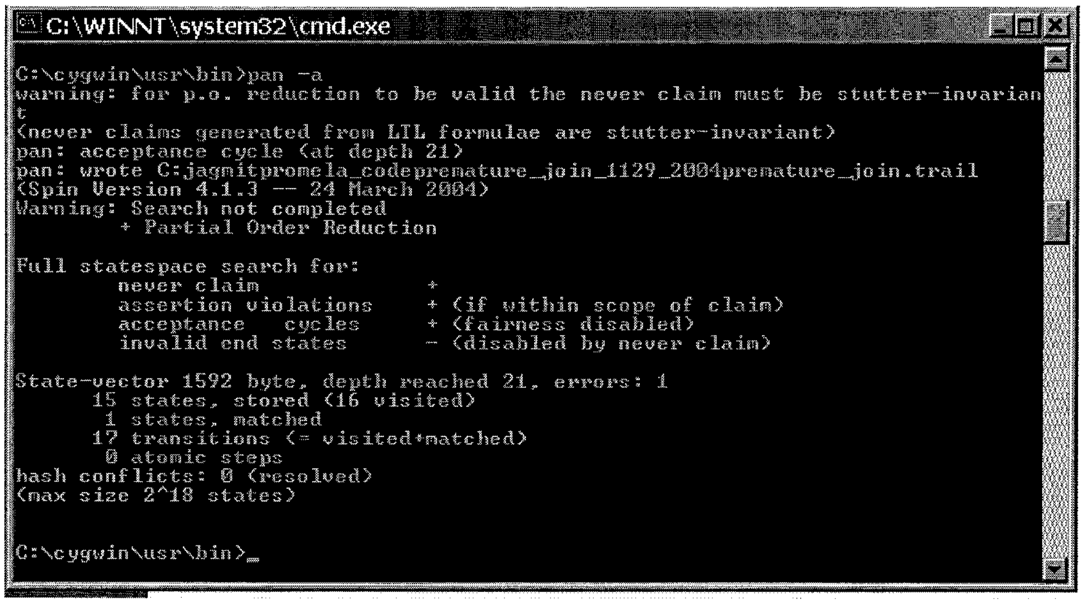
[57] Zeng, F. An Initial Study of Common Coding Pitfalls in Java Programs. Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '03), April 26th, 2003.

Appendix A - Snapshot of "Premature Join ()" Detection - Custom Detection


Profiling and Logging - Eclipse Platform
File Edit Navigate Search Project Profile Run Window Help
Console [<terminated> C:\wsdp-1.3\apache-ant\bin\ant.bat]
Console Tasks
Start
5:12 PM

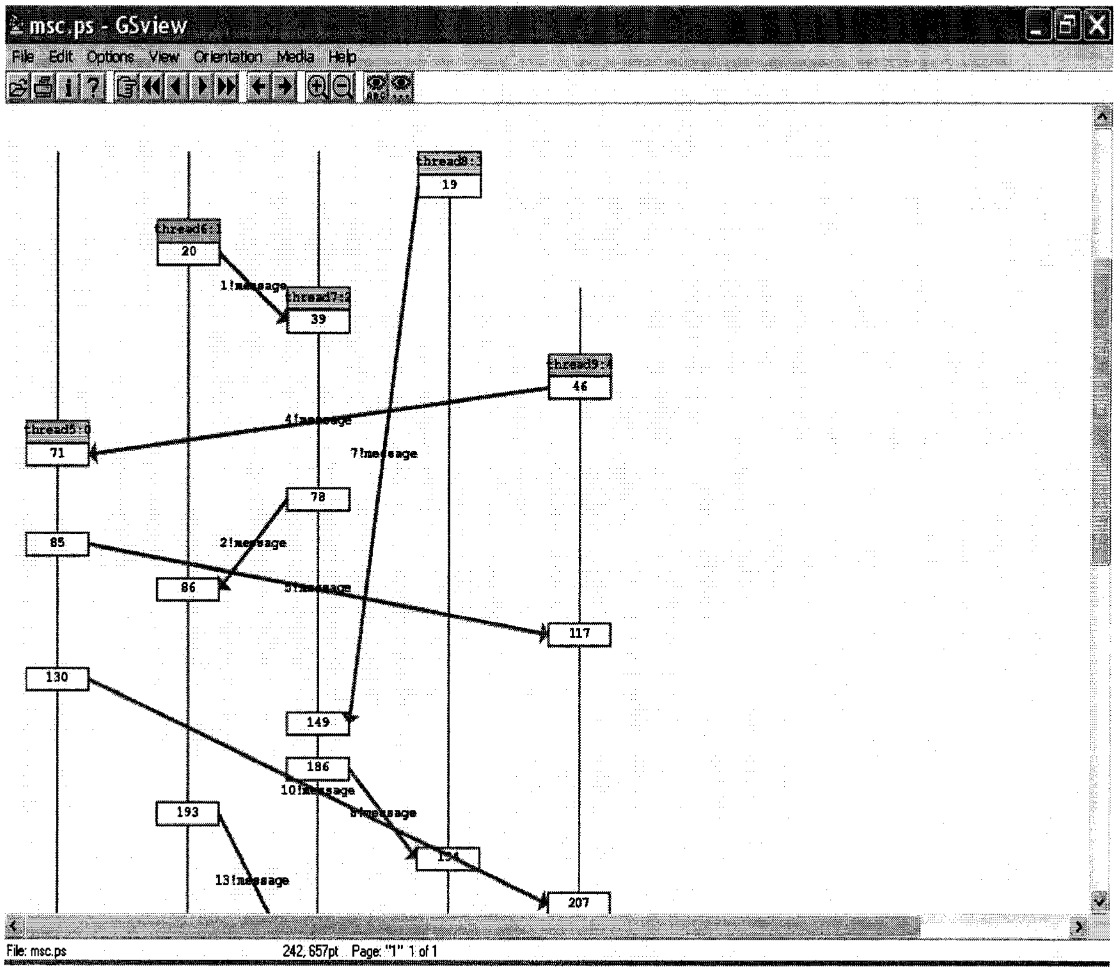
```
Buildfile: build.xml

compile:
     [echo] Compiling the schema external binding file...
      [xjc] Compiling file:/C:/jagmit/eclipse/workspace/premature_join_analysis/trace.dtd
      [xjc] Writing output to C:\jagmit\eclipse\workspace\premature_join_analysis
     [echo] Compiling the java source files...
    [javac] Compiling 109 source files to C:\jagmit\eclipse\workspace\premature_join_analysis

run:
     [echo] Running the sample application...
     [java] Detecting antiPattern premature call of join()method in a Trace
     [java] Premature call of join() method detected for threadId = 2

BUILD SUCCESSFUL
Total time: 31 seconds
```

Appendix B - Snapshot of "Premature Join ()" - Model Based Detection

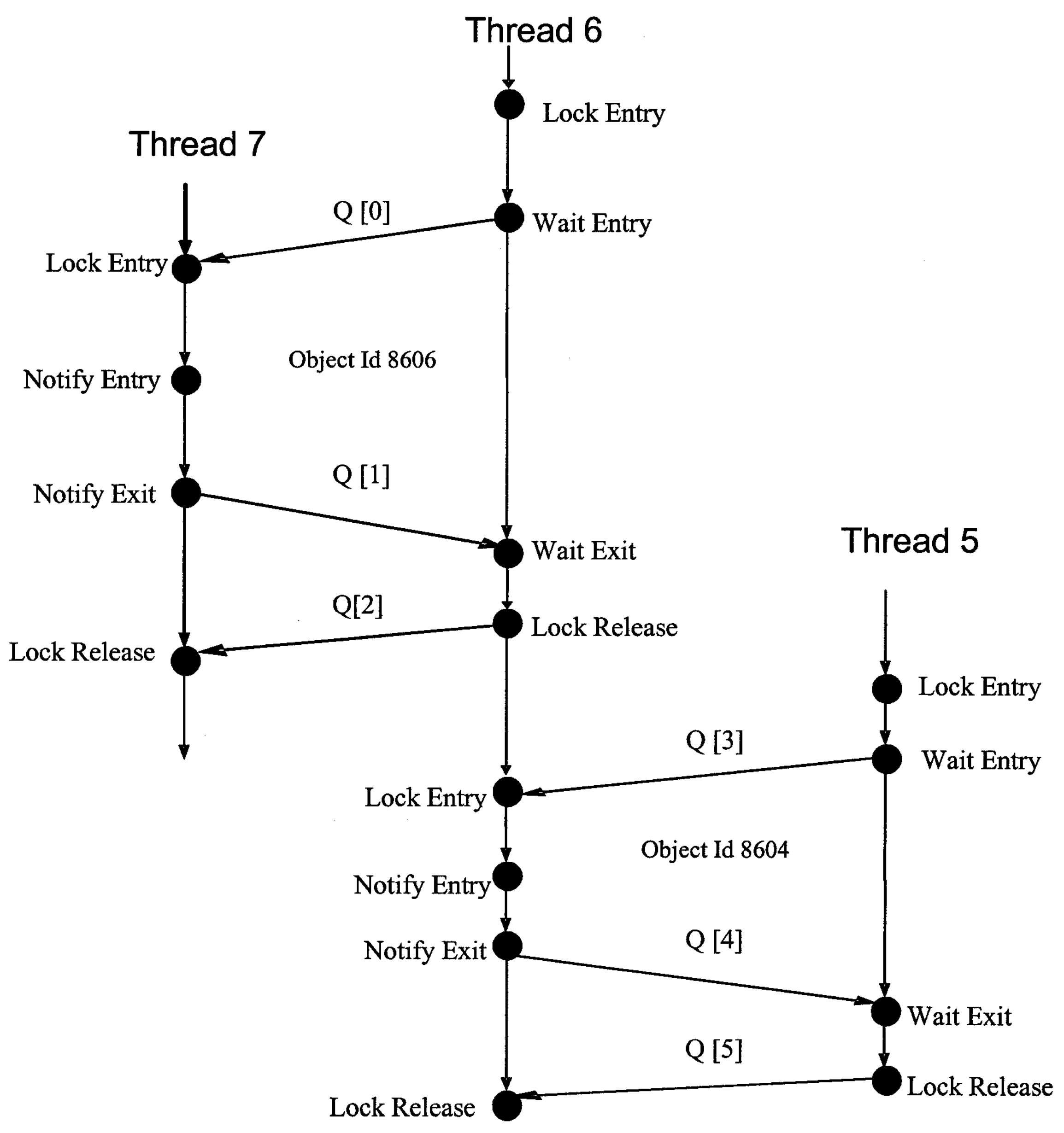


```
C:\WINNT\system32\cmd.exe

C:\cygwin\usr\bin>pan -a
warning: for p.o. reduction to be valid the never claim must be stutter-invarian
t
(never claims generated from LTL formulae are stutter-invariant)
pan: acceptance cycle (at depth 21)
pan: wrote C:jagmitpromela_codepremature_join_1129_2004premature_join.trail
(Spin Version 4.1.3 -- 24 March 2004)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 1592 byte, depth reached 21, errors: 1
      15 states, stored (16 visited)
       1 states, matched
      17 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)


C:\cygwin\usr\bin>_
```

