

EXTENDING ECLIPSE QUALITY ASSURANCE  
PLATFORM FOR DETECTION OF  
JAVA MULTI THREADED DEFICIENCIES

**Jagmit Singh**

Department of Electrical and Computer Engineering  
Concordia University, Montreal

A thesis submitted to the Faculty of Graduate Studies and Research in partial  
fulfillment of the requirements for the Degree of Master of Applied Sciences

© Jagmit Singh

**Acknowledgements**

**Abstract**

# Table of Contents:

<b>ABSTRACT</b> .....	<b>2</b>
<b>TABLE OF CONTENTS:</b> .....	<b>3</b>
<b>1 INTRODUCTION AND MOTIVATION</b> .....	<b>9</b>
<b>2 CHAPTER 2 JAVA MULTITHREADING</b> .....	<b>12</b>
2.1 INTRODUCTION TO JAVA MULTITHREADING .....	12
2.1.1 <i>Definition of Java Multithreading and Related Terms</i> .....	12
2.1.2 <i>Benefits of writing multithreaded programs</i> .....	13
2.1.3 <i>Java implementation of Multithreading</i> .....	14
2.2 RELATED TERMS OF MULTITHREADING.....	15
2.2.1 <i>Locks</i> .....	15
2.2.1.1 <i>Thread Synchronization in Java</i> .....	16
2.3 MULTITHREADED PROBLEMS .....	17
2.3.1 <i>Common Locking Problems</i> .....	17
2.3.2 <i>Other Multithreaded Problems</i> .....	18
2.3.2.1 <i>Race conditions</i> .....	18
2.3.2.2 <i>Livelocks</i> .....	19
2.3.2.3 <i>Efficiency and Quality Problems</i> .....	20
2.4 ANTIPATTERNS-BASED DETECTION APPROACH .....	20
2.4.1 <i>Design antipatterns</i> .....	21
2.4.2 <i>Errors or Bug Patterns</i> .....	22
2.5 ANTIPATTERN LIBRARY .....	22
2.5.1 <i>Classification of Antipatterns</i> .....	22
2.5.2 <i>Antipattern Template:</i> .....	23
2.5.3 <i>Antipatterns:</i> .....	24
<i>Synchronized method call in cycle of lock graph</i> .....	25
2.5.4 <i>Antipattern Detection Summary</i> .....	25
<b>3 CHAPTER 3 MT JAVA ANALYSIS APPROACHES</b> .....	<b>28</b>
3.1 INTRODUCTION .....	28
3.2 DYNAMIC ANALYSIS .....	29
3.3 STATIC ANALYSIS .....	35
3.3.1 <i>Definition</i> .....	35
3.3.2 <i>Benefits</i> .....	37
3.3.3 <i>Challenges</i> .....	37
3.4 MODEL CHECKING .....	42
3.4.1 <i>Benefits</i> .....	43
3.4.2 <i>Challenges</i> .....	44
3.5 THEOREM PROVING .....	45
3.5.1 <i>Benefits</i> .....	45
3.5.2 <i>Challenges</i> .....	45
<b>4 CHAPTER 4 JAVA TRACE COLLECTION</b> .....	<b>46</b>
4.1 INTRODUCTION .....	46
4.2 INSTRUMENTATION APPROACHES .....	46
4.2.1 <i>Custom JVM Instrumentation</i> .....	47
4.2.2 <i>Java Profiling Interface</i> .....	49
4.2.3 <i>Java Debugging Architecture</i> .....	51
4.2.4 <i>Java Platform Profiling Architecture of J2SE 5.0</i> .....	52
4.3 SOURCE CODE LEVEL INSTRUMENTATION .....	54
4.4 JAVA BYTECODE .....	54

4.4.1	Java Bytecode Definition.....	55
4.4.2	Java Bytecode Format.....	55
4.4.3	Bytecode Instrumentation.....	56
4.4.3.1	Overview.....	56
4.4.3.2	Supported Events.....	56
4.5	INSTRUMENTATION TOOLS.....	58
4.5.1	Java Virtual Machine Instrumentation Based Tools.....	58
4.5.1.1	Tool: JinSight.....	58
4.5.2	Profiling Interface Based Tools.....	59
4.5.2.1	Tool: Hyades.....	59
4.5.3	Tools for Source-Code Level Instrumentation.....	60
4.5.3.1	Tool: JavaScope.....	60
4.5.4	Java Bytecode Instrumentation Tools/Toolkits.....	61
4.5.4.1	Tool: JProbe Threadalyzer.....	61
4.5.4.2	Tool: ByteCode Instrumentation Tool (BIT).....	63
4.5.4.3	Tool: JTrak.....	64
4.5.4.4	ToolKit: Byte Code Engineering Library (BCEL).....	65
4.5.4.5	Tool: JIAPI: Java Instrumentation API.....	66
4.6	OUR INSTRUMENTATION AND TRACE COLLECTION APPROACH.....	68
4.6.1	Introduction.....	68
4.6.2	Hyades Profiling and Tracing.....	69
4.6.2.1	Event Structure and there Attributes.....	69
4.6.2.1.1	IDs.....	70
4.6.2.1.2	Common attributes.....	70
4.6.2.1.3	Structural elements.....	71
4.6.2.1.4	Trace Behaviour Elements.....	72
4.6.2.1.5	Thread elements.....	72
4.6.2.1.6	Class Elements.....	72
4.6.2.1.7	Object Elements.....	73
4.6.2.1.8	Method elements.....	73
4.6.3	Bytecode Instrumentation using JTrak.....	74
4.6.3.1	Additional Events logged using the bytecode Instrumentation:.....	76
4.6.3.1.1	Variable Updates.....	76
4.6.3.1.1.1	Primitive Field Variables.....	77
4.6.3.1.1.2	Local Variables:.....	77
4.6.3.1.2	Monitor Enter and Exit.....	78
4.6.4	Trace Reduction.....	82
4.6.5	Benefits/Limitations of our Instrumentation Approach:.....	83
4.6.5.1	Hyades Tracing.....	83
4.6.5.1.1	Benefits of the Hyades tracing:.....	83
4.6.5.1.2	Limitation of the Hyades tracing.....	84
4.6.5.2	Bytecode instrumentation based tracing.....	84
4.6.5.2.1	Benefits of bytecode instrumentation based tracing.....	84
4.6.5.2.2	Limitation/side effect of bytecode instrumentation based tracing..	85
4.7	EXECUTION OF THE INSTRUMENTED PROGRAM.....	85
4.8	CONCLUSION AND FUTURE WORK.....	86
<b>5</b>	<b>CHAPTER 5: CUSTOM BASED DETECTION APPROACH.....</b>	<b>87</b>
5.1	INTRODUCTION AND MOTIVATION.....	87
5.2	APPROACH OVERVIEW.....	88
5.3	ANTI-PATTERN FORMALIZATION.....	90
5.3.1	Formalization of the “double call of start () method” antipattern.....	90
5.3.1.1	FSM Formalization of “double call of start () method“.....	90
5.3.1.2	EFSM Formalization of “double call of start () method”.....	91

5.3.2	<i>Formalization of the “PREMATURE JOIN CALL” antipattern</i> .....	91
5.3.2.1	FSM Formalization of “premature join call” antipattern .....	91
5.3.2.2	EFSM Formalization of “premature join call” antipattern .....	92
5.3.3	<i>Formalization of Wait Stall</i> .....	93
5.3.3.1	FSM Formalization of wait stall .....	93
5.3.3.2	EFSM Formalization of wait stall .....	94
5.4	CUSTOM DETECTORS IMPLEMENTATION .....	95
5.4.1	<i>Double Start () Implementation in Java</i> .....	95
5.4.2	<i>Premature Join () Implementation in Java</i> .....	96
5.5	CUSTOM DETECTION RESULTS .....	97
5.5.1	<i>Antipattern: Double Call of Start () method</i> .....	98
5.5.2	<i>Antipattern: Premature Call of Join () Method</i> .....	100
5.5.3	<i>Antipattern: Wait Stall</i> .....	100
5.6	ADVANTAGES/LIMITATIONS OF CUSTOM BASED APPROACH: .....	101
5.6.1	<i>Advantages of Custom Based Detection Approach</i> .....	101
5.6.2	<i>Limitations of Custom Based Detection Approach</i> .....	102
<b>6</b>	<b>CHAPTER 6: MODEL CHECKING WITH SPIN</b> .....	<b>103</b>
6.1	INTRODUCTION .....	103
6.1.1	<i>Model – Checking</i> .....	103
6.2	SPIN MODEL-CHECKER .....	104
6.2.1	<i>Language of SPIN</i> .....	105
6.2.2	<i>Features of Spin</i> .....	105
6.2.3	<i>DOCUMENTATION</i> .....	106
6.2.4	<i>AVAILABILITY</i> .....	106
<b>7</b>	<b>CHAPTER 7: MODELING TRACE WITH SPIN</b> .....	<b>107</b>
7.1	INTRODUCTION AND MOTIVATION .....	107
7.2	APPROACH OVERVIEW .....	108
7.3	TRANSLATION .....	111
7.3.1	<i>XML to PROMELA Translation</i> .....	113
7.3.2	<i>Synchronization in Java</i> .....	114
7.3.3	<i>Modeling Synchronization in PROMELA Model of Trace</i> .....	115
7.3.3.1	Message Passing Approach .....	115
7.3.3.1.1	Advantage and Disadvantages of Message Based Approach .....	117
7.3.3.2	Variable Based Approach .....	118
7.3.3.2.1	Advantages and Disadvantages Variable Based Approach .....	119
7.4	XML to PROMELA TRANSLATION IMPLEMENTATION .....	119
7.5	PROPERTY SPECIFICATION IN PROMELA NEVER CLAIM AND LTL .....	121
7.5.1	<i>Temporal Logic Overview</i> .....	122
7.5.1.1	Linear-Time Temporal Logic .....	122
7.5.2	<i>Property Specification</i> .....	122
7.6	MODEL BASED APPROACH’S RESULTS .....	124
7.6.1	<i>Antipattern: Double Call of Start () method</i> .....	125
7.6.1.1	Verification Data .....	125
7.6.2	<i>Antipattern: Premature Call of Join () Method</i> .....	126
7.6.2.1	Verification Data .....	126
7.7	OPEN PROBLEMS AND ALTERNATIVES FOR TRACE MODELING .....	126
7.7.1	<i>Alternatives to Trace Modelling</i> .....	128
<b>8</b>	<b>CHAPTER 8 COMPARISON</b> .....	<b>129</b>
8.1	INTRODUCTION AND MOTIVATION .....	129
8.2	EXPERIMENTS RESULTS: .....	129
8.3	COMPARISON .....	131
<b>9</b>	<b>CHAPTER 9: CONCLUSION AND FUTURE WORK</b> .....	<b>134</b>

## List of Tables

Table 2.1: Antipattern Template .....**Error! Bookmark not defined.**  
Table 2.2: Synchronized method call in cycle of lock graph..... **Error! Bookmark not defined.**  
Table 2.3: Summary of detection techniques and tools .....**Error! Bookmark not defined.**  
Table 4.1: The summary of the analyzed instrumentation tools..... **Error! Bookmark not defined.**  
Table 5.1: Analysis time for double start () detection .....**Error! Bookmark not defined.**  
Table 5.2: Analysis time for premature join () detection...**Error! Bookmark not defined.**  
Table 7.1: Analysis time for premature join () detection...**Error! Bookmark not defined.**  
Table 7.2: Verification & Compilation Time (double start ()) ..... **Error! Bookmark not defined.**  
Table 7.3: Model Generation Time (premature join).....**Error! Bookmark not defined.**  
Table 7.4: Verification & Compilation Time (premature join) ..... **Error! Bookmark not defined.**  
Table 8.1: Experimental Results .....**Error! Bookmark not defined.**

## List of Figures

Figure 2.1: Code sample to Run a Runnable in a Thread .....	15
Figure 2.2: A simple deadlock example [Art01] .....	17
Figure 4.1: A trace snapshot of JProbe .....	62
Figure 4.2: Sample of the source code.....	79
Figure 4.3: <i>Sample of instrumented Bytecode</i> .....	80
Figure 4.4: <i>Sample of uninstrumented Bytecode</i> .....	81
Figure 4.5: Sample of the Instrumentor code.....	81
Figure 4.6: <i>Sample of the trace obtained after instrumentation (it contains the additional events logged)</i> .....	82
Figure 5.1: <i>Offline custom based trace analysis architecture</i> .....	89
Figure 5.2: <i>FSM formalization of double start ()</i> .....	90
Figure 5.3: <i>EFSM formalization of double start ()</i> .....	91
Figure 5.4: <i>FSM formalization of premature join ()</i> .....	92
Figure 5.5: <i>EFSM formalization of premature join ()</i> .....	93
Figure 5.6: <i>FSM formalization of wait stall</i> .....	94
Figure 5.7: <i>EFSM formalization of wait stall</i> .....	94
Figure 5.8: <i>Code sample for detection of double start ()</i> .....	95
Figure 5.9: <i>Code sample for detection of premature join ()</i> .....	97
Figure 5.10: <i>Sample of Guest application code</i> .....	99
Figure 5.11: <i>Sample of Java trace for double start () detection</i> .....	100
Figure 5.12: <i>Sample of java trace for wait stall detection</i> .....	101
Figure 7.1: <i>Diagram of the approach workflow</i> .....	110
Figure 7.2: <i>XML to PROMELA Translation</i> .....	112
Figure 7.3: <i>SPIN: Couple of PROMELA Constants and Constructs</i> .....	113
Figure 7.4: <i>Sample of the PROMELA Model, synchronization based on Message Passing Approach</i> .....	117
Figure 7.5: <i>Sample of the PROMELA model, synchronization based on the variable based Approach</i> .....	119





# 1 Introduction and Motivation

Two important aspects of program verification are testing and the use of formal methods. Traditional testing techniques, however are very ad hoc and do not allow for formal specification and verification of high level logical properties that a system needs to satisfy. On the other hand, traditional formal methods such as model checking and theorem proving are rarely used in practise.

The general idea of the runtime analysis is extracting the relevant events from the executing Java multithreaded application program, and then analyzing the events (collected in trace) for properties or antipatterns. Here in this thesis, we collected trace of Java multithreaded application and then analyzed the trace for multithreaded antipatterns, whose presence in the target program can cause concurrency related errors such as deadlocks, livelocks and data races. The Java language is quite popular programming language in web applications on the internet as well as distributed mostly client/server applications and it is multithreaded in nature.

The runtime analysis can be defined as combining testing and formal methods. By merging testing and formal methods, runtime analysis achieve the benefits of both the approaches, while avoiding some of the pitfalls of the ad-hoc testing and the complexity of the theorem proving and state explosion problem of model checking. We developed two runtime approaches namely custom based and model based runtime analysis approach. The custom based approach is semiformal and model based approach is formal approach. The main difference between these two approaches is in the antipatterns detection approach. In the model based approach we generate the model of execution

trace and then analyse the generated model using SPIN model checker against the MT antipatterns specified in LTL. Whereas in custom-based analysis approach, first antipatterns are coded in Java and then execution trace is analyzed for antipatterns, using these java detectors.

In the end we compare these two approaches based on criteria such as quality of analysis, resource consumption, time usage, complexity, easy of usage and other factors.

The aim of formal verification (such as model checking) and testing is to check whether a program is correct. In other words formal verification and testing attempts to ensure that all the possible executions of software yield correct results. But run-time analysis assures of the current execution of a program. However there is the shortcoming of this approach: the entire state space of the system cannot be covered. The suggested runtime analysis framework can only be used to examine single execution traces, and therefore cannot be used to prove a system correct.

However the single execution trace contains much more information than what appears.

Model based analysis approach exploits this hidden information and thus can perform predictive trace analysis, it can explore the various possible events interleaving.