Appendix C - Snapshot of MSC - A Message Sequence Chart

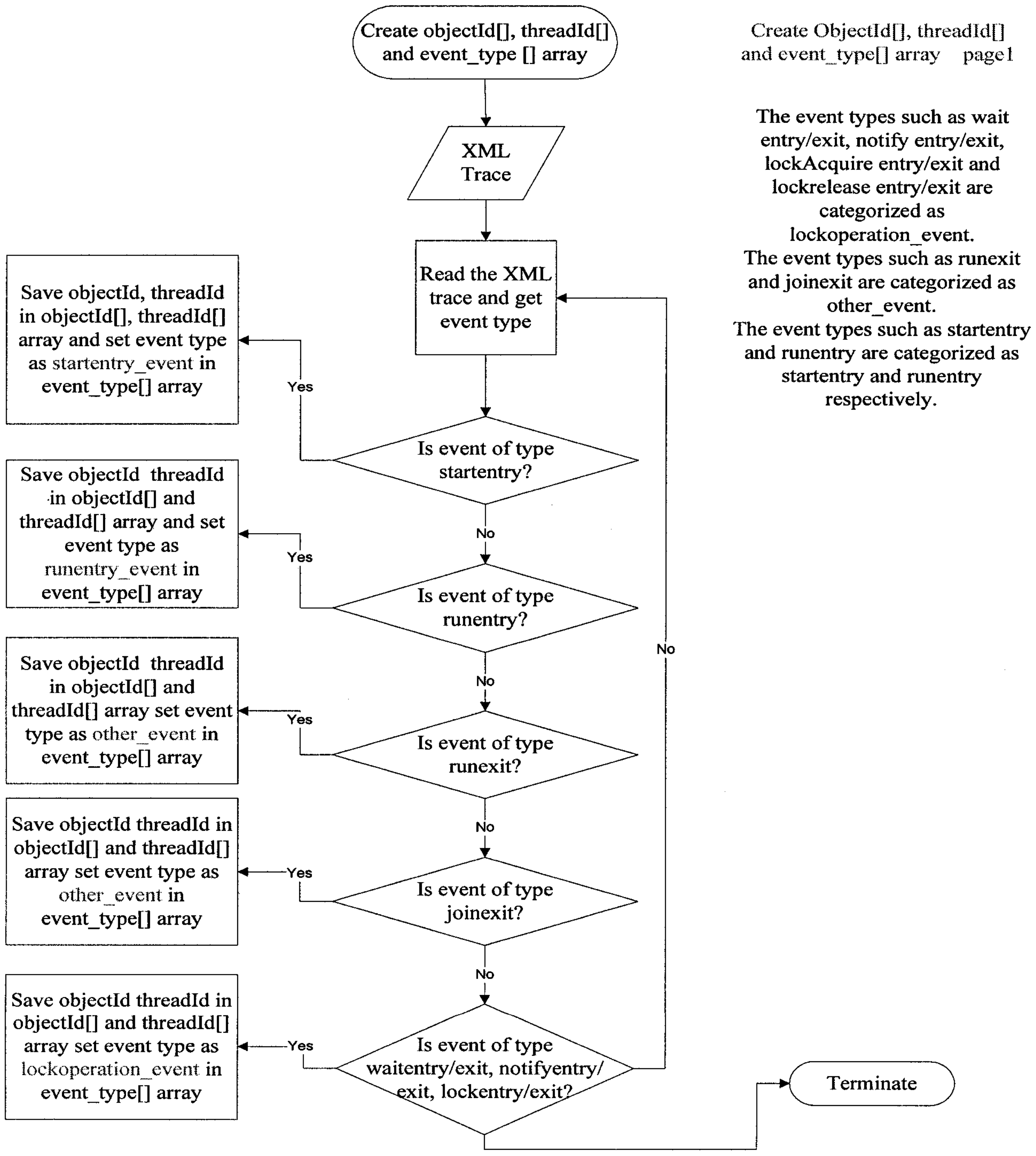


Appendix D - Ordering of the Events on the same Object

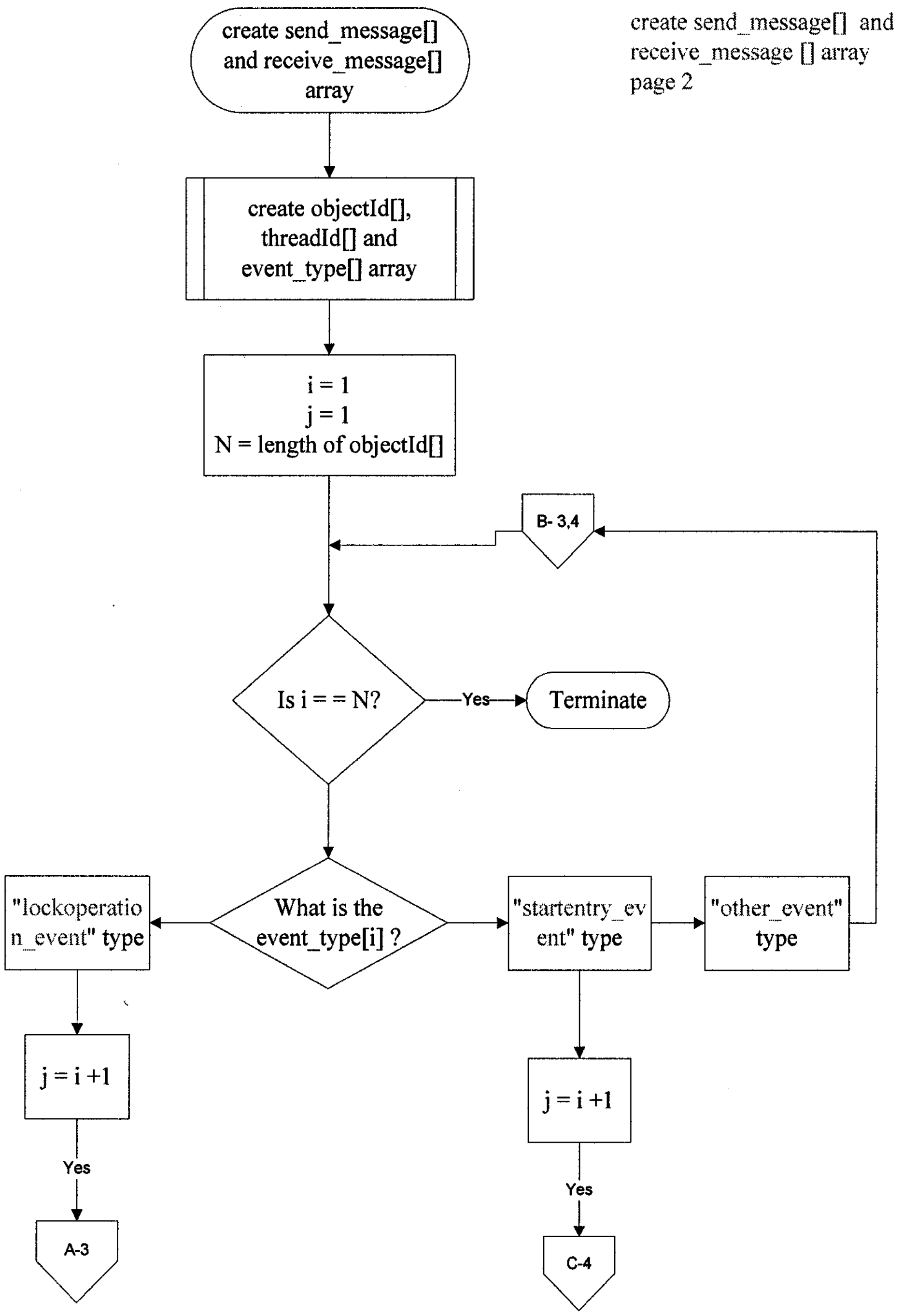


Appendix E - Flowchart for XML-PROMELA Extraction

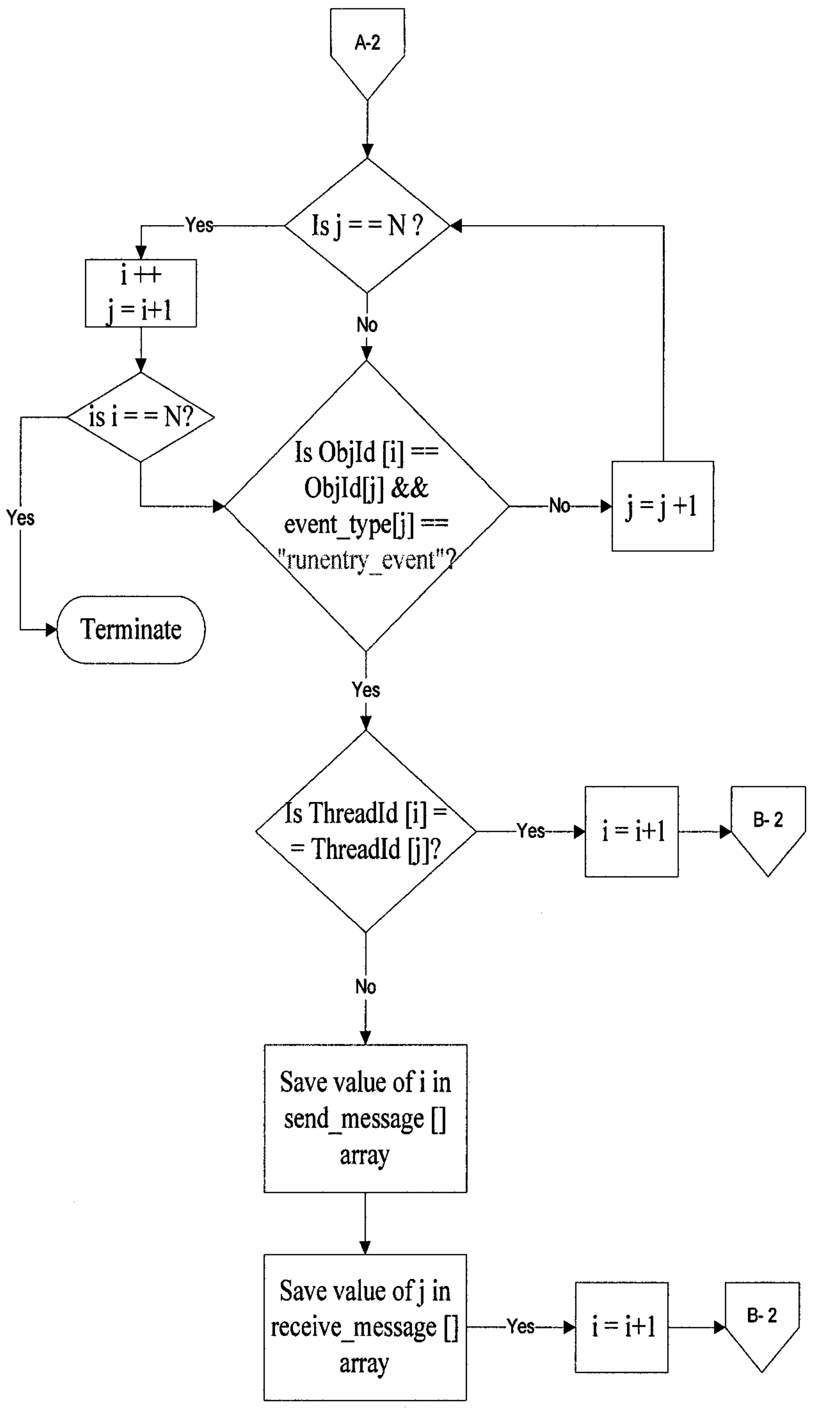
The XML trace is parsed and objectIds and threadIds of the relevant events such as waitentry/exit, notifyentry/exit, lockAcquire entry/exit, lockrelease entry/exit, startentry, runentry,runexit and joinexit are saved in objectId[] and threadId[] arrays. Then event types such as lockoperation_event, startentry_event,runentry_event and other_event are saved in event_type[] array .


Create objectId[], threadId[] and event_type [] array
XML Trace
Read the XML trace and get event type
Is event of type startentry?
Is event of type runentry?
Is event of type runexit?
Is event of type joinexit?
Is event of type waitentry/exit, notifyentry/exit, lockentry/exit?
Yes
No
Save objectId, threadId in objectId[], threadId[] array and set event type as startentry_event in event_type[] array
Save objectId threadId in objectId[] and threadId[] array and set event type as runentry_event in event_type[] array
Save objectId threadId in objectId[] and threadId[] array set event type as other_event in event_type[] array
Save objectId threadId in objectId[] and threadId[] array set event type as other_event in event_type[] array
Save objectId threadId in objectId[] and threadId[] array set event type as lockoperation_event in event_type[] array
Terminate
Create ObjectId[], threadId[] and event_type[] array page1
The event types such as wait entry/exit, notify entry/exit, lockAcquire entry/exit and lockrelease entry/exit are categorized as lockoperation_event.
The event types such as runexit and joinexit are categorized as other_event.
The event types such as startentry and runentry are categorized as startentry and runentry respectively.

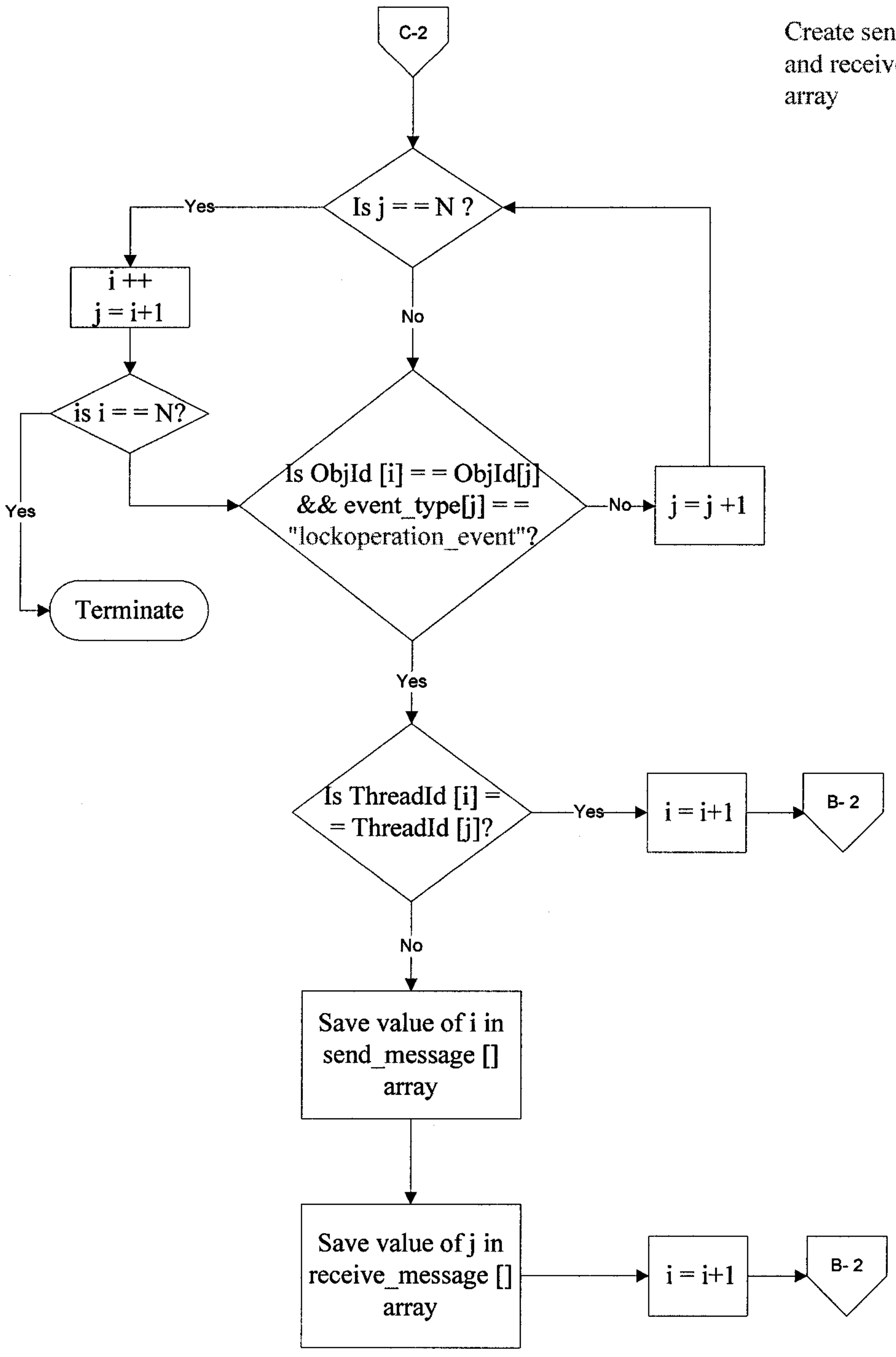
Here we iterate through the objectId[], threadId[] and event_type[] arrays and search and save those objectIds and threadIds in the arrays whose preceding objectId and event type is same but different threadId (exception in start and run synchronization, where event type start methodentry should be preceded by run methodentry event type). If such instances are found, we save the indexes of the objectIds in send_message [] array and save those index values from which they are different in receive_message [] array. These arrays are created to number send and receive messages to model synchronization.


create send_message[] and receive_message[] array
create send_message[] and receive_message [] array page 2
create objectId[], threadId[] and event_type[] array
i = 1
j = 1
N = length of objectId[]
B- 3,4
Is i = = N?
Yes
Terminate
What is the event_type[i] ?
"lockoperation_event" type
"startentry_event" type
"other_event" type
j = i +1
j = i +1
Yes
Yes
A-3
C-4