The run-time formal analysis can cause undesirable side effects to a target program; run-time formal analysis can slow down the target program and alter the behaviour of the program. Delay in the target program execution due to probes might alternate behaviour of the concurrent program

Motivation to conduct such a work is to study the feasibility of the application of formal methods in the runtime analysis of MT java application.

The paper is organized as follows. We discuss related work in section 2 and 3 and section 4 outline the instrumentation approaches, while section 5 describes the custom based detection approach. Section 6 describes model checking in general and section 7 describes in detail model based trace analysis approach, while in section 8 a detailed comparison is made between two approaches namely custom based detection with model based trace analysis approach. Finally section 9 contains conclusion and a description of future work

## **2 Chapter 2 Java Multithreading**

### **2.1 Introduction to Java Multithreading**

#### **2.1.1 Definition of Java Multithreading and Related Terms**

Multithreading is a way of building applications with multiple threads. Multithreading enables concurrent execution of several threads within the same program. Thus, it is a convenient way to decompose large programs into relatively independent smaller tasks and increase the overall efficiency [MT Java]. Multithreading is a necessity for all but the most trivial programs.

In multithreaded program, each thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently. For example one thread can run the GUI, while a second thread performs some I/O and third performs some calculation [MT Java].

Developing analyses for multithreaded programs can be a challenging task. The primary complication is characterizing the effect of the interactions between threads. The obvious approach of analyzing all interleaving of statements from parallel threads fails because of the resulting exponential analysis times. A central challenge is therefore developing efficient abstractions and analyses that capture the effect of each thread's actions on other parallel threads.

The Java language provides extending support for multithreading.

## 2.1.2 Benefits of writing multithreaded programs

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

Computers with more than one processor offer the potential for enormous application speedup. MT is an efficient way for developers to exploit the parallelism of the hardware. Different threads can run on different processors simultaneously with no special input from user.

- **Increased application throughput**

In single threaded program, when a request for service is made, it must wait till the service is complete, which makes CPU idle. 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

- **Increased application responsiveness**

In the case of single threaded application, where a single thread performs most of the operation. If that one part of that single thread operation is stopped /freezed, then the whole operation administered by that thread is stopped. Such a blocking situation is decreased user responsiveness. To prevent such a blocking situation, multithreaded program comes handy, that is even is one thread is stopped /freezed then other threads can still continue there 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 multi process environment the communication is done thro sockets, pipes

etc, and the same communication can be performed by multiple threads thru shared variables.

- **Efficient use of System Resources**
- **One binary that runs well on both uniprocessors and multiprocessors**

### **2.1.3 Java implementation of Multithreading**

A *thread* is a stream of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

Threads functionality is implemented in Java using the class `java.lang.Thread` and there are two ways to create a new thread of execution. One is to declare a class to be subclass of `Thread`. This sub class should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started. Another is that another thread can be created, which implements the *Runnable* interface. That class then implements the `run` method. 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 Run a Runnable in a Thread**

## **2.2 Related Terms of Multithreading**

### **2.2.1 Locks**

Multithreaded application use locks to synchronize and communicate their behaviour to one another. To prevent the multiple access condition the threads acquire or release the lock before accessing the shared resource [IBM01]. Lock around shared variable allows the java threads to easily synchronize and communicate. The threads that holds the lock on an object knows that and will not allow another 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 finish his work and release the lock.

Thread that accepts to acquire the lock in use, got to sleep until the thread holding the lock release it, when the lock is release the threads sleeping wakes up and moves to ready-to-run queue.

In java programming, each object has a lock; a thread can acquire the lock using the synchronized keyword.

### 2.2.1.1 Thread Synchronization in Java

Java offers a concept called *monitors* to prevent that two threads access the same resource at the same time. A monitor is a programming language construct providing abstract data types and mutually exclusive access to a set of procedures. In Java, a statement block, method or class can be declared `synchronized`. In java programming, each object has a lock; a thread can acquire the lock using the `synchronized` keyword. 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`.

By calling `o.wait` the current thread temporarily releases the locks it holds on `o` and is added to the wait set of `o`. It is suspended until another thread calls `o.notify`, `o.notifyAll` or if 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. As it is not guaranteed that the condition has been met, `wait` is often called within a `while` loop.

Note that if `t` is a `Thread` object, then calling `t.wait` does not necessarily suspend the execution of `t`. Instead, the *current thread* is suspended until another thread calls `t.notify` or `t.notifyAll`. Only in the case that `t` is the current thread, calling `t.wait` is equivalent to `this.wait` and `t` is suspended by itself.

`o.notifyAll` wakes up all threads in the wait set of `o`. It is used when it cannot be guaranteed that each thread in the wait set of `o` can continue execution.



## 2.3 Multithreaded Problems

Multithreaded programming cause many problems to developers, in many cases the developers are falling prey to incorrect application behaviour or deadlocked conditions.

Here we will discuss the common multithreaded problems and the solution to the common pitfalls.

### 2.3.1 Common Locking Problems

The use of lock, brings with it many problems, here we discuss the problems and there solutions.