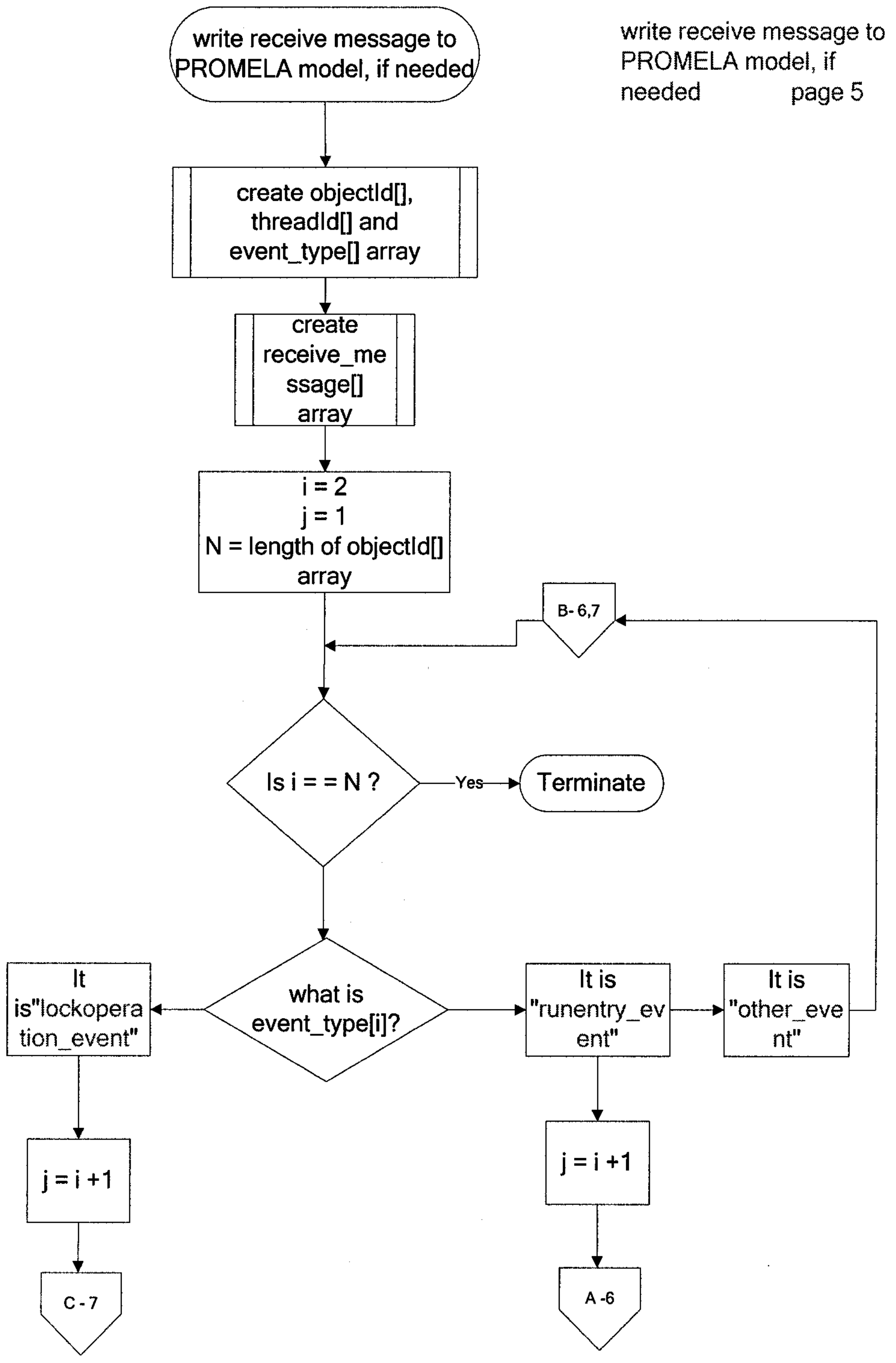
Create send_message[] and receive_message[] array- page 3


A-2
Is j = = N ?
Yes
i ++
j = i+1
No
is i = = N?
Yes
Terminate
Is ObjId [i] == ObjId[j] && event_type[j] == "runentry_event"?
No
j = j +1
Yes
Is ThreadId [i] = = ThreadId [j]?
Yes
i = i+1
B- 2
No
Save value of i in send_message [] array
Save value of j in receive_message [] array
Yes
i = i+1
B- 2

Create send_message[] and receive_message[] array page 4

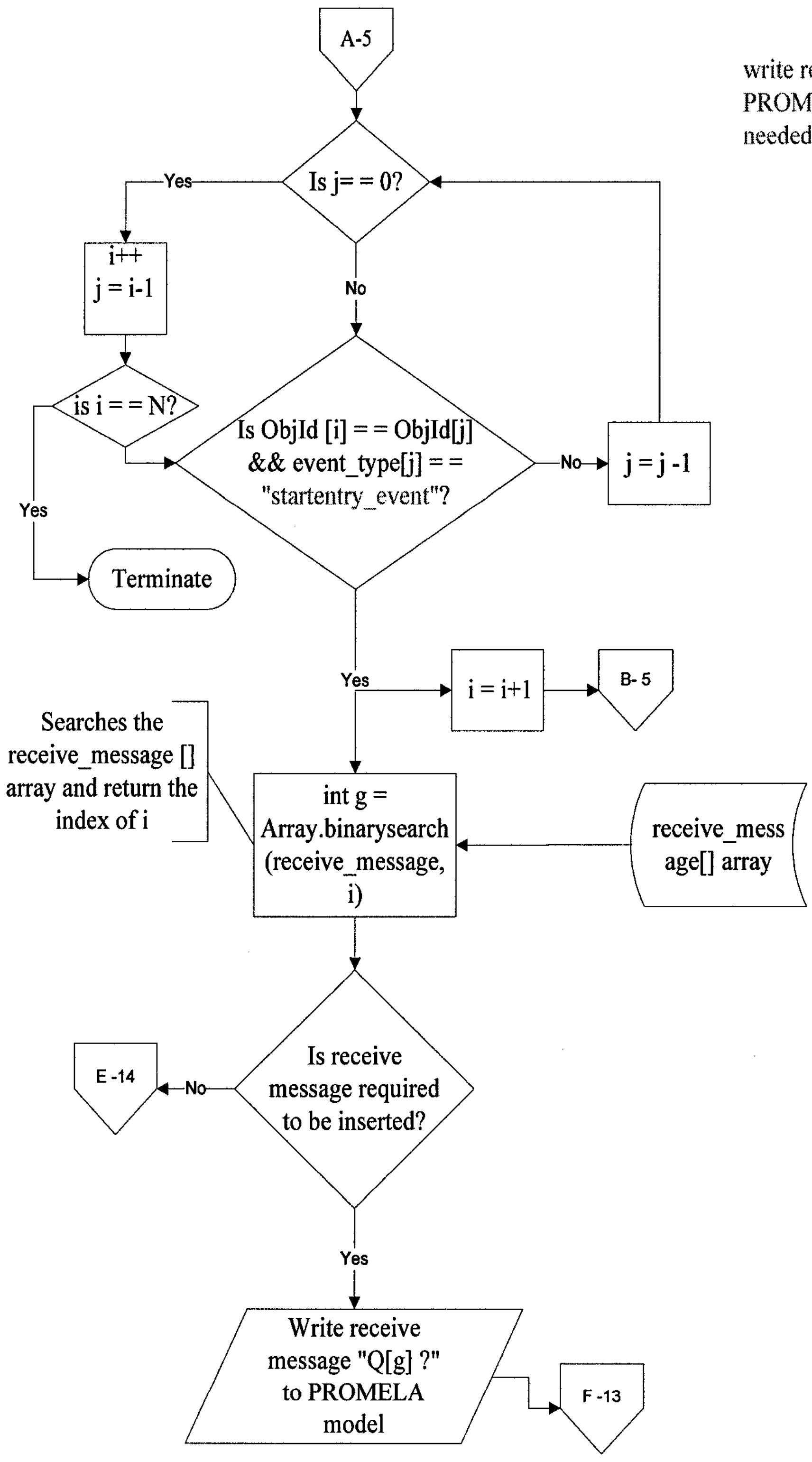

C-2
Is j = = N ?
Yes
No
i ++
j = i+1
is i = = N?
Yes
Terminate
Is ObjId [i] = = ObjId[j] && event_type[j] = = "lockoperation_event"?
No
j = j +1
Yes
Is ThreadId [i] = = ThreadId [j]?
Yes
i = i+1
B- 2
No
Save value of i in send_message [] array
Save value of j in receive_message [] array
i = i+1
B- 2

The algorithm followed in this flowchart is the same as followed before for the creation receive_message [] (in page 2, 3 & 4), except that the indexes are not saved in arrays but rather their index keys (2nd level) are found in receive_message []. While writing events to the PROMELA model, the receive messages are numbered according to these index keys.

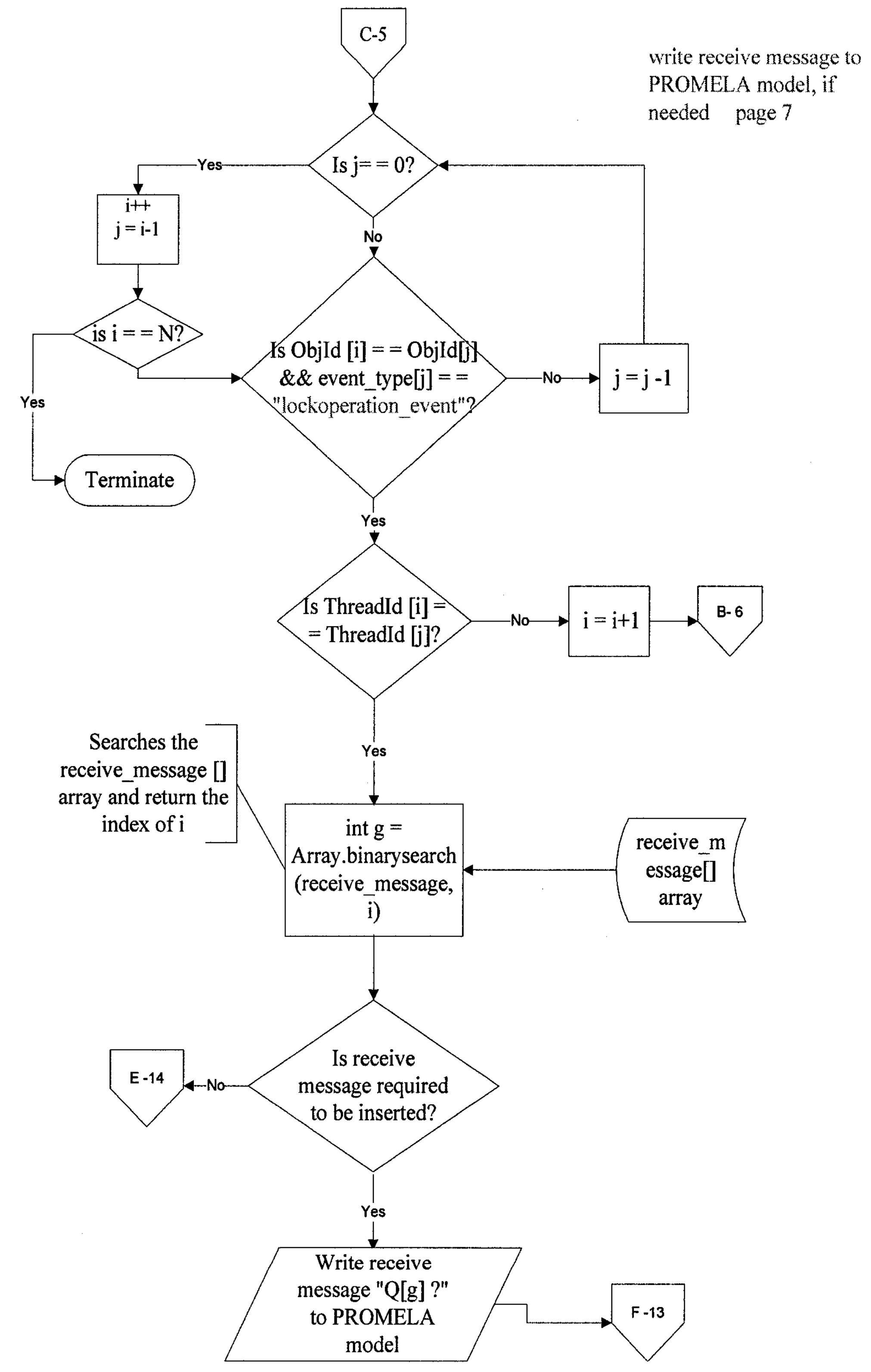


write receive message to PROMELA model, if needed
write receive message to PROMELA model, if needed page 5
create objectId[], threadId[] and event_type[] array
create receive_message[] array
i = 2
j = 1
N = length of objectId[] array
B- 6,7
Is i = = N ?
Yes
Terminate
what is event_type[i]?
It is"lockoperation_event"
It is "runentry_event"
It is "other_event"
j = i +1
j = i +1
C - 7
A -6

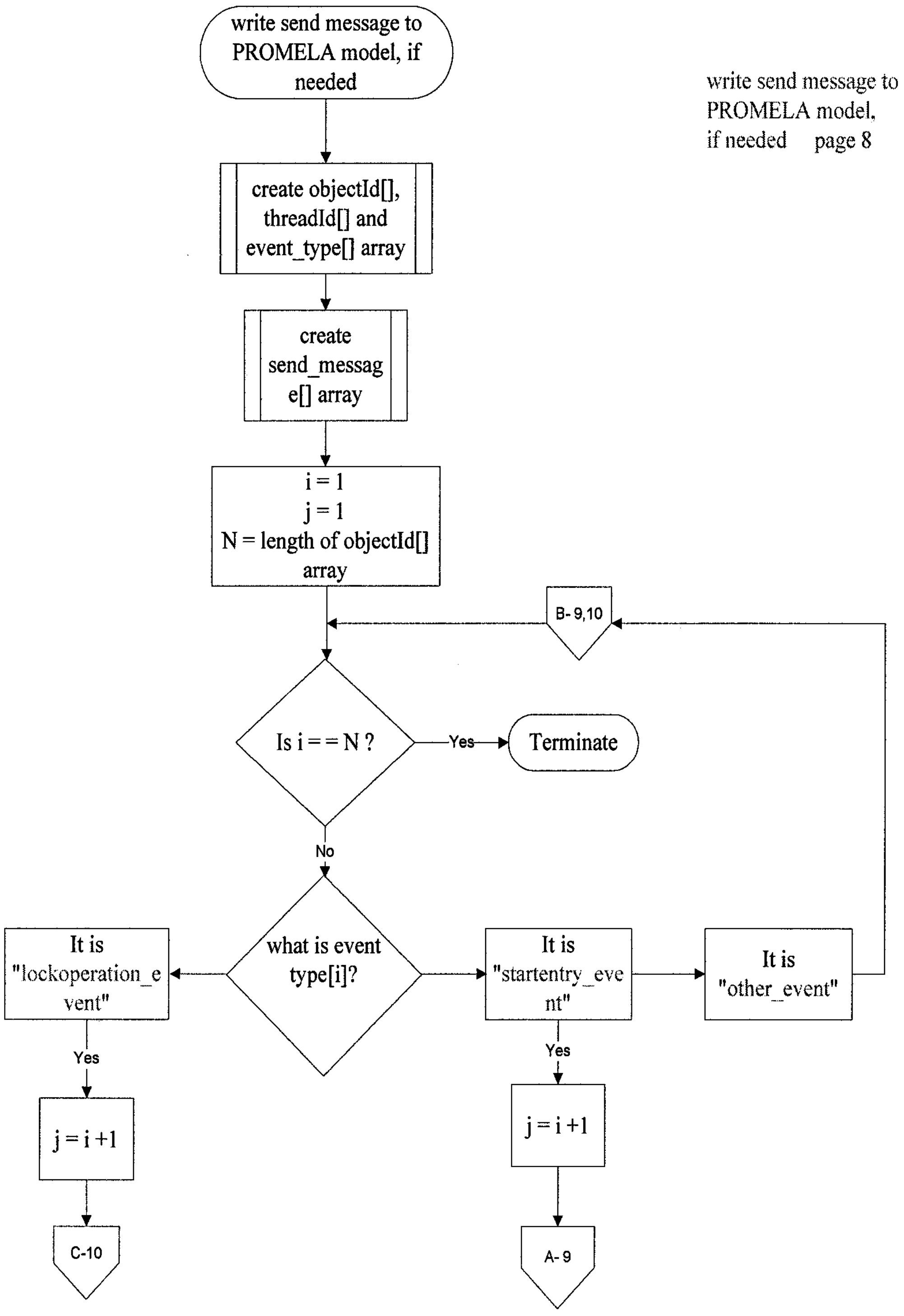
write receive message to PROMELA model, if needed page 6



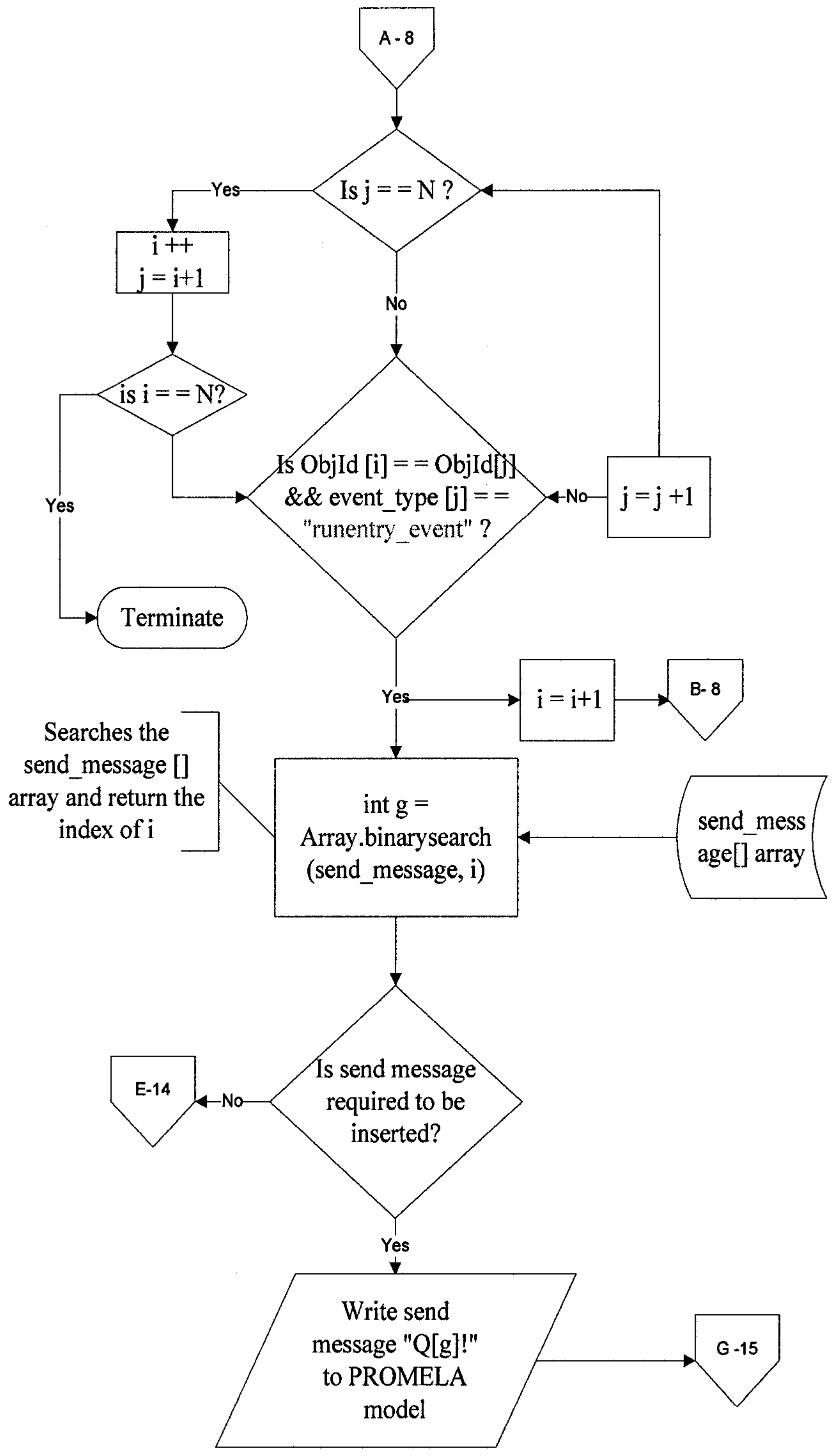
A-5
Is j= = 0?
Yes
No
i++
j = i-1
is i = = N?
Yes
Terminate
Is ObjId [i] = = ObjId[j] && event_type[j] = = "startentry_event"?
No
j = j -1
Yes
i = i+1
B- 5
Searches the receive_message [] array and return the index of i
int g = Array.binarysearch (receive_message, i)
receive_message[] array
Is receive message required to be inserted?
No
E -14
Yes
Write receive message "Q[g] ?" to PROMELA model
F -13