Deadlock: Deadlock conditions, arises when one shared variable in already locked and another threads tries to access this resource again.

```
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 [Art01]**

For proving the absence of deadlock, lockgraph is examined which shows the order in which thread acquires the lock. As depicted above in lockgraph, Thread1 first acquires the lock for the object “A” and then acquires the lock for object “B”. While

simultaneously Thread 2 is require to acquire lock for object “B” and then the lock for object “C”. Such a case will lead to deadlock condition. It is thus seen that absence of a loop in the lockgraph guarantee the absence of a deadlock.

Broadly, deadlock could results because of one of the following reasons:

1. Unrelated locks were used to protect a single shared variable
2. Test-and-set primitives were confused with mutexes at the application level
3. Locks were not ever released
4. Threads tried to reacquire locks that they already held

## **2.3.2 Other Multithreaded Problems**

### **2.3.2.1 Race conditions**

Races occur when the several threads access the same resource simultaneously without proper coordination [SBN97] [CS98]. As a result the program might end up producing output far different from the desired one. In 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 access to be simultaneous.

Detection:

For proving the absence of a race condition, a checker examines the *lock set*  $L$ . This is the set of locks held at a certain time, by each thread when accessing a field. A checker has to ensure that a field is:

- 1) Only read when a thread holds at least one lock in  $L_f$  and
- 2) Only written when a thread holds all locks in  $L_f$ .

The most common cause of data races can be compiled because of the following reasons

- When a shared variable was not protected by a lock
- Data race can arise when a lock was not acquired to protect an access even though one existed. Most frequently this happened when a lock was acquired outside a loop, but released within
- Data race can also arise from accidental sharing: one group made what should have been an automatic variable

### **2.3.2.2 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 [BAU03].

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 [MAG99]. 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.

### 2.3.2.3 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 [Ald99]. 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.

The following is a list of the most common examples of overusing monitors for synchronization in multithreaded Java applications [ALD99]:

- Reentrant monitors
- Enclosed monitor
- Thread-Local monitor
- Read-only methods.

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.

## 2.4 Antipatterns-based Detection Approach

The concept of patterns has been widely used in software design and development. In other word a pattern is “a consistent, characteristic form, style, or method” [Eng00]. A general characterises of these patterns are

- When developers write a code, they usually follow some pattern. The pattern followed is derived from there previous experience
- Some developers follow the same pattern

- Some patterns could lead to success and some bad patterns could lead to failure
- Usually patterns exist within a small amount of time and space
- Patterns instance are recognizable

*Design Patterns* are often used in software design and development particularly in the object-oriented design and development, it offers timeless and elegant solutions to common problems in software design. Its usage helps in saving the software production and maintenance cost. It describes patterns for managing object creation, composing objects into larger structures, and coordinating control flow between objects.

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.

The notion of an antipattern can be stated as “something that looks like a good idea, but which backfires badly when applied” [AntiPattern]

Two types of antipatterns are identified so far:

### **2.4.1 Design antipatterns**

Common software designs that have been proven to occur repeatedly, i.e., models of syntactic constructions (in particular) representing potential or confirmed sources of problems in a program [SMI00]. From this viewpoint, antipattern solutions mostly generate mostly negative consequences. Antipatterns are useful for refactoring, migration, update and reengineering.

## **2.4.2 Errors or Bug Patterns**

A bug pattern is a pattern which again and again leads to errors/faults in software applications. In the multithreaded context we view that bug patterns which could lead to MT problems e.g. deadlock, livelocks and race conditions as discussed before. An Bug which repeats itself over and over again the in the java program can be classified as bug patterns

## **2.5 Antipattern Library**

Here we present the antipatterns, we have catalogued so far in our library classified in their corresponding groups. We also report, for each class of antipatterns, our experience in using the antipatterns. We have identified 38 different antipatterns that relates to concurrency, synchronization and other common multithreaded java problems. This was a collaborative work of CRIM and Sap Labs Canada which aims at finding appropriate techniques and tools to analyze Java multithreaded programs developed in Eclipse development environment.

### **2.5.1 Classification of Antipatterns**

Antipatterns can be classified into the following categories [TR1]. This classification is based on the MT problems the antipatterns address.

1. Deadlocks,
2. Livelocks,
3. Race Conditions,
4. Efficiency Problems,
5. Quality and Style Problems.

A particular antipattern may belong to several categories. At the same time, some antipatterns might not be the elements of any of the above categories, i.e., they lead to unpredictable consequences in the application. Such antipatterns will be put into a sixth category, problems with unpredictable consequences, which we introduce here.

#### 6. Problems with unpredictable consequences.

It is important to note that program analysis does not reduce to antipattern detection.

Actually, we believe the process of antipattern detection should be followed by checking whether the application possesses some user-defined features; which, generally, relate to the functional requirements of the application. In this regard, the work could be steered in the direction of devising sound detection techniques and building libraries of user-defined properties to be verified in a MT application. In particular, we can build on the existing results in this domain, mainly on our previous work [...] as well the work by Dwyer et al. [Dwy], who define a library of generic patterns that could be instantiated in a wide variety of specific applications.

### **2.5.2 Antipattern Template:**

To archive the antipatterns in our library, we defined the following template.

We propose the following template 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, potential conflicts of applying the solution, possible detection technique(s), and some comments.

**Table 2.1: Antipattern Template**

<i>Name</i>	A concise definition of the problem.
<i>Description</i>	The situations in which this problem could appear. The effects it has on the code and the application.
<i>Category</i>	Deadlock, Livelock, Race Condition, Efficiency problem, Quality and Style Problem, Problem with unpredictable consequences.
<i>Example</i>	If available, sample code where the problem is illustrated.
<i>Detection</i>	How to detect the problem in the Java code. A high level description of the proposed algorithm to be used in the detection process.
<i>Re-Factoring</i>	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.
<i>Comments</i>	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 will be give example of few antipatterns.

### **2.5.3 Antipatterns:**

Some would argue that an antipattern library could never be completed. Actually, as long as antipatterns are related to programming styles (which are not well defined themselves), one may keep coming up with new additions to a library. Here we list few antipatterns and then make the summary of the most commonly considered antipatterns in the literature and the analysis tool, by which it was detected.



## Synchronized method call in cycle of lock graph

Name	Synchronized method call in cycle of lock graph
Description	When a synchronized method makes part of a cycle in the lock dependency graph, the cycle could lead to a deadlock. This antipattern is the result of bad synchronization between threads of an application.
Category	Deadlock
Example	<pre> Public class Deadlock {     Object a = new Object ( );     Object b = new Object ( );      Public void foo ( )     {         synchronized (a)         {             synchronized (b) { }         }     }      public void foo ( )     {         synchronized (a)         {             synchronized (b) { }         }     } } </pre>
Detection	<ol style="list-style-type: none"> <li>1. Compute the lock graph</li> <li>2. Detect cycles in the lock graph</li> <li>3. Identify synchronized methods that make part of the cycles</li> </ol> Detectable by JLint.
Re-Factoring	<p>Solution: Proper reordering of lock acquisition among the threads involved in the deadlock.</p> <p>Conflicts: Reordering the lock acquisition might lead to data races.</p>
Comments	<p>Source: <a href="http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm">http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm</a></p> <p>The detection results highly depend on the expressiveness of the computed lock graph. Moreover, the presence of a cycle in the lock graph is a necessary condition for deadlock, but not a sufficient one. Therefore, there is high risk of numerous false positives.</p>

**Table 2.2: Synchronized method call in cycle of lock graph**

### 2.5.4 Antipattern Detection Summary

The following table 2.3 list most commonly used antipatterns, and for each one shows the corresponding techniques/tool used to detect them.

**Table 2.3: Summary of detection techniques and tools**

Antipattern	Techniques and Tools
Synchronized method call in a cycle of Lock graph	Abstraction-based static analysis Detectable by Jlint

Method call leads to a cycle in lock graph	Abstraction-based static analysis Detectable by Jlint
Cross synchronization	Linear program scanning Abstraction-based static analysis Detectable by Jlint
Overriding a synchronized method	Linear program scanning Detectable by Jlint
A non synchronized method called by more than one thread	Linear program scanning Abstraction-based static analysis. Detectable by Jlint
A non volatile field used by more than one thread	Linear program scanning Detectable by Jlint
Non synchronized run () method	Linear program scanning Abstraction-based static analysis Detectable by Jlint
Overuse of synchronized methods	Linear program scanning Abstraction-based static analysis
Method wait () invoked with another object locked	Linear program scanning Detectable by Jlint
Call sequence to method potentially causing deadlock in wait ()	Linear program scanning Abstraction-based static analysis Detectable by Jlint
<i>Identifier</i> .wait () method called without synchronizing on <i>identifier</i>	Linear program scanning Abstraction-based static analysis Detectable by Jlint
Synchronized read only methods	Abstraction-based static analysis
Internal call of a method	Linear program scanning Abstraction-based static analysis
Locked but not used object	Linear program scanning Abstraction-based static analysis Dynamic analysis
Synchronization abuse	Abstraction-based static analysis
Wait () not in loop	Linear program scanning Detectable by FindBugs
Unconditional wait ()	Linear program scanning Detectable by FindBugs
Unconditional notify () or notifyAll ()	Linear program scanning Detectable by FindBugs
Reference Value is changed when it is used in synchronization block	Linear program scanning Detectable by Jlint
Overthreading	Linear program scanning Dynamic analysis
Blob Thread	Linear program scanning

	Dynamic analysis
Complex computation within an AWT/Swing thread	Dynamic analysis
Misuse of notifyAll()	Linear program scanning Abstraction-based static analysis Dynamic analysis
The double-check locking for synchronized initialization	Linear program scanning Detectable by FindBugs
Synchronized atomic operations	Linear program scanning
Synchronized immutability object	Abstraction-based static analysis
Unnecessary notification	Linear program scanning Dynamic analysis
Double call of the start method of a thread	Linear program scanning Abstraction-based static analysis
Waiting forever	Linear program scanning Dynamic analysis
Unsynchronized spin wait	Linear program scanning Detectable by FindBugs
Start() method in constructor	Linear program scanning Detectable by FindBugs
Get-Set methods with different declaration	Linear program scanning Detectable by FindBugs
Improper method calls	Linear program scanning Detectable by FindBugs
Wait stall	Dynamic Analysis Detectable by JProbe Threadalyzer
Premature join() call	Data flow analysis Dynamic analysis Detectable by FLAVERS
Dead interactions	Data flow analysis Dynamic analysis Detectable by FLAVERS
Join() with immortal thread	Data flow analysis Dynamic analysis Detectable by FLAVERS

## **3 Chapter 3 MT Java Analysis approaches**

### **3.1 Introduction**

In this chapter we discuss tools/techniques to detect the Antipatterns listed before. The tools/technique can be broadly classified as two types: namely dynamic checkers and static checkers.

Dynamic Analysis requires the execution of the program and then analysis the execution trace for the property verification. In static analysis, one does not run the program, but is

based on the analysis of the code (source code or bytecode) and are normally independent of the input order or thread scheduling since the code is analyzed without execution.

### 3.2 Dynamic Analysis

We tried to find the classified antipatterns/properties with both the static checkers and dynamic checker. But we conclude that most of the antipatterns can be detected by dynamic analysis, but dynamic analysis gave significant edge over static analysis in detection of certain antipattern/properties.

The advantages of dynamic analysis [Hav03]

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

At the same time, implementing the dynamic analysis approach faces a number of challenges, among which are [Hav03]

- Observation of a program behavior requires special efforts (instrumentation);
- Instrumentation of points of observation may cause side effects on behavior and timing characteristics
- The results concern only the executions taken during the observation period and not much can be said about other executions
- Similarly, the results are valid only for a scheduling performed while the program was executed
- Errors are often difficult to reproduce

Before discussing the advantage of dynamic analysis over static analysis, we would like list the most popular dynamic based tool (both commercial and research based) here.

In the following, we provide a list of most common dynamic tools (both commercial and research)

**MaC:** (Monitoring and Checking) is a monitor running systems against a formal specification [ART01]. It was developed by the Real-time systems group (RTG) at the University of Pennsylvania. It combines a high-level requirement specification and a low-level monitoring script that verifies the given requirements at source code level. The requirements are expressed in an extended form of linear temporal logic; the monitoring script is written in a simple event definition language. An *Instrumentor* generates a runtime checker based on the given data. This checker verifies the given properties after each method call. MaC has been applied to a couple of small test programs.

MaC is available for research purposes, including source code. It is written in Java and platform independent. It requires the JTrek library from Digital Equipment Corporation.

The tool is downloadable from <http://www.cis.upenn.edu/~rtg/mac>.

However, MaC does not yet have a framework for systematically testing multi-threaded programs.

**Visual Threads** is a tool created with a purpose to detect concurrency errors in multi-threaded programs, which uses POSIX Threads. Visual Threads is a part of the development toolset of Compaq's Tru64 Unix. Visual Threads is a diagnostic tool used to analyze and refine multi-threaded applications.

It can be used to debug potential thread-related logic problems, such as race conditions and deadlocks that only occur due to slight timing differences. It can also pinpoint bottlenecks and performance problems by using its rule-based analysis and statistics capabilities and visualization techniques. Some support for user-defined rules is provided.

The tool has been used on an experimental OS kernel, the AltaVista indexing engine, and a couple of other projects.

Visual Threads is available for Tru64 Unix system under Developers Toolkit license. The OpenVMS, Linux and HP-UX downloads are free (evaluation license), however, Linux and HP-UX versions offer a limited Java support.

The tool is downloadable from

<http://www5.compaq.com/products/software/visualthreads/>

**Java PathExplorer** (JPaX) is developed by the Automated Software Engineering Group at NASA Ames Research Center; main authors are Klaus Havelund and Grigore Rosu.

Java PathExplorer is a tool for monitoring the temporal behavior and finding concurrency faults (such as deadlocks and data races). The tool facilitates automated instrumentation of a program's bytecode, which then emits events to an observer during its execution.

The observer checks the events against user provided 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. It can be used during the development process to provide verification and can also be used during operations to further optimize systems as they mature. It also

includes automated test-case generation as well as automated generation of assertions and properties corresponding to test cases.

JPaX consists of the three modules:

1. Instrumentation Module – performs a script-driven automated instrumentation of the program to be verified.

The Java byte code instrumentation in JPaX is performed using JTrek a Java byte code engineering tool from Digital.

JTrek developed at Digital Equipment Corporation consists of the *Trek* class library that provides features to examine and modify Java class files. It also includes a set of console applications based on the *Trek* library.

JTrek reads Java class files (bytecode files), traverse them as abstract syntax tree and insert new code in highly flexible manner. The inserted code can access the contents of the method call-time stack at run-time and thus giving access to information needed in the analysis [Hav01b]. The extracted information is transmitted in the events. The observer receives the events and dispatches them to a set of observer rules, each rule performing a particular analysis.

2. Observer Module – performs two kinds of verification.

Logic-based monitoring module checks execution events against a user-provided requirement specification. These specifications are defined in Maude, a modularized specification and verification system. JPaX supports linear temporal logic (LTL), both



future time and past time. Future time LTL provides execution traces as models making it convenient for program monitoring. Past time is useful for verification of safety properties.

Similarly to Visual Threads, the tool performs error pattern detection, namely, prediction of deadlocks by lock graph analysis and race conditions with a modification of the Eraser algorithms.

Error pattern analysis explores an execution trace and detects potential problems such as error-prone programming techniques like locking practices that may lead to data races and/or deadlocks. The important and appealing aspect of error pattern analysis algorithms is that they find error potentials even in the case where errors do not explicitly occur in the examined execution trace. At the same time, the tool may generate false positives.

JPaX contains two algorithms focusing on concurrency errors: a data race analysis algorithm “Eraser” [Sav97] and a deadlock analysis algorithm.

3. Interconnection Module - receives information about potential errors and transmits them to the observation module

**Usage:** An initial prototype of this tool has been applied to two major case studies, the K9 rover developed at NASA Ames Research Center and the Deep-Space 1 attitude control system. The main challenge of the tool is an overhead to the normal execution of programs created by monitoring, and of course, the problem of the false positives.

To the best of our knowledge this tool is not available for download from the web.

**JMPaX** analyzes a multithreaded program against the safety properties expressed using temporal logic [Sen03]. 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.

The user of JMPaX specifies the safety properties of his interest, using a past time temporal logic, regarding the global state of the multithreaded program (which is assumed 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 [SEN03]. 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.

The instrumentation module uses BCEL [DAH01] Java library to modify Java class file. The BCEL library is used to get a better handle for a Java class file. It enables to insert

vector clocks as static member fields in a class, which is otherwise not possible with the tool JTreK (an instrumentation tool used in JPaX).

**Usage:** This tool is intended for use on real-world NASA-related large applications.

The tool is available for download from <http://fsl.cs.uiuc.edu/jmpax/>.

## 3.3 STATIC ANALYSIS

### 3.3.1 Definition

*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 are 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 domain [NASA02] [HAL04]. 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.

In addition to errors, discussed before, static analysis can also detect the following type of errors [NASA02]:

1. Attempt to read a non-initialized variable – read access to non-initialized data may cause non-determinism. Static Analysis tools locate code sections using data that is not initialized.
2. Access conflicts for unprotected shared data

3. Referencing through null or out-of-bound pointers
4. Out of bound arrays - out of bound array errors occur when an index goes outside the range of an array. Static Analysis checks whether the loop incrementing the index can exceed the array size.
5. Division by zero – Static Analysis can check that a division equation is properly coded with if statements to prevent the denominator from equalling zero. It can also provide a list of possible denominator variables to check the equation.
6. Invalid arithmetic operations (square root of negative number) – Arithmetical exceptions caused by procedural entities like modulo computation, square root and logarithm can be checked using Static Analysis.
7. Overflow, underflow of arithmetic operations for integers and floating-point numbers – overflow and underflow occur in numerical computation when a result is not compatible with the variable that stores it. Therefore, it cannot be represented in memory. Static Analysis locates and reports these problems.
8. Unreachable (dead) code – Static Analysis can locate and report codes segments that are never executed. For example: if statement never executed because its condition is never met.
9. Illegal type conversion – occurs when a result does not match its assignment
10. Unpredictable behaviour of multi-threaded applications with shared data. Depending upon the order in which threads read and update shared data, different results can occur for the same input

### 3.3.2 Benefits

Verification can begin earlier in the Software Life Cycle resulting in early detection/resolution of problems and thus reduction in development cost [NASA02].

### 3.3.3 Challenges

The biggest challenge for Static Analysis is generation of false positives sometimes due to overapproximation [NASA02]. However, while the number of false positives may seem large in some cases, subsequent errors can be the result of an initial or upstream error. Correcting this error can eliminate some false positives.

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

Here we provide a short summary of the several tools that rely on static analysis

#### **Tool: Bandera**

**Purpose:** Build a model suitable for model checkers from Java source code [ART01].

**Producer:** Laboratory for Specification, Analysis, and Transformation of Software in the CIS Department at Kansas State University.

**Technologies:** Program slicing, program abstraction, static model checking; two-way conversion between abstraction levels.

**Overview:** Bandera tries to bridge the gap between software source code and an abstract representation of it. A special annotation language allows to express assertions and temporal or quantified properties in the source code. *Predicate definitions* for each method are used in *property specifications* which contain the program properties (invariants or sequences of states through which the program always has to go). Using program analysis (slicing), the first stage of Bandera generates a simplified version of the program, containing only the statements of interest for the correctness of the program. This can drastically cut down the complexity of the model that is generated from the program.

The second stage reduces the model size further via data abstraction. It generates an intermediate representation of a finite-state model in an intermediate format.

This format is then translated into the specification language of a model checker of choice; so far, SPIN is supported. Translators for the Symbolic Model Verifier (SMV), developed in the Carnegie Mellon University, and Stanford's forthcoming SAL model checker are under construction.

A newer component is the *counter-example generator* that checks faults found in the abstract model for their validity in the actual program, and reports where in the source code the fault was found.

**Availability:** Available for free download from <http://bandera.projects.cis.ksu.edu/>

**Usage:** Bandera has been applied, in conjunction with JPF, to a couple of small programs, including Doug Lea's concurrency package.

**Tool:** ESC/Java

**Purpose:** Detect common programming errors at compile-time [Art01].

**Producer:** Compaq Systems Research Center

**Technologies:** Generator of background predicates and verification conditions, simplify theorem prover

**Overview:** The “Extended Static Checker” for Java has been developed by Digital Equipment Inc. (now part of Compaq). The first version has been written for checking Modula-3 programs. ESC/Java statically checks a program for null reference errors, array bounds errors, potentially incorrect type casts and race conditions.

ESC/Java requires annotations in the source code in its own annotation language. In an internal study, the annotation overhead in the source code was about 13.6% [38].

However, less scrupulous annotations can be made, ignoring certain types of faults.

The checker first generates type-specific *background predicates* to encode data types and type relations for each class and interface. Then, each routine is translated into a *verification condition*. As an intermediate step, a sequence of commands similar to Dijkstra’s guarded commands is produced. The Simplify theorem prover then tries to disprove each one of these verification conditions. If it succeeds, the front end transforms the *counter-example context* into a warning and (optionally) a counter-example [36].

**Availability:** The checker has recently been released and is freely available for research and educational use. A binary version can be downloaded for Alpha Unix, Solaris, Linux and Windows 9x/NT. The front end has been written in Java while the theorem prover Simplify is written in Modula-3. – A Modula-3 front end is also available, but for Alpha Unix and Intel Windows 9x/NT only.

**Usage:** ESC/Modula has successfully found fault in several small projects, being totally 20 K LOC in size. There are no numbers available yet for ESC/Java.

**URL:** <http://research.compaq.com/SRC/esc/>

**Tool: Jlint**

**Purpose:** Semantic verifier detecting certain deadlocks, race conditions and a few other faults [Art01].

**Producer:** Moscow State University, Research Computer Center; main author: Konstantin Knizhnik.

**Technology:** Control flow/lock dependency analysis, specialized checks for other faults.

**Overview:** Jlint comes as two programs, a simple syntax verifier (AntiC) and a semantic verifier (Jlint). The former checks for a few common potential syntax errors. The latter is much more interesting, for it extracts information from (non-annotated, normally compiled) Java class files and performs consistency and flow analyses on them. Jlint is capable of dealing with missing debugging information which some Java compilers cannot (yet) generate. It also allows a hierarchical selection of the checks that should be performed.

The core algorithm checks Java class files for loops in the lock dependency graph. This graph includes both static and dynamic 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.

**Availability:** Freely available for download at <http://artho.com/jlint/> ; written in C and C++, and should work on any platform.

**Usage:** No other numbers are available, but Jlint has been applied successfully at Trilogy to large scale software (several projects of several ten thousand LOC each).

**URL:** <http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm>

**Tool:** JPF

**Purpose:** Integrate model checking, analysis and testing.

**Producer:** Automated Software Engineering Group (ASE) at NASA; main author: Klaus Havelund.

**Technologies:** Slicing, abstraction; 1.0: Java to PROMELA translator; 2.0: special JVM (MC-JVM) and model checker

**Overview:** The “Java PathFinder” has been developed at the Automated Software Engineering (ASE) department at NASA. Currently, JPF can only check invariants and deadlocks. Invariants are given as a Boolean Java method.

After an abstraction and a slicing stage, which both reduce the state space of the program a lot, a depth-first search is performed on the program stages. A special JVM, which allows to move forward and *backward* one state in the bytecode execution, is used for this.

The first version was a translator from Java to PROMELA []. Special assertion and error methods specify the properties to be checked. It has, however, only supported a fairly

restricted subset of Java. Because it was too difficult to extend the program to support more Java constructs, a different approach has been taken for the second version, which works directly on bytecode. It can therefore fully support all Java features [1].

**Availability:** It is not available for free download

**Usage:** JPF has been applied to the Remote Agent Spacecraft Controller (RAX), where it found a deadlock, and the DEOS Avionics Operating System. After the slicing stage, the largest package was 1443 LOC in size [1].

**URL:** <http://ase.arc.nasa.gov/jpf/>

### 3.4 Model Checking

Model Checking - automated technique for verifying finite state concurrent systems. The model checker evaluates the model by beginning with the initial states and repeatedly applying transitions to reach all possible states [NASA02]. In order to explain Model Checking, we define the following terms:

- Formal Model – computer model of system
- State – snapshot of the system that captures values of variables at a particular instant in time
- Transition – the change described by the state before an action occurs and the state after the action occurs

To use a model checker, an engineer must create a model of the system. This model represents valid states and transitions of the system. The model checker evaluates the model beginning with the initial states and repeatedly applying transitions to reach all possible states. If a property violation occurs, the model checker reports the error via an

execution trace called a counterexample. Counterexamples show where the violation occurred.

Model checkers verify properties of dynamic operations using the Temporal Logic formalism. Two kinds of Temporal Logic are described:

- Linear Temporal Logic (LTL) – time is described in a linear fashion with no branching
- Computational Tree Logic (CTL) – time is described in a branching fashion

Three Model Checking tools are:

- SPIN – an explicit model checker that enumerates all individual states to be verified. This approach is limited by the size of the state space of the system. Although SPIN provides many optimizations to cover very large state spaces (millions of states and more), it is unlikely to scale well for very complex models.
- SMV – a symbolic model checker that uses efficient data structures (binary decision diagrams, or BDDs) to represent and process sets of states in a single operation. The symbolic processing allows SMV to explore much larger state spaces than explicit state tools such as SPIN. However, SMV is still limited by the complexity of the generated BDD structures, which can vary wildly and are hard to optimize.
- UPPAAL - a toolbox for validation and verification of real-time systems described as networks of timed automata

### **3.4.1 Benefits**

1. 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 nondeterministic system in a plausible time frame for software development [NASA02].

2. Model checker can backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between tests required in traditional scenario based testing.

It stores and compares states to detect those already explored, thus exploring all states exactly once even in the case of looping executions.

3. Detects problems in the early stages of development; thereby greatly reducing overall development costs.

### **3.4.2 Challenges**

There are two challenges associated with model checking [NASA02]:

- Models must be translated into special model checking language like PROMELA (for SPIN) or SMV
- State space explosion – Because of the way software components interact with each other and because data structures can have different values, it is common for a model checker to run out of memory before exploring the entire state space.

The following tools/techniques mitigate state space explosion [NASA02]:

1. Symbolic model checkers, like SMV, offer a technique for mitigating state space explosion. Instead of generating and exploring every state like explicit model checkers, symbolic model checkers manipulate whole sets of states at a time.
2. A set of states is evaluated for each transition by implicitly representing the states as the logical conditions the states satisfy. Sets of states are encoded as Binary Decision

Diagrams (BDDs). BDDs are a special representation for Boolean formulas that is often more compact than

### **3.5 Theorem Proving**

*Theorem Proving* – use of logical induction over the execution steps of the program to prove system requirements. In other words, system requirements can be translated into complex mathematical equations and solved by verification experts. Solving these equations proves that the system is accurate [NASA02].

#### **3.5.1 Benefits**

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

#### **3.5.2 Challenges**

Requires significant effort and expertise making Theorem Proving suitable for analysis of smallscale

## **4 Chapter 4 Java Trace Collection**

### **4.1 Introduction**

Here we discuss advantages and drawbacks of existing trace collection approaches and tools. Based on this comparison we outline our approach, which combines the best features of custom bytecode instrumentation and of an industrial strength profiler.

### **4.2 Instrumentation Approaches**

As discussed before, that for the runtime time analysis of the program, we need to extract the relevant events and then analyze the properties in question, based on the information provided by the extracted events.

The event collection could be performed (with an added instrumentation) on the following levels:

- Operating system
- Java Virtual Machine (we include standard instrumentations such as profiling and debugging services in this category)
- Source code or
- Compiled code (bytecode)

Often, instrumentation is not a part of the system design, but is added to existing systems.

An *instrumentor* [KIM01] is a tool that receives as input the program (source code or bytecode) and *instruments* it, at different locations with additional statements for monitoring purposes.

During the execution of the program, the instructions embedded by the instrumentor are executed. The relevant execution events generated by instrumentation are saved in a file.

The JVM instrumentation is the least portable, since JVM implementation of different producers could vary. Since bytecode is simpler than Java, instrumentation at the bytecode level is easier than at source code level, and is, therefore, most common [KIM01].

Now we will briefly discuss the different instrumentation approach in detail.

#### **4.2.1 Custom JVM Instrumentation**

Java Virtual Machine (JVM) instrumentation consists in 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.

At the same time, custom JVM level instrumentation suffers from the following disadvantages [KIM01]:

- The reengineering of an existing JVM requires special efforts. The process could be error-prone without deep knowledge of the application.
- The JVM uses Just-In-Time (JIT) compilation and Hot Spot dynamic compilation [Arm98] for performance enhancement. However, when these features are enabled, simple modification of the bytecode interpreter unit is not sufficient [Kim01]. One has also to modify compilation, inlining, and interpreter units. This increases the complexity of JVM modification.
- JVM has been updated frequently (there have been four major and two minor updates through 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.

Therefore, custom modification/instrumentation of the JVM does not seem to be a practical solution.

To alleviate the above-mentioned difficulties of custom instrumentation, modern JVMs 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 use of the standard debugging and standardizing profiling architectures of JVM is discussed below.



## 4.2.2 Java Profiling Interface

The JVMPI is a two-way function call experimental interface between the Java virtual machine and an in-process profiler agent. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, memory allocation, thread start, lock contention etc. On the other hand, the profiler agent issues control 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 agent is provided within Sun SDK since version 1.2; some JVMPI support is provided by other JVM producers.

The possible monitored events 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

Aiming mainly at performance issues, JVMPI does not provide logging of monitor entry/exit and variable access events required for data race detection, as well as

scheduling. In order to increase the level of observed details, different techniques could be applied. For example, a Linux Monitoring Tool used in conjunction with JVMPI in the development of Java Profiling [MEI03] tool is able to observe even the execution thread scheduling.

The JVMPI allows the profiler agent to instrument every class file before it is loaded by the virtual machine. The profiler agent may, for example, insert custom byte code sequence that records method invocations, control flow among the basic blocks, or other operations (such as object creation or monitor operations) performed inside the method body.

Along with event logging, JVMPI provides a sampling mechanism, that gives, on request, the program snapshots. Sampling suits better for performance evaluation purposes; however, it could be employed in correctness problem detection; e.g., deadlocks could be detected with thread/lock dumps.

The JVMPI is used by the profiler agent that runs in the same process as the Java virtual machine. Programmers who write the agent must be careful in dealing with threading and locking issues in order to prevent data corruption and deadlocks.

Events are sent in the same thread where they are generated. For example, a class loading event is sent in the same thread in which the class is loaded. Multiple events may arrive concurrently in different threads. The agent program must, therefore, provide the necessary synchronization in order to avoid data corruption caused by multiple threads updating the same data structure at the same time [JVMPI].

### 4.2.3 Java Debugging Architecture

JPDA 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 [JPDA].

JPDA is somewhat similar to JVMPI, and certain functionalities overlap. However, 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 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

Local variables and arrays are not observed. Often, source code modification is recommended to transform arrays and local variables into observable entities.

Controls over the Virtual Machine allow class redefinition, this makes class instrumentation possible.

Until the SUN SDK version 1.4 a program could only be debugged in interpreter mode.

Since SDK 1.4, the interpreter mode is only used when breakpoints are inserted. Setting a breakpoint only inhibits compilation (full speed execution) for the method containing the breakpoint.

Since JPDA does not track lock accession events [LEE03] extended (instrumented) JPDA of Sun SDK 1.3 for model checking Java.

Another way of overcoming JPDA limitations for lock entry/exit events is single step execution mode and breakpoints. With proper breakpoints and source code analysis, thread synchronization and collaboration events could be traced without intrusive Virtual Machine instrumentation. However, such a solution could be rather sluggish and does not apply to code whose source is not available. Breakpoint debugging works well for programs that do not interact with any other dynamic entities (other programs or real-world devices). However, programs in distributed and real-time domains may have their behavior and results altered if interrupted by a debugger. Events may go undetected, message queues may overflow, and moving parts may fail to stop in time, causing real-world damage to machines or people.

Sun SDK includes a proof-of-concept JPDA based command line debugger, JDB, and a method call tracing tool, Trace.

#### **4.2.4 Java Platform Profiling Architecture of J2SE 5.0**

The J2SE 5.0 (a 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 [J2SE 5.0].

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

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

- **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

- **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

- **Logging Monitoring and Management Interface**

The `java.util.logging.LoggingMXBean` interface enables you to retrieve and set logging information

- **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.

### 4.3 Source Code Level Instrumentation

Source code instrumentation is to add code (particular instructions, packages etc) called probes to report the events in the program to be analyzed [CAI03].

Automatic source-code transformation usually requires parsing and analysis of the abstract syntax tree of an application.

Java language provides already some logging capabilities with the package

*java.util.logging*. The core package includes support for delivering plain text or XML-formatted log records to memory, output streams, consoles, files, and sockets. In addition, the logging APIs are capable of interacting with logging services that already exist on the host operating system.

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.

Among the disadvantages of source code instrumentation is the need for source code that is not always available.

### 4.4 Java Bytecode

Before discussing the Bytecode instrumentation, we give a brief introduction about the Java Bytecode format and related terms.

### 4.4.1 Java Bytecode Definition

Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs. This knowledge is crucial when debugging and doing performance and memory usage tuning. Knowing the assembler instructions that are generated by the compiler for the source code you write, helps you know how you might code differently to achieve memory or performance goals [HAG01]

### 4.4.2 Java Bytecode Format

Here we give the brief details about the format of classfile and the bytecode instruction. The format of classfiles and the byte code instruction set are described in more detail in the Java Virtual Machine specification [JVM].

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* [DAH01]. Attributes are a way of putting additional, user-defined information into class file data structures. For example, a custom class loader may evaluate such attribute data in order to perform its transformations.

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

```
getstatic      java.lang.System.out
ldc           “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)

### **4.4.3 Bytecode Instrumentation**

#### **4.4.3.1 Overview**

The Java compiler converts the Java source code into the class file format [JVM]. Instead of the source code, the resulting Java bytecode is modified. The Java bytecode is a stack based programming language. One can modify the bytecode without changing the semantics of the program, if he introduces statements into the bytecode that have no effect on the stack. Actions, which perform loading of values onto the operand stack, such as `iload` and `getField`, are followed by operations, which indicate the reference of the variable. In this case, the inserted code must not affect the operand stack. The generation of the modified bytecode can be performed either by a customized compiler or by using an existing compiler and modifying its resulting bytecode

#### **4.4.3.2 Supported Events**

Bytecode instrumentation can guarantee to fully track the access and/or modification of variables. It is possible to instrument all Java class files with probes, so libraries and third party components can be instrumented. This, however, will result in anomalies being reported for code beyond the user control.



One could distinguish dynamic and static instrumentation of bytecode. Dynamic instrumentation is performed during program execution, while static instrumentation is performed prior the execution.

Bytecode instrumentation offers numerous advantages:

- 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.
- Many of the issues of interest for run-time monitoring (actual access to variables, power consumption of instructions) are revealed precisely at the bytecode level.
- Java bytecode prohibits pointer arithmetic, which enables the detection of the updating of variables and also it is strongly typed.
- Bytecode instrumentation adds the least overhead to a Java programs execution.
- Java Bytecode is platform independent.
- 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.
- 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.

## 4.5 Instrumentation Tools

In this section, we discuss tools designated for event collection as well as event collection parts of tools for profiling and trace analysis, and general-purpose Java instrumentation tools.

### 4.5.1 Java Virtual Machine Instrumentation Based Tools

#### 4.5.1.1 Tool: JinSight

**Source:** IBM AlphaWorks

**Overview:** JinSight 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.

JinSight consists of two parts:

- Instrumented Java Virtual Machine, which generates trace data 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 details are particularly useful since tracing every detail of a program's execution will generate trace information rapidly (30MB/min); the resulting traces quickly running into the hundreds of megabytes and more.

- JinSight visualizer, which reads the trace data and presents graphical views of program execution, recurring method call patterns, object interconnection, call graph etc.

The latest version of JinSight (for Java 2) is based on JVMPI platform.

**Availability:** Only Windows 95/NT and AIX are supported. The tool is free for download (90 days evaluation period) from IBM.

<http://www.alphaworks.ibm.com/tech/jinsight> .

## **4.5.2 Profiling Interface Based Tools**

### **4.5.2.1 Tool: Hyades**

**Producer:** The tool is an Eclipse project.

**Overview:** The Hyades 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 provides an extensible framework and infrastructure that embraces 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.

Since Hyades launch in December 2002, its development has been actively supported by IBM, Parasoft, Telelogic, and Scapa Technologies.

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 can be stored in XML compliant files, in the following fashion:

The above fragment of the trace produced by an instrumented SAP vending machine server shows that along with detailed events of class definitions, object allocations, method enter/exit, the trace presents information on the used data collection options and filters. Hyades also provides statistics and graphical views on the results of profiling.

The instruments to create Analysis Engines and Log Analyzers are provided.

**Availability:** The platform is downloadable for free from <http://hyades.eclipse.org> and covered by Common Public License.

Among other JVMPI based tools, we could mention the free method call profiler EJP (<http://ejp.sourceforge.net/>), which presents the method call trace in an hierarchic form with the method execution time.



### 4.5.3 Tools for Source-Code Level Instrumentation

#### 4.5.3.1 Tool: JavaScope

**Producer:** Sun Microsystems.

**Overview:** 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.

**Availability:** Unavailable for download.

Some tools, like the open-source (LGPL) tool RECODER automate a general source code transformation. However, a complete automation of software changes is far beyond today's possibilities: In general, it is impossible to do certain design decisions automatically. For example, in general, it is not possible to analyze the behavior of reflective programs (analyzing reflection requires value analysis, which is not always computable). Therefore, no guarantees can be given that the observable behavior of reflective programs is retained.

#### **4.5.4 Java Bytecode Instrumentation Tools/Toolkits**

Tools described here provide an ability to modify Java bytecode. While some tools are ready to perform instrumentation (e.g., ProbeMeister), others are tools or libraries, which could be used to implement a custom instrumentation tool.

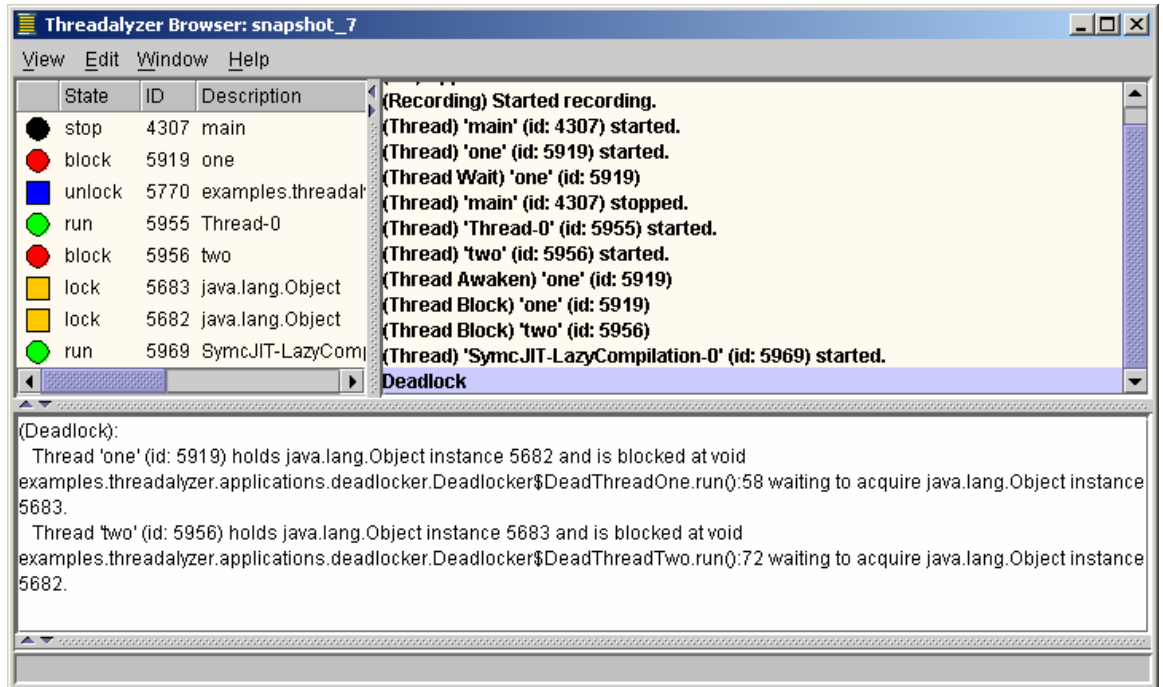
##### **4.5.4.1 Tool: JProbe Threadalyzer**

**Producer:** Quest Software

**Overview:** JProbe Threadalyzer detects thread problems that can threaten the application performance. 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.



**Figure 4.1: A trace snapshot of JProbe**

Here, we discuss the thread stall and data races that JProbe Threadalyzer reports.

JProbe Threadalyzer reports the following possible thread stalls:

- A thread blocks, waiting to acquire a lock and does not acquire the lock within a user-defined time;
- A thread waits (*wait()* is called with no timeout value) for a *notify()* event that is not sent within a user defined time.

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. This is done in the Threadalyzer tab of the Run Settings dialog.

Threadalyzer identifies and flags any data races it encounters while running the program. The data race detection process is usually resource intensive and may slow down Threadalyzer performance.

Powerful lock analyzers help in identifying thread problems before they happen.

JProbe supports Java 2 broad platform and Eclipse.

**Availability:** It is available online for free download at

“<http://www.quest.com/solutions/download.asp>” (trial version only).

#### **4.5.4.2 Tool: ByteCode Instrumentation Tool (BIT)**

**Authors:** Han Lee and Ben Zorn.

**Overview:** Bytecode Instrumenting Tool (BIT) is a collection of Java classes for building tools to instrument Java Virtual Machine (JVM) bytecodes [LZ97]. BIT employs an instrumented program-based technique for extracting dynamic behavior of Java Virtual Machine (JVM) bytecodes. BIT allows the user to insert calls to analysis methods anywhere in the bytecode, so that information can be extracted from the user program while it is executed. This information, in turn could be used in performance measurement and optimization. BIT is an effective framework for understanding a dynamic behavior of JVM bytecodes.