C-5
write receive message to PROMELA model, if needed page 7
Is j= = 0?
Yes
i++
j = i-1
is i = = N?
Yes
Terminate
No
Is ObjId [i] = = ObjId[j] && event_type[j] = = "lockoperation_event"?
No
j = j -1
Yes
Is ThreadId [i] = = ThreadId [j]?
No
i = i+1
B- 6
Yes
Searches the receive_message [] array and return the index of i
int g = Array.binarysearch (receive_message, i)
receive_message[] array
Is receive message required to be inserted?
No
E -14
Yes
Write receive message "Q[g] ?" to PROMELA model
F -13

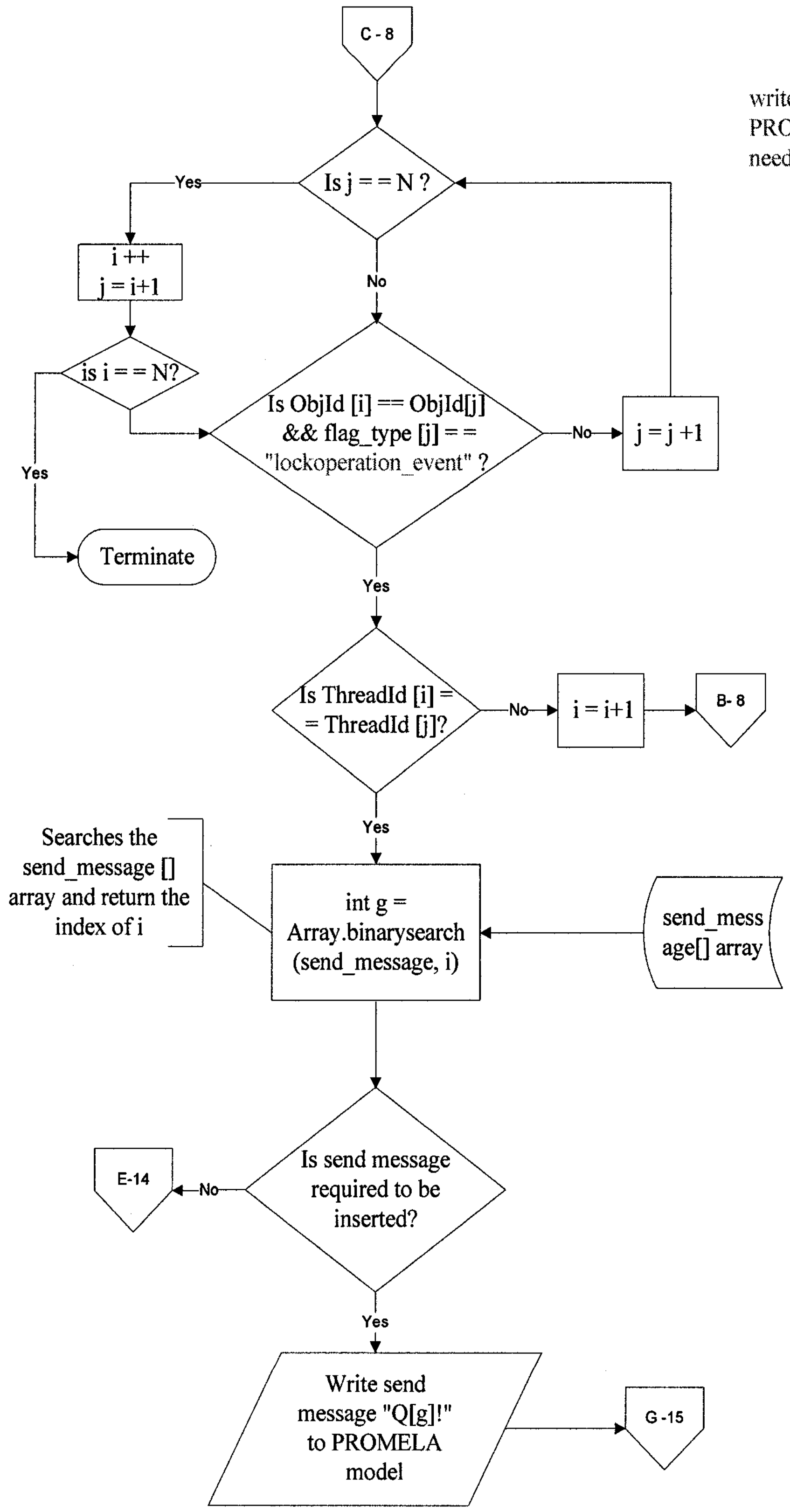
The algorithm followed in this flowchart is the same as followed before for the creation of send_message[] (page 2,3 & 4), except that the indexes are not saved in arrays but rather their index keys (2nd level) are found in send_message array. While writing events to the PROMELA model the send messages are numbered according to these index keys.


write send message to PROMELA model, if needed
write send message to PROMELA model, if needed page 8
create objectId[], threadId[] and event_type[] array
create send_message[] array
i = 1
j = 1
N = length of objectId[] array
B-9,10
Is i = = N ?
Yes
Terminate
No
what is event type[i]?
It is "lockoperation_event"
It is "startentry_event"
It is "other_event"
Yes
Yes
j = i +1
j = i +1
C-10
A-9

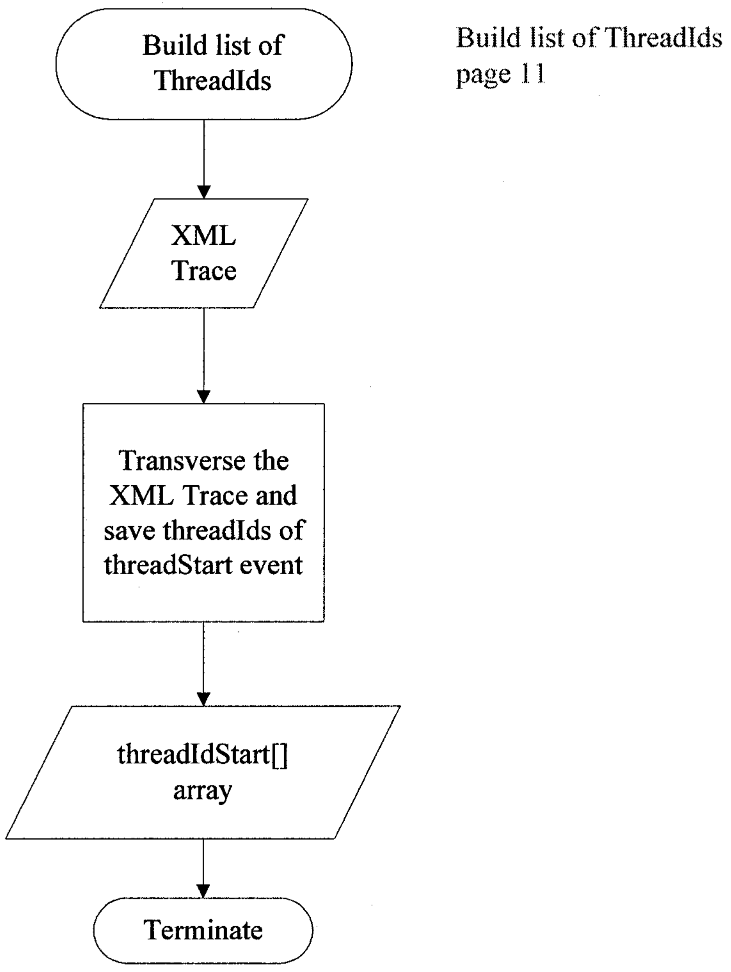
write send messages to PROMELA model, if needed page 9


A - 8
Is j = = N ?
Yes
No
i ++
j = i+1
is i = = N?
Yes
Terminate
Is ObjId [i] = = ObjId[j] && event_type [j] = = "runentry_event" ?
No
j = j +1
Yes
i = i+1
B- 8
Searches the send_message [] array and return the index of i
int g = Array.binarysearch (send_message, i)
send_message[] array
Is send message required to be inserted?
No
E-14
Yes
Write send message "Q[g]!" to PROMELA model
G -15