The architecture of BIT is based on the observation that many of the dynamic behaviors of a program can be obtained by instrumenting a few key locations, e.g., before and after methods, before and after basic blocks, and before and after instructions. Thus, BIT provides classes and methods for inserting a method invocation at each of these key locations. BIT uses an internal representation of the bytecode, to which modification can be made and then written back to a class file.

BIT consists of two Java packages. One for performing low-level operations such as reading and writing class files, interpreting constant pool entries, reading code buffers, and other low-level class file parsing. Another package is used for performing higher-level operations such as finding and constructing basic blocks, decoding instructions from the code buffer, inserting method calls, and navigating through higher intermediate representations. The first package provides a low-level representation of a class file while the second package provides higher-level functionality.

BIT is implemented using the Java programming language. BIT consists of 43 Java classes. Various program analysis tools for measuring and understanding the dynamic behavior of the program can be built using BIT classes

**Availability:** It is available for download at

<http://www.cs.colorado.edu/~hanlee/BIT/index.html>

#### **4.5.4.3 Tool: JTrek**

**Producer:** Digital Corporation.

**Overview:** 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.

**Usage:** 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.

**Availability:** It is available for free download at “<http://www.cis.upenn.edu/~rtg/mac/>”

#### **4.5.4.4 ToolKit: Byte Code Engineering Library (BCEL)**

**Overview:** The Byte Code Engineering Library (formerly known as JavaClass) is a toolkit for the static analysis and dynamic transformation of Java class files [DAH01]. 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.

BCEL consists of several components such as:

- `org.apache.bcel.util.Class2HTML`, which generates nice HTML pages for a class

- `org.apache.bcel.util.JavaWrapper`, which replaces standard java interpreter to use own class loader to let one modify a loaded class via `modifyClass` method that can be overloaded
- `org.apache.bcel.verifier.GraphicalVerifier`, which allows to verify the class file with verbose output

**Usage:** Java MultiPathExplorer (JMPaX), a trace verification tool, uses 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.

**Availability:** It is available for free download at <http://jakarta.apache.org/bcel/> under Apache Software License (open source).

#### 4.5.4.5 Tool: JIAPI: Java Instrumentation API

**Producer:** Open source project.

**Overview:** A high level API to instrument Java bytecode based on BCEL.

The goal of the project is to provide:

- A framework to implement instruments which manipulate Java bytecode
- Implementations for common bytecode manipulations
- A abstraction of some details in bytecode structure
- Easy to use event-based API and run-time hooks
- Instrumentation configuration tool
- ClassLoaders utilizing bytecode instrumentation
- Class loading plugins for application servers

The event model includes

- Method exit/entry
- Field access
- Exception throw

**Availability:** Available for free download from <http://jiapi.sourceforge.net/> under lesser Public GNU License

**Table 4.1: The summary of the analyzed instrumentation tools**

<b>Tool</b>	<b>Availability</b>	<b>Source</b>	<b>Byte/Source code/JVM level</b>	<b>Usage</b>
BIT	Free for Download	By Han Lee and Ben Zorn at University of Colorado	Bytecode	Used for research purposes and making customized tools
Jikes	Download for free (90 days evaluation period)	IBM Alpha Works	Bytecode	Development of tools for customized instrumentation
JOIE	Free for download (beta version)	Duke University, USA	Bytecode	Used in projects like Ivory, Saffasi and Naccio
Hyades	Free download	Eclipse Project	JVMPI	
JProbe Threadalyzer	Download for Free (trial version)	Quest Software	Bytecode	Used by SAP
Jiapi	Free	Public project	Bytecode	
BCEL	Free for download, Apache open source	Apache Open Source Community	Bytecode	Used in JMPaX, JContractor, JRaT
Omniscient debugger	Free download	Bil Lewis	Bytecode	Used to debug its own code
ProbeMeister	Free for non commercial purpose only	Object services & Consulting, MD	Bytecode	To monitor distributed applications
CFParse	Download for free (90 days Evaluation period)	IBM Alpha works	Bytecode	Used for debugging service and controlling the functionality of a applet
AspectJ	Free for	Tool available	Source code	Custom logging

	Download	at Mozilla Public license		
JTreK	Not available for download	Hewlett Packard	Bytecode	Used in JPaX and Java.Mac
JSpy	Not available for download	NASA Ames Research Center	Bytecode	To be used in JPaX

## 4.6 Our Instrumentation and trace collection Approach

### 4.6.1 Introduction

In our runtime analysis approach, for instrumentation purposes we use integrated approach, combining both the approaches (i.e profile interface based approach – Hyades base and bytecode instrumentation based approach – JTREK). In this way we can use best of both the two approaches. As discuss before, Hyades framework consist of Java profiling interface which provides non-intrusive trace collection. With this approach, events for example such as method entry/exit, trace start/end, exception throw are collected. These events are emitted as XML fragments, when Java profiling agent attaches to JVM to capture and record the Java application behaviour.

The limitation of Hyades based approach is that, we can only collect limited types of events. The events such as monitor enter/exit, variable updates (either write and read) could not be collected, these events we require for monitoring of concurrency errors, such as deadlocks and data races. To improve the Hyades based trace collection approach we supplement it with bytecode instrumentation tool –JTREK.

To record monitorenter and monitorexit events, JTreK instrument the bytecode with empty methods *Object.lockentry* and *Object.lockexit* before the start of synchronized statement. Similarly to record variable updates JTrek instrument the bytecode with

empty methods such as *variable\_write* and *variable\_read* before the monitored variable write or read operation.

When this instrumented bytecode is executed using the Profile interface tool (Hyades framework), the additional events are logged in Java trace (in XML format).

## **4.6.2 Hyades Profiling and Tracing**

The Hyades Profiling and Tracing Tool consist of the Profiling and Logging Perspective. It enables to profile the application, to work with profiling resources, to interact with the application when profiling, and to examine the application for concurrency or memory related problems. The Profiling Tool collects data related to the Java program's run-time behaviour. Data collected from a profiling session is saved to a file in *.xml* format for later analysis. To profile an application/program, it needs to be running, an agent (Java profiling agent) needs to be associated with it, and we need to attach to that agent. If the application is already running, we only need to attach to its agent. *Attach* means that a monitor is created to record runtime behaviour of an application. The Java Profiling Agent is an agent instance that is deployed with the Agent Controller. The *Application Process* is the Java Virtual Machine that executes the Java application. If the application is not running, then we need to launch it. *Launch* means that the application is started, an agent is associated with the process, and a monitor is created to record runtime behaviour of an application.

### **4.6.2.1 Event Structure and there Attributes**

Here we briefly discuss the event structure (which is in XML format) of the Java trace.

The data output of the Java Profiling Agent ( Hyades Tracing) is a set of XML elements,

that are either emitted as fragments within a non-XML trace stream or as part of a valid XML document.

The event structure consists of the following elements

- 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

#### ***4.6.2.1.1 IDs***

Attributes of the elements has various kinds of IDs. Threads, classes, methods, and objects each have unique IDs. 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, among other entries, 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.

#### ***4.6.2.1.2 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. 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. Expressed as an unsigned 32-bit value formatted as a string

#### **ThreadId/ThreadIdRef**

ThreadId defines and threadIdRef refers to the thread that the element occurred in.

ThreadId's are unique within the scope of a trace regardless of how many threads are started and ended. 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. Expressed as a 32-bit unsigned value in string format

#### **ObjId/ObjIdRef**

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

#### **ClassId/ClassIdRef**

ClassId defines and classIdRef refers to the class associated with the event. 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

#### **4.6.2.1.3 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>
```

The information conforms to the associated DTD (trace.dtd) and schema (trace.xsd), which declare these elements such as TRACE, code, tag and methodbody.

#### ***4.6.2.1.4 Trace Behaviour Elements***

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

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

#### ***4.6.2.1.5 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
```

#### ***4.6.2.1.6 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.

#### ***4.6.2.1.7 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.

#### ***4.6.2.1.8 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 irregardless 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.

### **4.6.3 Bytecode Instrumentation using JTrek**

Here we describe in detail the bytecode instrumentation, using JTrek.

An executable Java program consists of a set of classfiles. One classfile contains definition of one class. A Java classfile is loaded into running Java virtual machine at run-time. Thus classfiles are linked dynamically at run-time rather than statically. 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 the instrumentor.

A instrumentor takes two inputs namely, Java classfile (\*.class) and instrumentation script, which contain information such as monitored variables/methods. Based on these two inputs, the instrumentor inserts methods into the target program. The Java bytecode instrumentation is performed using the JTrek Java bytecode engineering tool [JTrek].

JTrek makes easily possible to process Java class files, examination of their contents and insertion of new code. The inserted code access the contents of various runtime data structures, such as the call-time stack, and when executed (either on Hyades framework/platform) emit events carrying this extracted information to the trace or console output.

The process works as follows, 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. For each class, and for each method in that class, JTrek iterates through the bytecode. At each byte code, 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, each targeting a particular Java construct that can be accessed from an instruction, such as for example Statement and Method, with each method containing various kind of information, that is either integers or strings, or other objects that can be used for further navigation. 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. Further in the Statement class in turn contains a method, Method `getMethod()`, returning the method in which the statement occurs.

As an example, the method in which an instruction `instr` occurs can be obtained by the expression: `instr.getStatement().getMethod()`.

In the *void at (Instruction instr)* overridden method, a switch-statement branches out depending on what is the opcode of the instruction. In case an instruction is for instrumentation, JTrek inserts the call of a method either after or before the instruction.

For each kind of bytecode that we want to instrument we have defined a class that contains essentially one methods: `void instrument (Instruction instr)`, which performs the required instrumentation by inserting a call to the second method.

### **4.6.3.1 Additional Events logged using the bytecode Instrumentation:**

Here we list the additional events logged in the Java trace, and how we precisely did the instrumentation:

#### ***4.6.3.1.1 Variable Updates***

For data race analysis, we need to extract additional information, such as when and which variables are accessed and also their updated values. Further we also require to extract information as these variables are accessed by which threads.

For example we are interested to monitor a *class variable*  $x$  of type *int* and log the event `<methodEntry variableWrite>` and its (updated) value in Java trace. To achieve this, JTTrek iterates through bytecode and search for `putstatic` instruction (an instruction updating a class variable) which has class variable name, type of the variable and the type of the parent class as operand. The JTTrek iterates using `dotrek` method.

The empty methods such as `variable_write` or `variable_read` are inserted after the `putstatic` instruction. In the operand of the `variable_write` or `variable_read` methods we pass parameters such as variable type and variable value. When the instrumented bytecode is executed on Hyades platform (which contain Java Profiling Agent) the events such as `<method Entry variable_write>` or `<method Entry variable_read >` are logged at the Java trace and their parameters such as variable type, variable value are printed at the console output.

For data race analysis and other concurrency errors analysis we combine both the traces namely the Java trace (in XML format) and the updated value, obtained at the console output.

We monitor two types of variables namely primitive field variable and Local Variables. Below we discuss about there variable and way to instrument it.

#### **4.6.3.1.1.1 Primitive Field Variables**

A field variable is either a class variable or instance variable. A class variable is updated by putstatic instruction. An instance variable is updated by putfield instruction. Both instructions (putstatic and putfields) have variable name and a parent class name as parameters. Both instructions take top stack operand value as updated value of the variable. The instrumentation process is similar as described before that is instrumentor iterates the bytecode and search for putstatic or putfield instruction and inserts instruction/probe to fetch the updated value of the monitored variable from the operand stack during runtime.

#### **4.6.3.1.1.2 Local Variables:**

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

When we recognize that an instruction (for example call it  $i$ ) is recognized as updating the monitored variable, monitorenter instruction is inserted right before  $i$  and monitorexit instruction is inserted right after  $i$  for making the update of the variable.

Then the probe invoking void Variable\_update (Object parentAddress,  $\langle T \rangle$  value, string varName) (parentAddress is an address of an object whose member field varName is monitored) is inserted right before  $i$ .

When the instrumented program is executed on the Hyades platform/framework, the event `<methodEntry Variable_write>` is printed on Java trace it's parameters such as variable (updated) value and variable name is printed to console output.

#### ***4.6.3.1.2 Monitor Enter and Exit***

For monitoring of concurrency errors, such as deadlocks and data races, we need to extract information, as when locks are acquired and released.

At the JVM level, locks are obtained when a synchronized method or monitorenter instruction is executed. As stated in the JVM specification [JVM]: “Normally, a compiler for the Java programming language ensures that the lock operation implemented by a monitorenter instruction executed prior to the execution of the body of the synchronized statement is matched by an unlock operation implemented by a monitorexit instruction whenever the synchronized statement completes, whether completion is normal or abrupt.”

For a monitorenter instruction, which indicate that a thread takes a lock when entering a synchronized statement, we extract information as which object is locked and which thread does it, and similarly we get this information when monitorexit is executed. Thus instrumenting all monitorenter and montitorexit correctly tracks the number of locks held by a thread on an object relating to synchronized statements. This emit these events when executed: *Object.lock(t; o)* (thread *t* locks object *o*) and *Object.unlock(t; o)* (thread *t* unlocks object *o*). When this instrumented bytecode is executed by Hyades Framework, these additional events e.g. `<method entry LockAcquire >` and `< method exit LockRelease>` are logged in the Java trace.

Thus instrumenting all `monitorenter` and `monitorexit` correctly tracks the number of locks held by a thread on an object relating to synchronized statements. However if a synchronized method abruptly terminates, then the lock obtained on entry to the method is released by the JVM, but there is no way to instrument the bytecode to record the release of the lock. The possible fix to this problem is to modify the body of each synchronized method to surround the whole body with a try block that catches all throwables, logs the release of the lock, and re-throws the exception.

Below we show two same methods of the class, the first method contains the instrumented bytecode and the second method contains the uninstrumented bytecode. The instrumented bytecode is shown in bold letter. First is shown the source code, which is then compiled to the bytecode

```
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
    // 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:aload
// 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:aload
// 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 4.3: Sample of instrumented Bytecode

```

private void forksAvailable(int i)
{
// 0 0:aload_0
// 1 1:getfield #36 <Field Object[] convey>
// 2 4:iload_1
// 3 5:aload
// 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:aload
// 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 4.4: *Sample of uninstrumented Bytecode*

This is sample of source code of the instrumentor, to insert the empty method *Object.lockAcquire* after the *monitorenter* instruction. As we see from the sample of instrumented bytecode, instructions such as *aload\_0*, *getfield* <Field Object [] convey>, *iload\_1*, *aaload* and *invokevirtual* <Method void Object.lock1Release ()> are inserted. To insert *aload\_0* (a bytecode instruction), we write the instrumentor code `code.append(42)`, which inserts the bytecode instruction “*aload\_0*” where opcode “42” refers to *aload\_0*. Similarly to insert the *getfield* <Field Object [] convey > we write instrumentor code such as `code.append(180, filterLock)` where the opcode “180” refers to instruction “*getfield*” and *filterLock* refers to <Field Object [] convey>. Similarly to insert the *invokevirtual* <Method void Object.lock1Release we write instrumentor code such as `code.append(182, monitorenter2)` where the opcode “182” refers to instruction “*invokevirtual*” and *monitorenter2* refers to the < Method void Object.lock1Release>.

**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 the Instrumentor code

Below is shown the sample of the Java trace, obtained by executing the instrumented bytecode on the Hyades platform/framework. As we see in the trace lock is acquire before the notify instruction and the <lockAcquire methodentry> refers to the same objectIdRef that is “8605” as that of <notify methodEntry>.

<! -- below are 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:** *Sample of the trace obtained after instrumentation (it contains the additional events logged)*

#### 4.6.4 Trace Reduction

The Java trace obtained by the Hyades framework is of large size, usually of the order of 6-20 MB. It becomes difficult to unmarshal, handle and further verify the property on such a large size of trace. The large size of the trace is because most of the irrelevant events are filtered in. These events are irrelevant from the certain property verification. To reduce the size of the trace, we used fine-tuned filters of Hyades framework to filter in the relevant events and filter out the irrelevant one to reduce the trace size. For example if we are required to only obtain the wait and notify methodentries in Java trace. We write filter such as “*Java.lang.Thread.\* Wait Include*” , “*Java.lang.Thread.\* Notify*

*Include*” and “\* \* *Exclude*”. This filter setup will only include wait and notify methodentries in trace, excluding most other methodentries, except for few.

After using these filters, we were able to get the smaller size trace of the order of 63-200 Kb. The size of the reduced trace depends upon the type of filters used, no of the filters used, size of the original program and other factors. The reduction in trace size, lessen the property verification time.

#### **4.6.5 Benefits/Limitations of our Instrumentation Approach:**

By using our combined approach, we can reap the benefits of both the approaches, and lessen there shortcomings. Because our in the combined approach we individually list the advantages and limitation of both the approach.

##### **4.6.5.1 Hyades Tracing**

Here we will discuss the advantages/limitation of Hyades Tracing

###### ***4.6.5.1.1 Benefits of the Hyades tracing:***

- 1) Events obtained from the program execution are in XML format, to analyze this trace we need to parse it and there are available many Java-XML parsers such as SAX, DOM etc.
- 2) Along with events such as <method entry> and <method exit> logged, these events contain timestamp information. These timestamps helps in verifying certain properties in which one need to compare there values.

- 3) Filtration option is available in Hyades, to filter out the irrelevant events and filter in the relevant event. This filtration, reduces the size of the Java trace, which further lessen the property verification time ().
- 4) Hyades tracing is non-intrusive one, which save us from complicated and error-prone process of custom instrumentation
- 5) Hyades 3.0 (a latest version of Hyades) also provides support for bytecode instrumentation known as Java probe insertion kits

#### ***4.6.5.1.2 Limitation of the Hyades tracing***

- 1) There is the limit to the type of events we can obtain in a trace, using the Hyades tracing.
- 2) With current Hyades version, we cannot obtain the updated values of variables in the XML trace, but these values are printed at the console output. To analyze the trace, we need to combine both the traces namely the Hyades XML trace and the trace obtained at console output.

#### **4.6.5.2 Bytecode instrumentation based tracing**

Here we discuss the benefits and limitations of bytecode instrumentation based tracing

##### ***4.6.5.2.1 Benefits of bytecode instrumentation based tracing***

- 1) We can customize the bytecode instrumentation, to obtain the type of events and there attributes as per our requirement.
- 2) The approach was employed successfully in several projects (JRat, JMPaX etc)
- 3) Any events of interest could be recorded and collected

#### ***4.6.5.2.2 Limitation/side effect of bytecode instrumentation based tracing***

- 1) For the instrumentation, we need to write instrumentation code, to first read the XML file and insert the bytecode instructions where required
- 2) We need to understand the Java bytecode language to have the proper bytecode instrumentation. Learning such low-level language requires additional efforts (as opposed to other approaches)
- 3) A speed of the target program can be slowed due to the instrumentation. A real time application may violate temporal requirements because the instrumentation slows down the application.
- 4) Thread Scheduling: The execution order among the thread of the program may get disturbed due to the slowed execution speed. This changed order may violate certain requirements. It might give a synchronization error (it is highly improbable) when the extra delay become large

### **4.7 Execution of the Instrumented Program**

As described above, the Java class file (bytecode) is instrumented with calls to methods such as *Object.lockAcquire/Object.lockRelease* and these methods are written to Java trace when the instrumented program is executed. In addition, other classes can be constructed “on the fly” as required, and added to user packages containing the target code to be instrumented. The instrumented program is started as usual, e.g. an application is started by invoking the “main” method of the specified class. When an instrumentation method is first encountered in the code, the instrumentation class is loaded and a static initializer for the class is executed.

## 4.8 Conclusion and Future work

We presented the instrumentation approach/package used in our runtime analysis approach. Our is an integrated instrumentation approach, which helps us to get the benefit of both the approaches. Through this instrumentation approach we were able to extract the required information from the target program for our analysis. But there are few immediate tasks which we want to accomplish, listed here.

Explore the best possible way to integrate the two traces namely one obtained at the console output and other Java trace obtained in XML format

Also there is some other future work.

1. Experiment instrumentation using the bytecode instrumentation toolkit plugin integrated in the recent version of Hyades (Hyades 3.0) and compare it with out instrumentation approach
2. Explore ways to reduce the overhead caused by instrumentation on the target program execution.
3. Explore the aspect base instrumentation approach and is possible look for the way to integrate in our instrumentation package.

We are planning to investigate ways by which we can further reduce the overhead. Our possible we domain we are interested to explore the aspect based instrumentation. The work on the aspect based instrumentation is already being carried at the NASA, we further are interested to explore we can used the aspect based instrumentation approach in our instrumentation package and

## 5 Chapter 5: Custom Based Detection Approach

### 5.1 Introduction and Motivation

The custom based detection is a semiformal approach to analyze the execution trace of java program (collected in XML format) against certain properties/antipatterns. The custom based detection is a post-mortem/offline-trace analysis approach. In this approach first the MT antipatterns to be analyzed are formally specified at EFM or EFSM. The formally specified antipatterns are shown in section 3. Then these antipatterns are coded as java detectors. Sample of such detectors are shown in section 4. The execution trace of the program collected in XML format is unmarshaled using XML-Java technology called JAxB. The java trace is then analyzed for antipatterns using the Java detectors. The output of such analysis is a (possibly empty) set of antipatterns/property violations printed on console output.

Motivation of custom based analysis approach is to develop a semi-formal runtime analysis approach based on the idea of concluding antipatterns/properties of interest in a target program from the single run of the program. Our approach is a semiformal one because antipatterns/properties to be detected are first formally specified as FSM or EFSM.

Reduce the overhead caused by the instrumentation in the target program. The instrumentation approach followed in our approach helps in reducing overhead compared to other instrumentation. Because our is an integrated instrumentation approach which combine two instrumentation approach namely bytecode instrumentation (a lightweight approach) and Hyades tracing, which is a non-instruction trace collection approach.

Reduce the trace analysis time. The collected trace in our approach is of XML format. Because of this, trace analysis is comparatively faster as data in XML format is quite representative (in XML, the data is collected in tags). Also there are available many parsers such as SAX and DOM parser, which make it easier and faster to read and analyze. XML has being accepted as de-facto standard for the data inter-exchange over the internet. In the particular case of runtime analysis in multiprocessor environment and where the inter processor communication is over the internet, the XML-base trace analysis proves to be very handy and extendable.

## **5.2 Approach Overview**

The architecture of custom based approach is shown in Figure 5.1. The input to tool consists of two entities: the Java program in byte-code format to be monitored (created using a standard Java compiler) and the properties/antipatterns to be verified.

The tool can be regarded as consisting of three main modules: event collection module, parsing module and an analysis module.

### **Event Collection Module:**

In event collection module, Java application (instrumented/uninstrumented) is executed on Hyades platform/framework. Hyades framework, contain a library known as Java Profiling Agent that attaches to a Java virtual machine (JVM) to capture and record the behaviour of a Java application. The output from the agent (events) is emitted as XML fragments. There is an option for mode selection and event filtration in Java Profiling agent. It can run in one of the four modes namely standalone, enabled, controlled and application.



### Parsing Module:

In this the obtained XML trace is read and unmarshalled using JAXB package. JAXB package is a XML-Java technology developed by Sun Microsystems to unmarshall the java trace.

### Analysis Module:

Here the generated java execution trace is analyzed against the properties/antipatterns. First we formally specify the properties/antipatterns as FSM or EFSM. Then these properties/antipatterns are implemented in Java. The advantage of formally specifying the property (in FSM or EFSM format) is that it helps us to properly understand the property and thus one can correctly implement it in java.

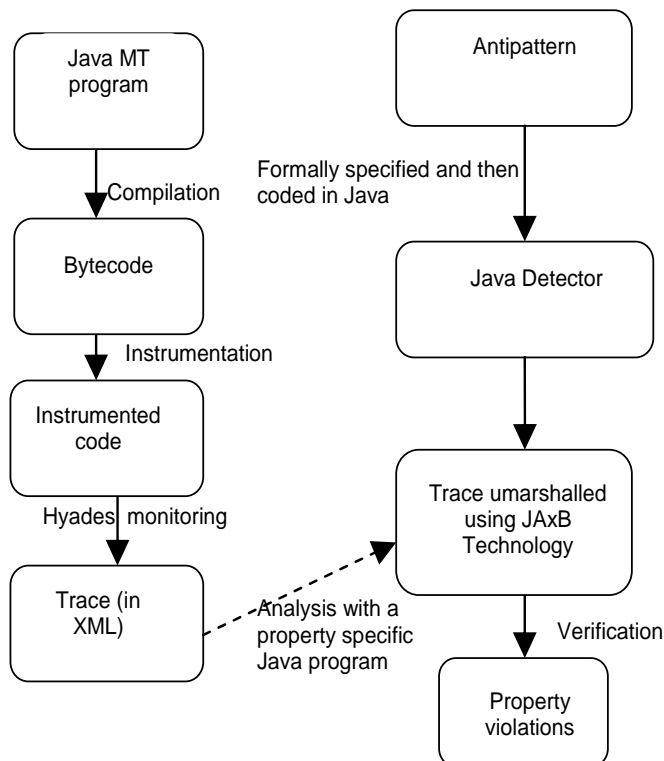


Figure 5.1: Offline custom based trace analysis architecture

## 5.3 AntiPattern Formalization

### 5.3.1 Formalization of the “double call of start ( ) method” antipattern

#### 5.3.1.1 FSM Formalization of “double call of start ( ) method”

In order to implement pattern correctly and efficiently we started from formal, automata like description. Formal description of the pattern is necessary step in model checker based trace verification, and, as we believe, beneficial in custom detector development. This antipattern can be instantiated in a set of automata (finite state machines), where each automaton corresponds to a thread present in a trace, or in single trace-independent extended automaton. In the first case, we can build the following automaton, where black state indicates that the antipattern is detected.

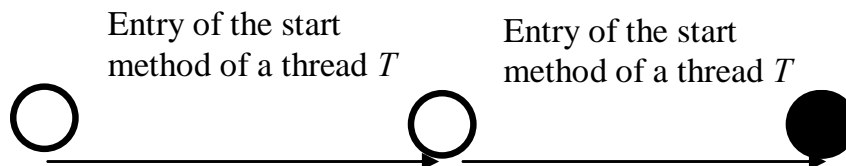


Figure 5.2: *FSM formalization of double start ( )*

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. (Meanwhile, we skipped the identification of the start method id – once created, methods, objects, classes, and threads are refereed by ids.) Similar preprocessing will be required for property verification, based on model checker, unless the former provides a richer language than automata.

### 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

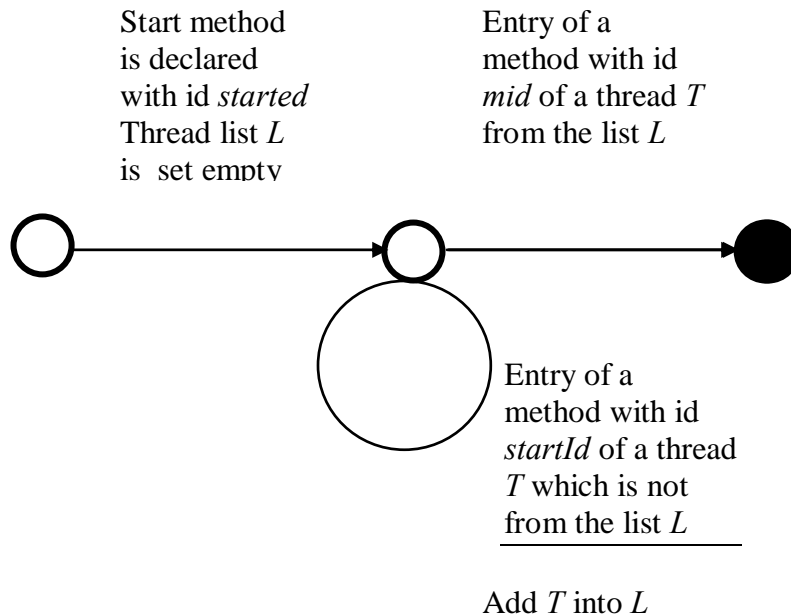


Figure 5.3: EFSM formalization of double start ()

The formal descriptions were found helpful both for antipattern understanding and detector implementation.

## 5.3.2 Formalization of the “PREMATURE JOIN CALL” antipattern

### 5.3.2.1 FSM Formalization of “premature join call” antipattern

Here we give the formal, automata like description of the “premature join () call” antipattern. 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 a trace. Firstly we build the following automata in which black state, when reached indicates that the antipattern is detected.

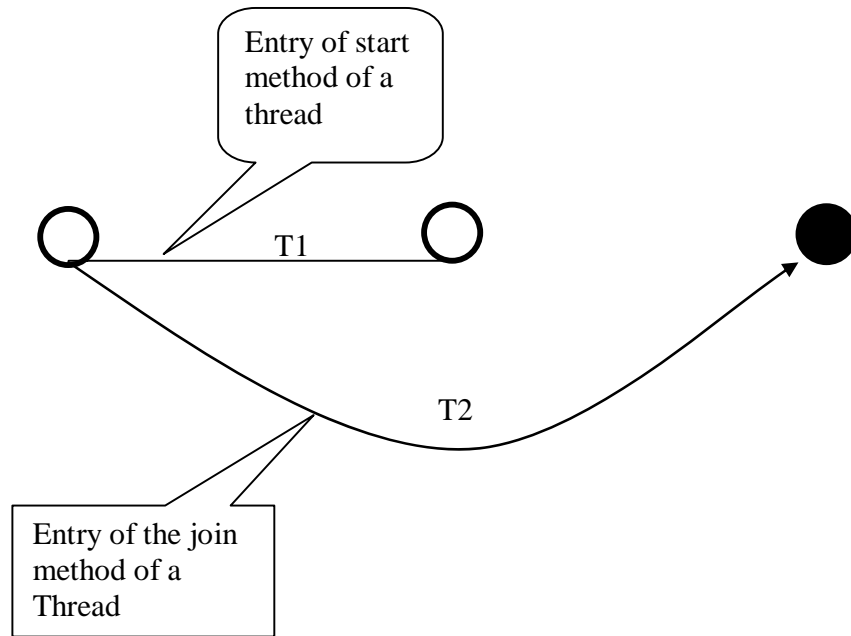


Figure 5.4: *FSM formalization of premature join ()*

Implementation of a premature join () call antipattern detector based on the above automaton involves trace pre-processing to build the list of thread Ids (more exactly ids corresponds to objects), and then scanning the trace with evaluation of a set of automata. Informally it can be said that the methodEntry of join () and start () method are located for each thread T. Then timestamps of methodEntry for join () method and start () method are compared. If the join () methodEntry happened before start () methodEntry on a particular thread T, then “*premature call of join () method*” message 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:

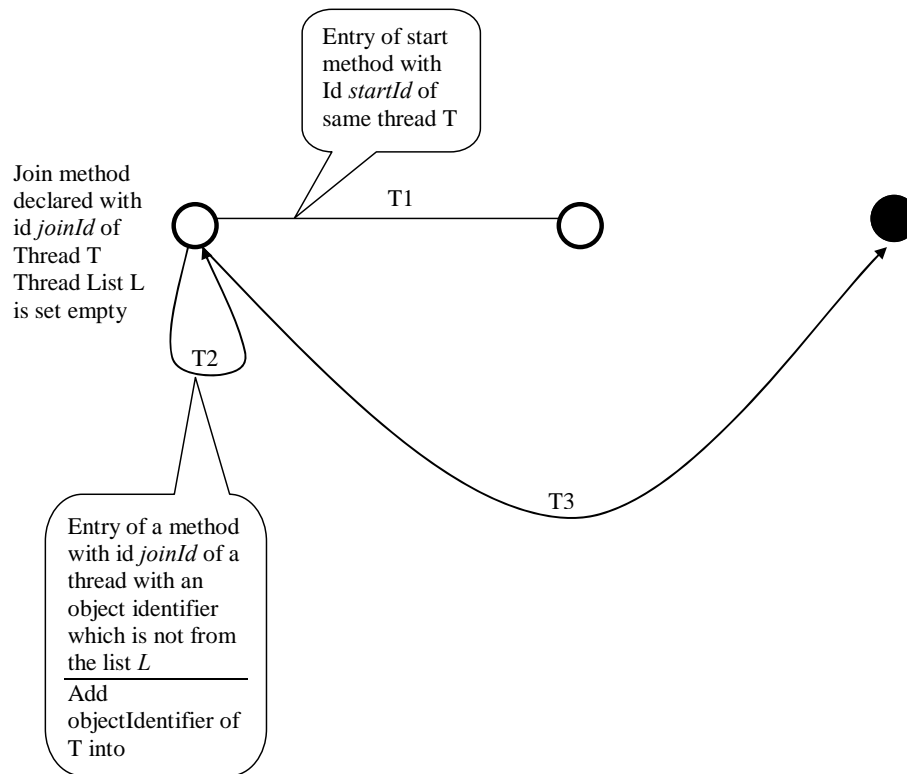


Figure 5.5: EFSM formalization of premature join ()

### 5.3.3 Formalization of Wait Stall

#### 5.3.3.1 FSM Formalization of wait stall

Here we give the formal, automata like description of the “wait stall” antipattern.