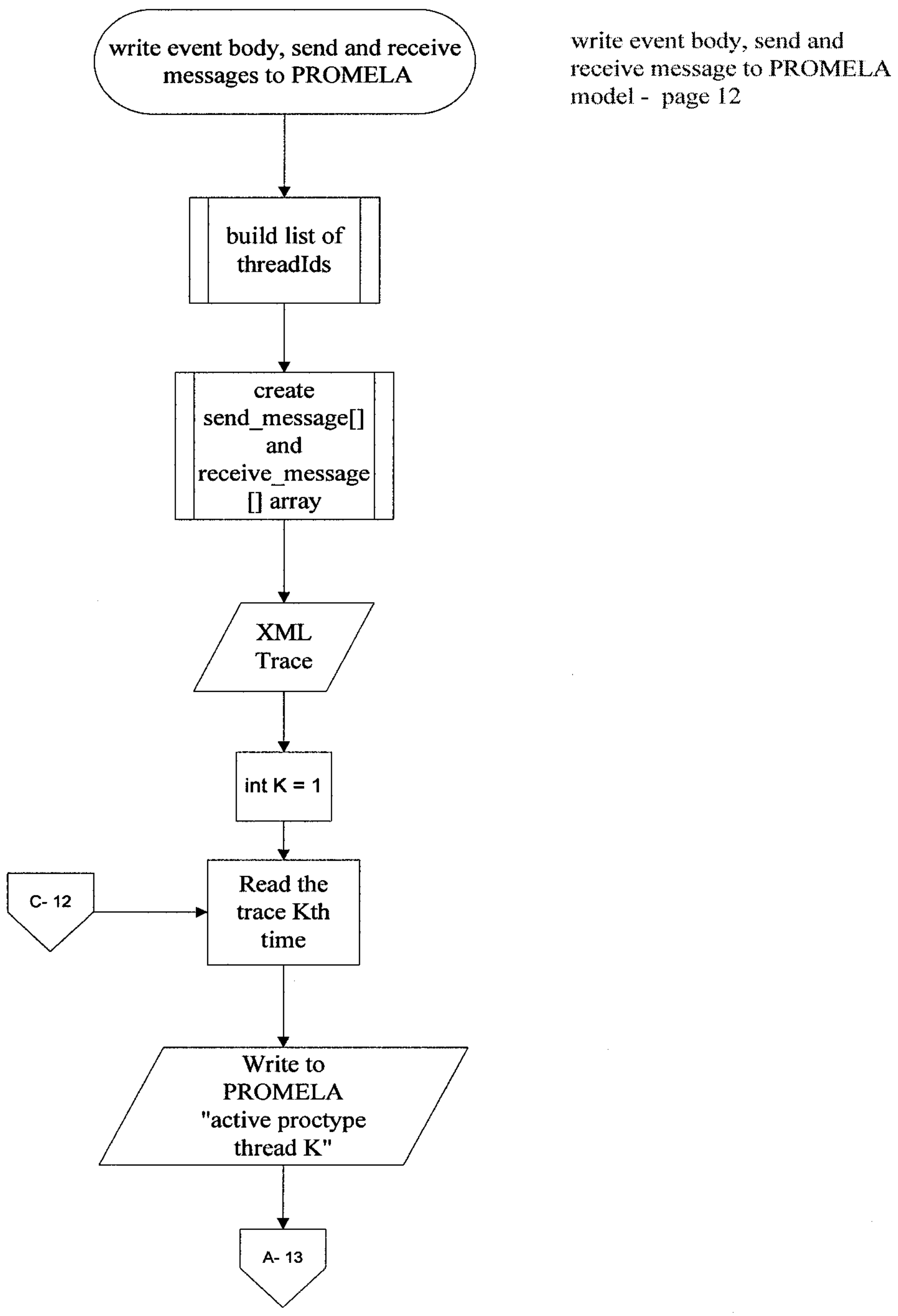
write send messages to PROMELA model, if needed page 10


C - 8
Is j = = N ?
Yes
i ++
j = i+1
is i = = N?
Yes
Terminate
No
Is ObjId [i] == ObjId[j] && flag_type [j] = = "lockoperation_event" ?
No
j = j +1
Yes
Is ThreadId [i] = = ThreadId [j]?
No
i = i+1
B- 8
Yes
Searches the send_message [] array and return the index of i
int g = Array.binarysearch (send_message, i)
send_message[] array
Is send message required to be inserted?
No
E-14
Yes
Write send message "Q[g]!" to PROMELA model
G -15

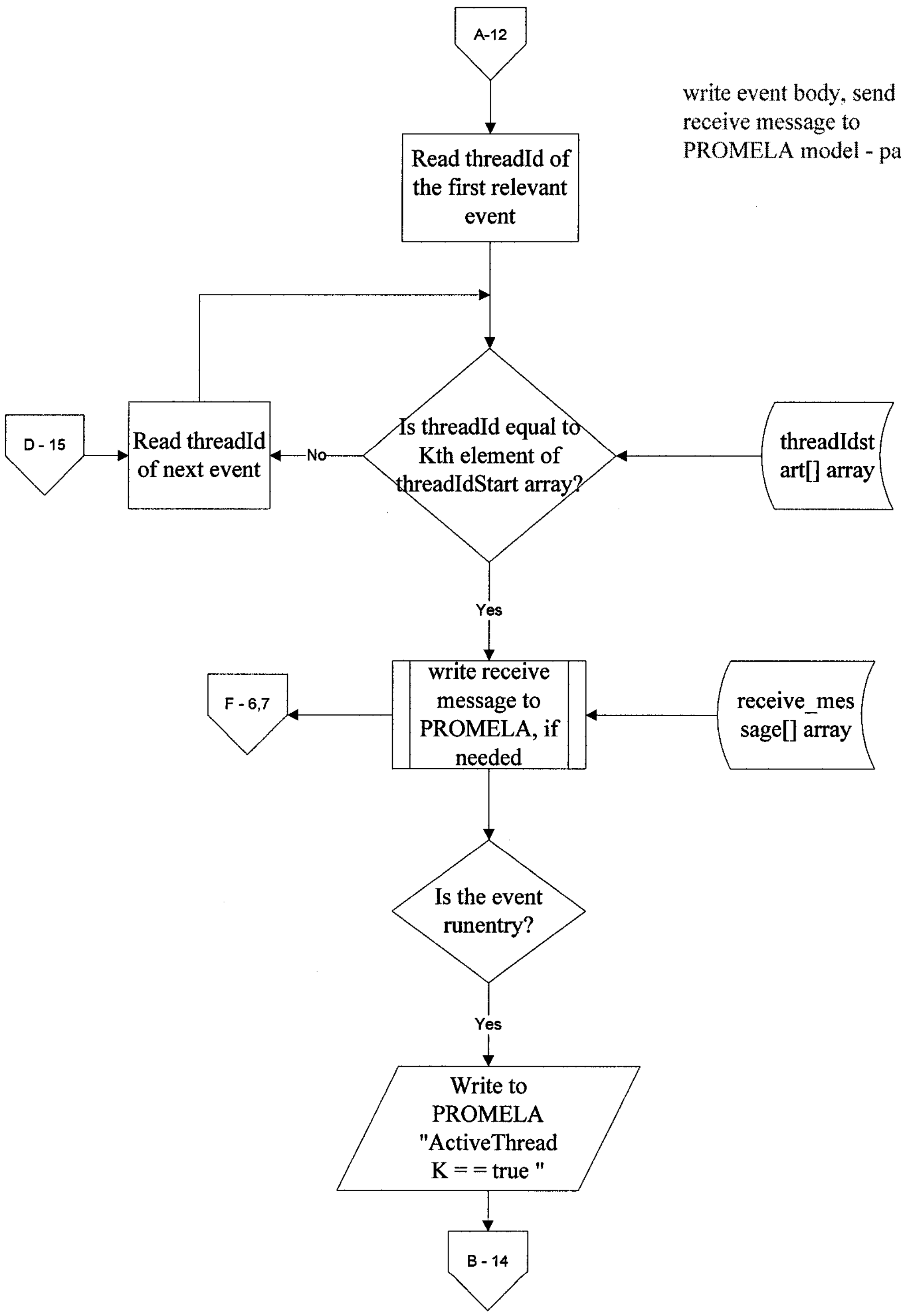
This flowchart creates threadIdStart [] array, containing the list of threadIds of the XML trace.