Informally this antipattern can be described as “A wait stall can occur when a thread calls a wait () method with no timeout specified”.

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 a trace. Firstly we build the following automata in which black state, when reached indicates that the antipattern is detected.

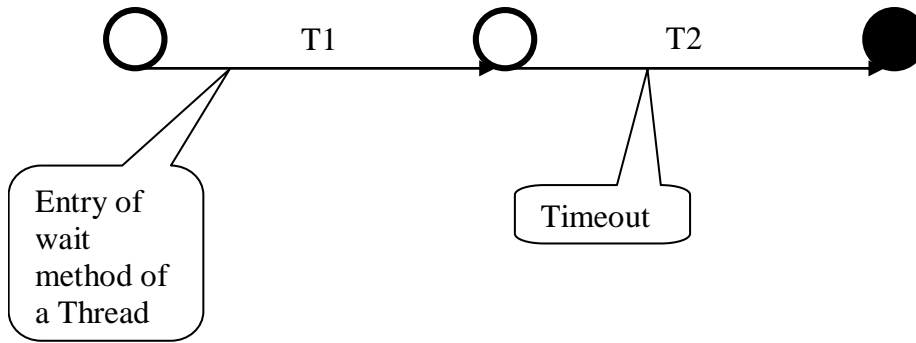


Figure 5.6: FSM formalization of wait stall

### 5.3.3.2 EFSM Formalization of wait stall

In a single extended machine (EFSM) the “Wait Stall” antipattern can be formalized as:

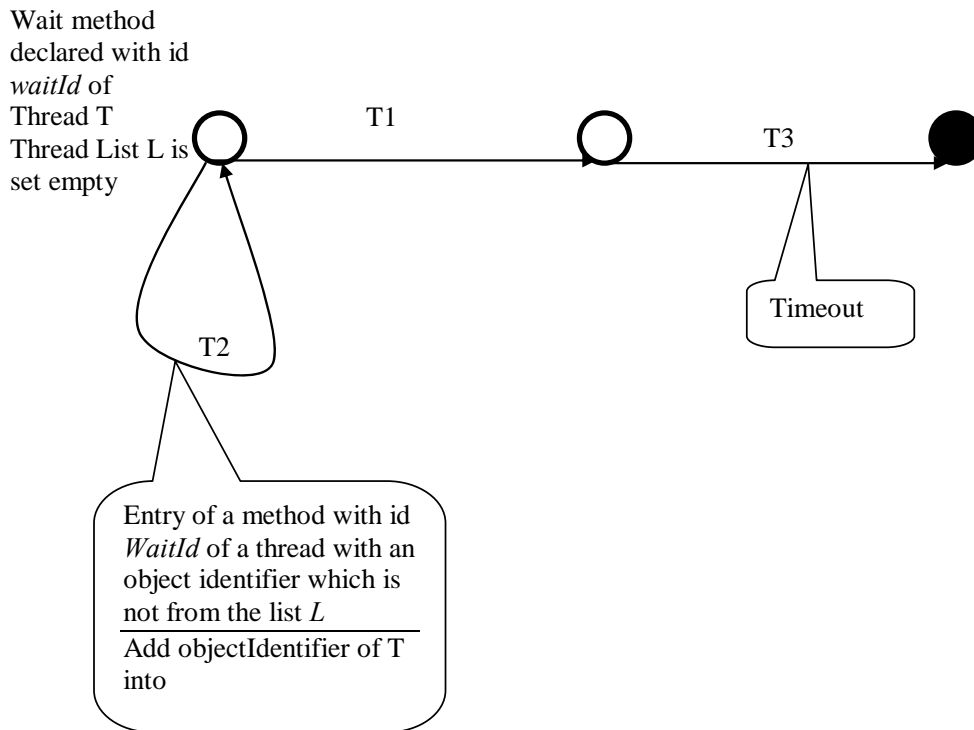


Figure 5.7: EFSM formalization of wait stall

## 5.4 Custom Detectors Implementation

### 5.4.1 Double Start () Implementation in Java

Here we discuss the algorithm for the detection of double start (). Before we formally specified this antipatterns as FSM and EFSM. As shown below in figure 5.8 a sample of the double start () detector code, initially we define the variable `start_count = 0`. Then for each thread (referred by `threadIdi`, where *i* is threadId no) we iterate and look if there is call for `start` methodEntry twice. If we found such an instance then we compare there respective objectIds. If the objectId are found same we print the double call of `start()` method detected for `threadIdi`. Otherwise we print that double call of `start()` method not detected for `threadIdi`.

```
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 5.8: Code sample for detection of double start ()

## 5.4.2 Premature Join () Implementation in Java

As we know from the definition of the premature call of the join () method. It consists in invocation of the join method to the thread, which is not yet started [Hal04]. In the Java trace thread should terminate (runExit event) should happen after the corresponding joinEntry and before it joinExit event. In the Java trace we verify this ordering for every threadId and if we see any instance where this ordering is violated, premature call of join () message is printed.

As shown in the figure 5.9 for every thread, represented by the threadId $i$  (where  $i$  is an integer of value 2, 5, 6 etc) we compare the time stamp of the runExit event with the corresponding joinExit event. The comparison is made using the compareTo, and if the timestamp of runExit is greater than that of joinExit timesamp, 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.

```
// 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++)
        {
// starting (if loop) comparing objectid of runexit and joinexit array
            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_list.joinexit_thr
                    ead_array[i]) < 0 ||
event_list.runexit_thread_array[j].compareTo(event_list.joinexit_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 5.9: Code sample for detection of premature join ()

## 5.5 Custom Detection Results

Although with custom based detection approach we can verify most of the antipatterns identified before, but it gave significant edge in the detection of certain antipatterns such as double call of start () method, premature call of join ( ) method and wait stall( ).

Here we will mostly discuss about these antipatterns detection. The following technologies were used for the custom based trace analysis are:

1. Compiler: Java 1.4
2. IDE: Eclipse
3. Java - XML Tool: JAXB (Java Architecture for XML Binding)

The experiments were performed on the following the system configuration:

1. Operating System: Window 2000
2. CPU: AMD Athlon 900 MHz
3. RAM: 512 Mbytes

### 5.5.1 Antipattern: Double Call of Start () method

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

Application: It is a fragment of Java multi-threaded platform Guest [Mag02]

Trace size	Execution time	Total time ( Execution + Compile Time)
63.6 KB (using fine tuned filters)	4s (approx)	27 second

**Table 5.1: Analysis time for double start () detection**

The custom detection gave a significant edge over static analysis particularly in the double call () detection. Here we explain the in detail the detection by static analysis and then compare it detection using the custom based detection approach.

Double call of start () method antipattern was detected by static analysis tool Extended-JLin at the following location.

```
FILE NAME=ca/crim/guest/main/NativeAgentSupport.java..POSITION COLUMN=17
LINE=225
TESTNAME=MultiStartMethodCallPattern.
MESSAGE=Another start method Call
FILE NAME=ca/crim/guest/main/NativeAgentSupport.java. POSITION COLUMN=9
LINE=261
TEST NAME=InternalMethodCallPattern.
```

The message *Another start method Call*, is emitted corresponding to the *Double call of the start method of a thread* antipattern, is a false positive. The tool detects the antipattern in the following segment of the code. Actually, it is signaled for the second start() method call in line [9]. However, the 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 5.10: *Sample of Guest application code*

Based on the information (like `methodIdRef`, `ObjIdRef`) 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 trace which were used for detection are given below.

```
<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 5.11: Sample of Java trace for double start () detection**

From the XML element <methodDef> we get the methodId = “302” corresponding to the start method. The methodEntry event and its attribute objIdRef corresponding to the methodIdRef = “302”(start) is located. The objIdRefs attribute of the corresponding methodEntry event is compared and found not the same. These objIdRefs are also referring to different threads. Thus we conclude from the analysis that the *double call of the start() method* is not on the same thread. Thus using the dynamic analysis false positive from the static analysis is detected.

### 5.5.2 Antipattern: Premature Call of Join () Method

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 [TR2].

Application: Custom Race program from [JPROBE]

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

**Table 5.2: Analysis time for premature join () detection**

A Snapshot of the console output for “premature join” detection based on custom detection Approach is shown in appendix A

### 5.5.3 Antipattern: Wait Stall

Description: The thread should not wait (after calling the wait method) for more the user specified amount of time [TR2].

**Detection:** Initially as for start method, from the XML element <methodDef> we get the methodId = “302” ( a method identifier) for the wait method. The methodEntry and its methodExit of wait method for a particular thread ( referred by threadId) are located. Then time difference between methodEntry and its methodExit is calculated and compared with the user specified time period (in this detector it is 0.5 seconds). If the calculated time difference is more than the user specified period then “*wait stall*” message is printed.

This antipattern can only be detected by Dynamic Analysis, because of the timestamp information provided by the methodEntry, methodExit and threadstart event in the XML trace.

The relevant events 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 5.12: Sample of java trace for wait stall detection**

## **5.6 Advantages/Limitations of custom based approach:**

### **5.6.1 Advantages of Custom Based Detection Approach**

- 1) It could detect most of the antipatterns identified by us.
- 2) It could detect false positive message given by static analysis particularly in the detection of “double call of start () method” antipattern. Thus this approach gave a

- significant edge over static analysis. It could detect false positive because the trace provide the information that two start methodenties are referring to different objects (objectIdRef).
- 3) It is scaleable (i.e. it can used to analysis execution traces of large size applications).  
But it gave memory exception, when the trace size exceeds 25MB or so.
  - 4) This approach has also been used in other runtime analysis tools such as JPaX to detect errors.
  - 5) Do not require heavy bytecode instrumentation, because most of the information from a Java application is collected and recorded using the Hyades platform.

### **5.6.2 Limitations of Custom Based Detection Approach**

Provides less coverage than heavyweight formal methods:

## 6 Chapter 6: Model Checking with Spin

### 6.1 Introduction

#### 6.1.1 Model – Checking

Model checking is a *push-button* technique for verifying finite state concurrent systems against required specification of the system. The tasks involved in model checking are as follows [SEN03]:

A formal model of the system is build in terms of a state transition system. The state transition system is a tuple  $M = (S, S_0, R, L)$  where

- $S$  is the finite set of states
- $S_0$  a subset of  $S$  is the set of initial states from which system can start its execution
- $R \subseteq S \times S$  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

The state transition system of a concurrent program can be constructed automatically by exploring all possible states of the system that can be reached from the initial state [SEN03].

- The properties that the model must satisfy are stated as a specification. The specification is usually given in some logical formalism. The commonly used logics are temporal logics

- 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 task is completely automatic

In model checking, all possible computations of the systems are analyzed. So the method 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 problem is called the *state space explosion* problem. In sequential programs input variables may have many possible values leading to a large number of possible states. In concurrent programs, nondeterministic execution can lead to a large number of states. If the total number of possible states of the system is large, model checking becomes intractable which makes this technique not scalable. We take the formal logics used to specify safety properties and incorporate the logics in our approach. This makes our approach more formal compared to the *ad hoc* testing used in traditional debugging.

## **6.2 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 been used to trace 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 reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

### **6.2.1 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.

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).

### **6.2.2 Features of Spin**

SPIN can be used in three basic modes:

- 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 (a proof approximation technique)

### **6.2.3 DOCUMENTATION**

Gerard J. Holzmann, The Spin Model Checker “Primer and Reference Manual”.

Addison-Wesley 2004

Basic and more advanced usage of Spin, such as language features and validation modes for proving linear temporal logic formulas, are described in the book. The more recent extensions of the tool are described in the papers.

The book contains an explanation of the code and describes the main validation strategies.

The Spin software is written in ANSI standard C, and is portable across all versions of the UNIX operating system. It can also be compiled to run on any standard PC running a Windows 98/2000/NT/XP operating system.

### **6.2.4 AVAILABILITY**

This software is available for free download at <http://spinroot.com/spin/whatispin.html>

The current version is Spin 4.2, which runs on any UNIX workstation, as well as Windows 98/NT/2000/XP.

## 7 Chapter 7: Modeling Trace with Spin

### 7.1 Introduction and Motivation

Previously we discussed trace analysis of Java Multithreaded applications based on custom based trace analysis approach. In this chapter we will discuss extension to custom based trace analysis approach another trace analysis approach, namely *model-checker based trace analysis*. In this approach, execution trace of Java program obtained in XML format, containing the relevant events is translated to PROMELA. PROMELA is input language for SPIN Model-checker. XML to PROMELA translation is done automatically by a java program based on a translation schema. The generated PROMELA model is then verified using SPIN model-checker against the multithreaded antipatterns (such as double start and premature join). These antipatterns were formally specified in LTL (linear temporal logic). In other words the subsequent model checking is guided by the trace generated during the runtime analysis. SPIN 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 for developing the model checker based trace analysis, is our interest to try applying formal techniques for trace analysis. We implemented our formal analysis using SPIN model checker. As far as we know there is no such work similar to our as done in industry/universities. As far we know the closest work is done at NASA Ames Research centre [HAV04] that is combining runtime Analysis with Model Checking. In

this work, they tried to combine the runtime analysis and model checking approach in such as way that warnings produced from the runtime analysis are used to guide a model checker. This technique was implemented the NASA developed tool Java model checker called Java PathFinder [JPF].

The work done at NASA was an extension to JPF to perform runtime analysis on multi-threaded Java programs in simulation mode, either stand-alone, or as a pre-run to a subsequent model checking, which is guided by the warnings generated during the runtime analysis.

Secondly as we already implemented and experimented custom based trace analysis approach. We were interested to evaluate other trace analysis approaches in comparison to Custom based trace analysis approach with respect to parameters such as:

- Quality of analysis
- Time usage
- Resource consumption etc

Thirdly the model based trace analysis would possibly we used for the predictive trace analysis. As Model checker analyze various possible event interleaving.

The full blown model checking has a major limitation that it suffers from the state explosion problem. To overcome this limitation, we model check the trace, a trace is an abstract representation of the target application. In brief our is an abstract model checking approach.

## **7.2 Approach Overview**

The architecture of our approach/tool is shown in Figure 7.1. The input to tool consists of two entities: 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 console output.

The tool can be regarded as consisting of three main modules: an instrumentation module, a translation module and an analysis module. The instrumentation module performs instrumentation on a program to be analyzed based on the user specified instrumentation specification. The instrumentation specification, contains information such as classes, methods, class /instance variable (updates), to be monitored and thus instrumented accordingly. The methods (such as lockAcquire, lockRelease, variable\_write, variable\_read) are inserted in the bytecode based on the input information from the user. The bytecode instrumentation is performed using JTrek, a Java byte-code engineering tool [JTREK] from Digital. This tool allows to read Java class files (bytecode), traverse them as abstract syntax tree while examining their content, and insert new code in a highly flexible manner [JPaX]. The instrumented program when run, on the Hyades framework will emit relevant events in XML format to an external file. These events are then input to the translation module.

The translation module receives the events, and generates a PROMELA model based on an input translation schema.

The analyzer module performs functions such as property/antipatterns verification and print out the property violation (if any) to the console output. Explaining it more elaborately, the analyzer receives the generated PROMELA model from the translation module and then verify it against properties/antipatterns specified in the specification

script. Verified can be done using two approaches namely custom-based detection approach or model-checker based approach. In the previous chapter, we discussed custom-based approach, in this chapter we will focus mostly on model-checker based approach. In model-checker based approach the property/antipatterns is specified as LTL formula which further translates to never claim. During the verification, it checks for never claim in the generated PROMELA model. If it finds the instance property violation (no never claim) it prints the warnings the console output. Along with warning, it prints other verification details also, such as depth reached in the model, no of transitions covered, no of matched states, errors if any (antipatterns) found and verification time. A sample of the verification output snapshot is shown in Appendix C.

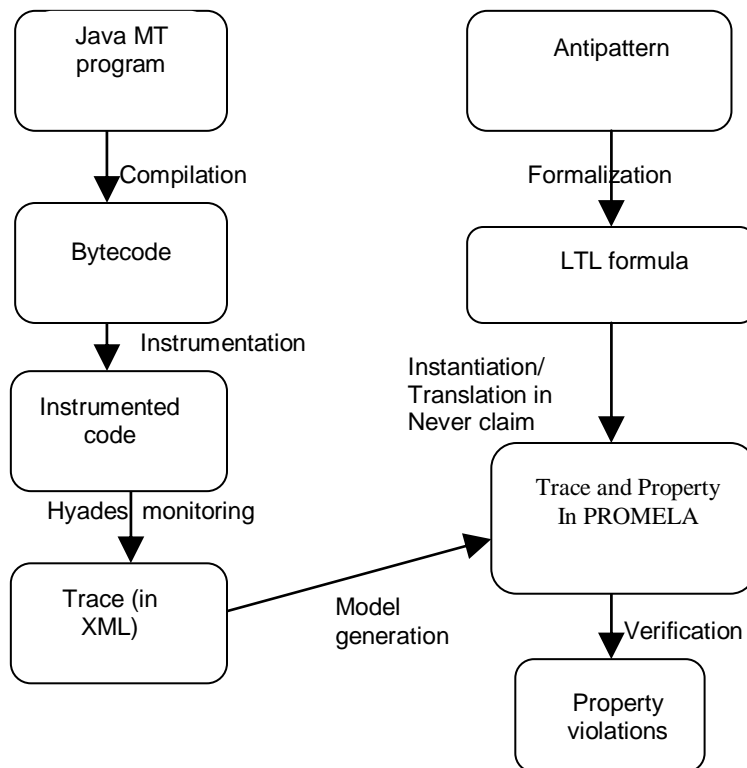


Figure 7.1: Diagram of the approach workflow

### 7.3 Translation

General Idea: The general principal regarding translation is the following. The XML trace generally consist of a set of tags and declarations and these tags focuses on providing information about the data itself and it's relation to other data tags of the events logged. For example, such as for methodEntry event is logged (in XML format) as `<methodEntry threadIdRef="7" time="1093561048.957960600" methodIdRef="103" objIdRef="8605" classIdRef="120" ticket="1162" stackDepth="5"/>`, whose attributes consist of (mostly) threadIdRef, time, methodIdRef, objIdRef, classIdRef, tickets and stackDepth etc. The data type of attribute's value can be either strings or integers.

As illustrated in the figure 7.2 the methodentry event logged in XML format is translated to the PROMELA code. The translation procedure is as follows explained below in steps:

1. Before the start of event in the PROMELA model a comment message is written stating the "type of methodentry of which thread "Id" and for which method "Id"
2. Initially in the PROMELA model event type is written as "name = methodEntry or methodExit" depending upon the type of event
3. Attributes of the events such as threadIdRef = 6, classId Ref = 116, objIdRef = 8606 are mapped one- to- one to the PROMELA code as shown in the figure 7.2
4. In the XML trace timestamp of the event is the absolute value such as "time = "1091230513.794350600" but in the PROMELA model, time is logical not absolute such as "time = 7", which signifies that this particular event is the seventh (7<sup>th</sup>) one in the sequence.

5. Attributes of the events such as `methodIdRef`, `objIdRef`, `classIdRef`, `threadIdRef` which are of *String* datatype in the XML trace are declared as *int* datatype in the PROMELA model as shown in the figure 7.3.
6. Events of type, for example `notify methodentry`, `wait methodexit`, `notifall methodentry` are assigned *mtype* datatype in the PROMELA model as shown in the figure 7.3.
7. The attributes of the event, such as `ticket`, `stackDepth` and their respective values which are not relevant from the analysis viewpoint are not translated to PROMELA model.

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

Figure 7.2: XML to PROMELA Translation