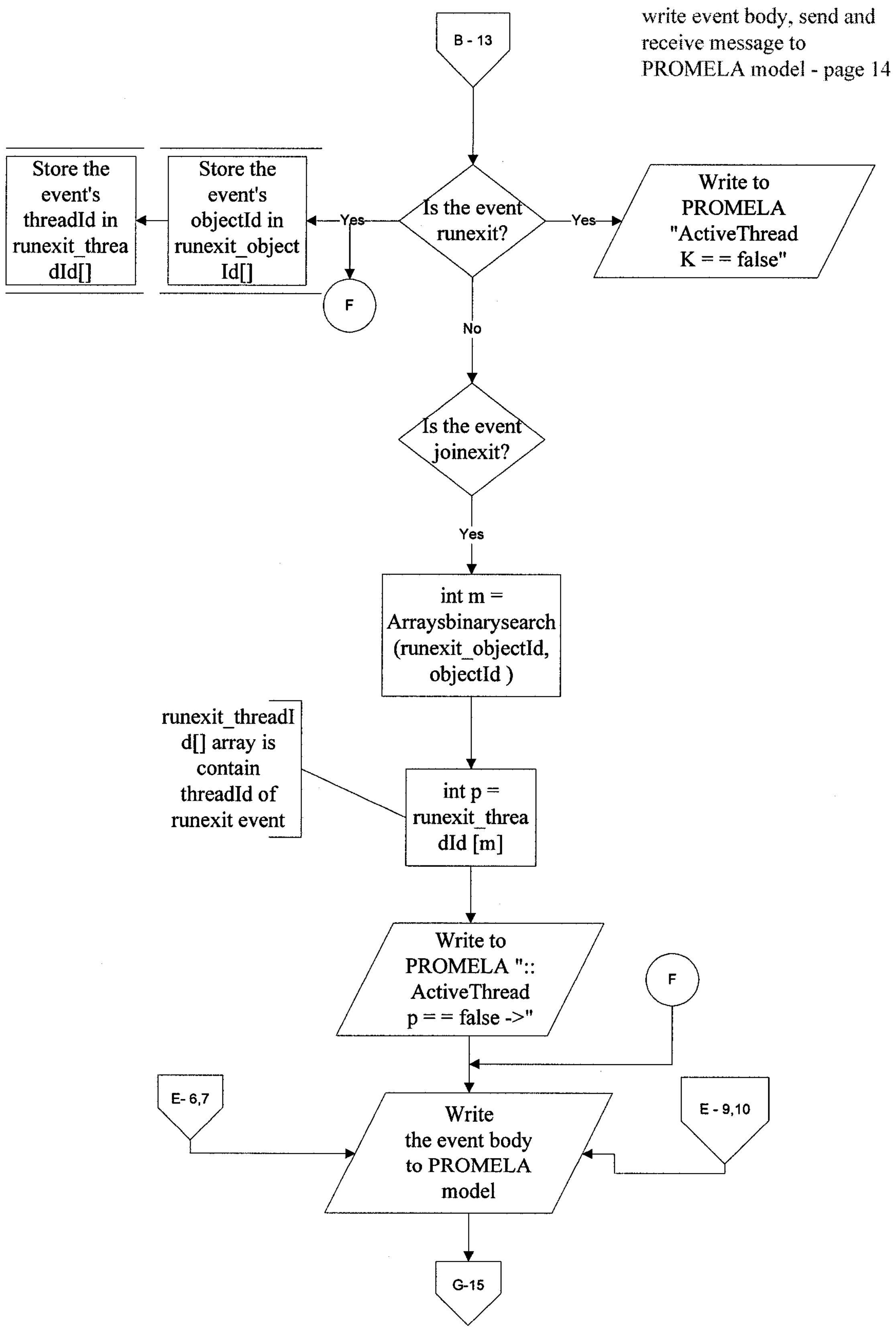

Build list of ThreadIds
Build list of ThreadIds page 11
XML Trace
Transverse the XML Trace and save threadIds of threadStart event
threadIdStart[] array
Terminate

This flowchart, explains the algorithm for writing events to PROMELA model. Events of the XML trace are not written to the PROMELA model in the order they are read from the XML trace, but in different order. For example, all the relevant events of thread Id 4 are written first, then the subsequent events of thread Id 5 are written next and so on.


write event body, send and receive messages to PROMELA
write event body, send and receive message to PROMELA model - page 12
build list of threadIds
create send_message[] and receive_message [] array
XML Trace
int K = 1
C- 12
Read the trace Kth time
Write to PROMELA "active proctype thread K"
A- 13

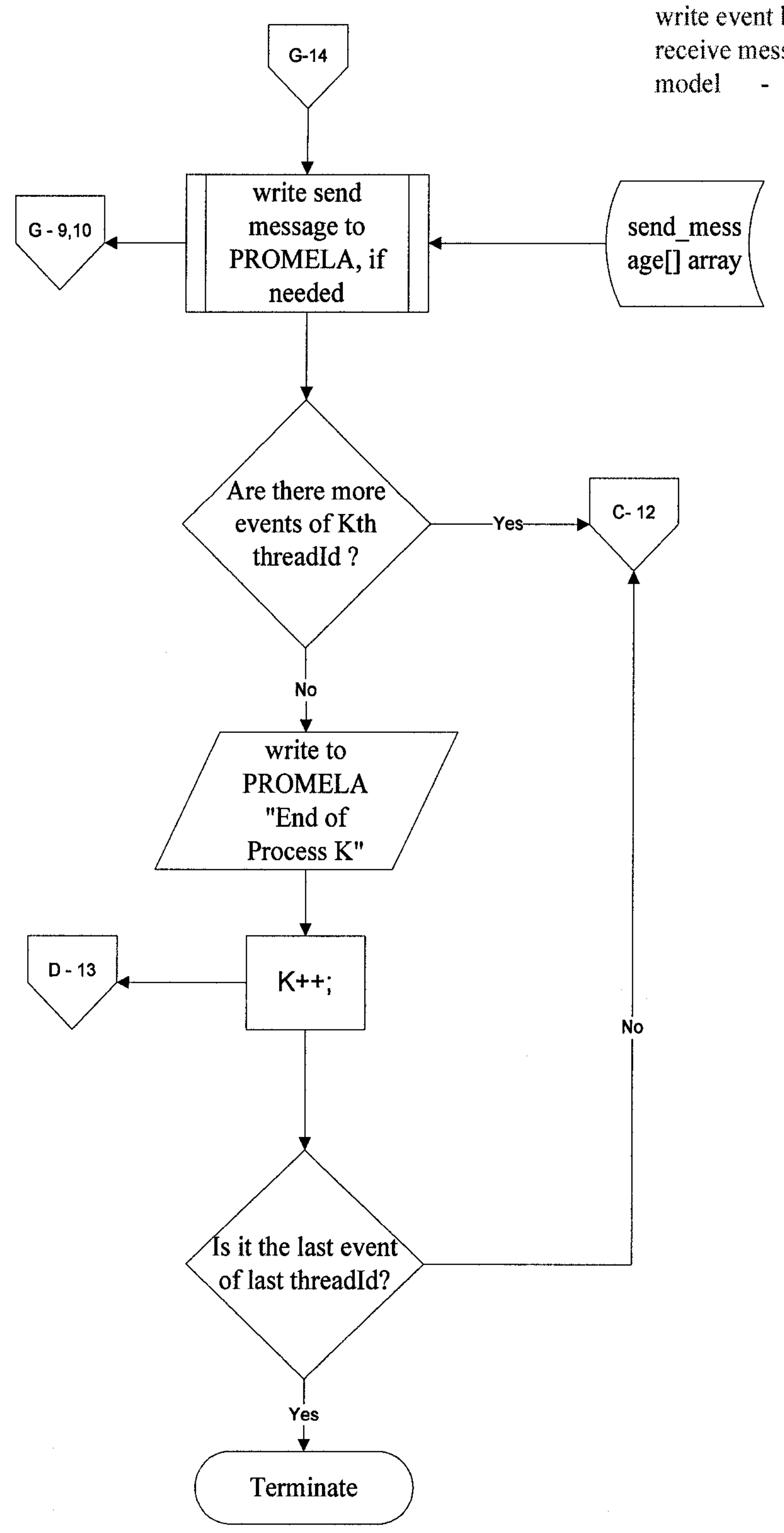
write event body, send and receive message to PROMELA model - page 13


A-12
Read threadId of the first relevant event
Is threadId equal to Kth element of threadIdStart array?
threadIdstart[] array
No
D - 15
Read threadId of next event
Yes
write receive message to PROMELA, if needed
F - 6,7
receive_message[] array
Is the event runentry?
Yes
Write to PROMELA "ActiveThread K = = true "
B - 14



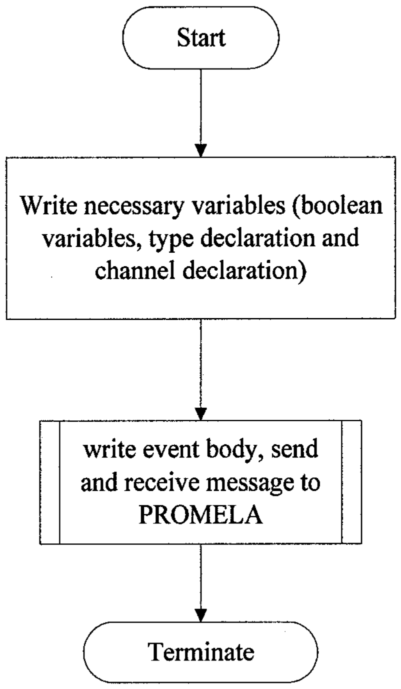
write event body, send and receive message to PROMELA model - page 14
B - 13
Is the event runexit?
Yes
Write to PROMELA "ActiveThread K = = false"
Yes
Store the event's objectId in runexit_object Id[]
Store the event's threadId in runexit_threa dId[]
F
No
Is the event joinexit?
Yes
int m = Arraysbinarysearch (runexit_objectId, objectId)
runexit_threadI d[] array is contain threadId of runexit event
int p = runexit_threa dId [m]
Write to PROMELA ":: ActiveThread p = = false ->"
F
E- 6,7
E - 9,10
Write the event body to PROMELA model
G-15

write event body, send and receive message to PROMELA model - page 15


G-14
write send message to PROMELA, if needed
G - 9,10
send_message[] array
Are there more events of Kth threadId ?
Yes
C- 12
No
write to PROMELA "End of Process K"
K++;
D - 13
No
Is it the last event of last threadId?
Yes
Terminate

This flowchart describes the creation of a PROMELA model with send and receive messages inserted in addition to event body. Before writing event body to PROMELA model, we check if send or receive messages are required to be written; if required we number and insert send or receive message before or after the event body.

The main program - page 16



Start
Write necessary variables (boolean variables, type declaration and channel declaration)
write event body, send and receive message to PROMELA
Terminate