```

#define N    120    /* nr of rendezvous 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 7.3: *SPIN: Couple of PROMELA Constants and Constructs*

### 7.3.1 XML to PROMELA Translation

Each thread is modeled by a PROMELA process. The trace events themselves are translated in more or less direct way, where each event attribute is modeled by PROMELA variable. For few instructions, `join ()` and `start ()` we model their Java semantics. For other thread related Java constructs, we follow distributed trace approach that assumes that, only events of same thread (process) or involved in a communication are ordered. 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 are guaranteed 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 [Java].

*Variable/Data Type 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.

*Process Declaration:* Each thread in trace translates to active process in PROMELA. For example, a trace that consists of three threads namely *thread2*, *thread5*, *thread6*, translates to a PROMELA system that consists of three active processes namely *process2*, *process5*, and *process6* respectively.

*Relevant events:* Currently start (), join () wait (), notify (), notifyall(), LockAcquire(), LockRelease() method entry and exit, 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.

*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. D\_step insures that each event is atomic and instant.

*TimeStamp:* The events in the PROMELA model are assigned the logical timeStamp value, rather than real-time value as of the trace.

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

### **7.3.2 Synchronization in Java**

A race condition between two (or more) threads occurs when they modify a member variable of an object simultaneously.

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 semantic of these Java construct very closely, predicting new executions rather than 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 are likely results in imprecise model, at the least with our level of trace detail. Thus, when it concerns operations on locks, we just enforce order on event that relates to the same lock or thread. We detail the approach below.

### **7.3.3 Modeling Synchronization in PROMELA Model of Trace**

We consider three main types of MT synchronization events in PROMELA

1. Thread start (StartEntry) and Run method entry (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 thread the order is enforced. To enforce this ordering in PROMELA a message is exchanged between events such as wait, notify, notifyAll, entry/exit, lock entry and exit. To implement order/synchronization in PROMELA model, we use two approaches namely “variable/flag” based and “message passing” based approach. In some models we combine both the approaches to model synchronization.

#### **7.3.3.1 Message Passing Approach**

Message passing 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/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.

The example of the implementation of message passing based approach is shown in the figure 7.4. In this example a message is send from one event to another when they happen on the same objects (objectIdRef) but on the different threads (threadIdRef). As shown in the figure 7.4 that event Start\_MethodEntry happen on the threadId 2 and objectId 5317 and the consequent event Run\_Methodentry happen on the threadId 5 and objectId 5317.

To model this synchronization a message (Q[1]!message) is send fro the Start\_methodentry event and is received at the run\_methodentry (Q[1]?message).

```
/* Starting of the process 2*/
active proctype thread2()
{
/* Message is send to another object on different thread (at methodEntry event, at start) */
Q[1]!message->

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

d_step
{
/* Run MethodEntry for methodId 311 */

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

}/* End of process 5*/

```

Figure 7.4: *Sample of the PROMELA Model, synchronization based on Message Passing Approach*

### ***7.3.3.1.1 Advantage and Disadvantages of Message Based Approach***

Message based synchronization is used in our early research prototypes, since it is easy to visualize message exchange with message sequence diagrams in Spin. However we found that this limits scalability of approach, due to Spin limitations on number of messages and queues. Eventually we are going to replace message based event ordering with variable based. Spin visualization could be replaced with designed problem traceability/visualization module that completely hides model checking machinery from user.

A sample of the MSC for the model developed based on message based approach is shown in Appendix B.

### 7.3.3.2 Variable Based Approach

Variableflag based approach is particularly used to model behaviour of the join method.

Here we explain the variable based Approach as illustrated in figure 7.5. The global Boolean variable *ActiveThread<sub>i</sub>*, where *i*(which in this sample is 2, 5,6) is a thread identifier, is initially declared false. When the thread is started (i.e. RunEntry event) this variable is set to true (“*ActiveThread<sub>i</sub>* = true”). Similarly when the thread terminates (i.e Run Exit event) this variable is set to false (*ActiveThread<sub>i</sub>* = false). To enforce order between the RunExit and JoinExit, where later should happen before former and also on the same object but different threads, JoinExit event is executed only when this condition satisfies (“*::ActiveThread<sub>i</sub>* = = 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 7.5: Sample of the PROMELA model, synchronization based on the variable based Approach

### 7.3.3.2.1 Advantages and Disadvantages Variable Based Approach

Variable Based Approach is particularly advantageous to model large traces. In the message based approach messages are exchanged between events using channels (a construct in PROMELA). In the PROMELA model we declare an array of channels of size (for example L) and there to size of the (it is 255). Message Passing Approach fails to model those large traces in which the size of array of channel exceeds 255. Because of this inherent limitation is PROMELA variable based approach is advantageous to model large traces.

However “variable\flag based” approach has disadvantages over “message passing based” approach, as we cannot obtain the MSC (Message sequence chart) in former.

## 7.4 XML to PROMELA translation Implementation

We implemented the PROMELA modeling in Java. The Java code consists of about six (6) java classes.

The Java code for PROMELA model generation can be broadly divided in six (6) classes. These classes are main, logical Timestamp, Method\_def, Receive\_send message, thread\_count and list\_entry\_exit classes. Here we explain in steps implementation details

for the PROMELA model generation based on message passing approach. Please refer to the flowchart for the model generation in Appendix E for more details

1. First the XML trace is transversed and objectId's and threadId's of the events e.g. such as waitentry/exit, notifyentry/exit, lockAcquire entry/exit, lockrelease entry/exit startentry, runentry,runexit and joinexit are saved in objectId[] and threadId[] array and then there event types such as lockoperation\_event, startentry\_event, runentry\_event and other\_event are saved in event\_type [] array respectively
2. Then we iterate through the objectId[], threadId[] and event\_type[] arrays and search for those objectIds and threadIds values in the arrays whose preceding objectId and event type is same (exception in start and run case, where start methodentry should be preceded by run methodentry) but different threadId. If such an instances are found we save there indexes of the objectIds in send\_message [] array and save those index values from which they are different in receive\_message [] array
3. Events of the XML trace are not written (in the same order as in XML trace) to the PROMELA model, but in different order for example all the events of thread (e.g. Id 4) are written first, then the events of thread (e.g. Id 5) are written second and so on. Because of the way model is generated, the send and receive messages are not numbered according to the order obtained from the send\_message [] and receive\_message[] arrays.
4. But rather while iterating through the objectId [], threadId [] and event\_type[] arrays iterate in different manner that is first we iterate through the objectId array corresponding to the threadId for example 2, then 3, then 4 and so on. While iterating in such as manner we find those instances where consecutive threadsIds are different,



- but corresponding objectIds, are same. Whenever we find such instance save it index and then find it's relative position in the send\_message array using the int  $g = \text{Arrays.binarysearch}(\text{send\_message}, i)$  and then number the send\_message such as  $Q[g]!$
5. Similarly we perform the same functions to number the receive messages and thus number it  $Q[g]?$
  6. The XML trace is transverse again and threadId's of the threadStart event are saved in threadIdStart [] array
  7. The event body is written to the PROMELA model. Events of the XML trace are not written (in the same order as in XML trace) to the PROMELA model, but in different order for example all the events of thread (e.g. Id 4) are written first, then the events of thread (e.g. Id 5) are written second and so on. The information from the threadIdStart [] obtained in step 5 is used while writing events to the PROMELA model
  8. After the event body is written, to enforce the synchronization based on message passing approach, send and receive messages are inserted. The send and receive messages are numbered according to certain algorithm (described briefly in step 2 & 3). Before writing any event body to the PROMELA model, we check if it send or receive event. If the event is found to be of such a type, the send and receive messages according to the algorithm followed in step 2 & 3

## **7.5 Property Specification in Promela Never Claim and LTL**

Before discussing the property specification in LTL, we give here a brief introduction to LTL.

## 7.5.1 Temporal Logic Overview

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. In 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.

### 7.5.1.1 Linear-Time Temporal Logic

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 and past-time operators. The following syntax is used in our approach for these four operators:

- 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* and or a diamond (<>) to represent a sometime operator.
- Until (for the future). One can use the English keywords *Until*, or U to represent the until and since operator.
- Next iteration (for the future). One can use the English keywords *Next*, or a cross (X) to represent these operators.

## 7.5.2 Property Specification

Here we consider formalization of premature join antipattern in LTL. It consists in invocation of the join method to the thread, which is not yet started [Hal04]. Obviously it is impossible to specify such a pattern in LTL independently of number of threads.

Consider instantiation of antipattern for one particular thread  $T_i$ , that is join to thread  $T_i$  is called before start of  $T_i$ . Actually, it is more convenient to formalize absence of an antipattern, so a model-checker could pinpoint the problem with a counterexample to the correctness claim. Obviously a formalization of antipattern requires predicates which indicate invocation of the join method,  $\text{Join}(T_i)$  and thread start,  $\text{Start}(T_i)$ . To formalize absence of premature join to the thread ( $T_i$ ), we could use pattern specification system [SpecPattern]. The most adequate pattern is precedence:  $S = \text{Start}(T_i)$  precedes  $P = \text{Join}(T_i)$ , which is mapped into LTL as  $!P \text{ W } S = ! \text{Start}(T_i) \text{ W } \text{Join}(T_i)$ , where  $\text{W}$  is the weak until operator. On a trace that consists of  $n$  threads  $T_1, \dots, T_n$  instantiations of antipattern 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

$$! \text{Start}(T_1) \text{ W } \text{Join}(T_1) \ \& \ ! \text{Start}(T_2) \text{ W } \text{Join}(T_2) \ \& \ \dots \ \& \ ! \text{Start}(T_n) \text{ W } \text{Join}(T_n)$$

We follow the second approach, which is more convenient for us, while the first one provides better diagnosis: it is immediately clear which exactly thread is involved in premature join. Instantiation of predicates  $\text{Start}(T_i)$  and  $\text{Join}(T_i)$  is implementation dependent.

Similarly, double start absence is formalized with “bounded existence” or “absence of  $P = \text{Start}(T_i)$  after  $Q = \text{Start}(T_i)$ ” specification pattern, slightly modified with the next operator  $\text{X}$  to represent an open “after  $Q$ ” scope for:  $[(Q \rightarrow \text{X}[](!P))$ .

The properties could be formalized with automata using automatic transformation tool or an automata specification system, like in [Hal03].

Similarly we specify the double call for the start () method property. To formalize the “double start” property, we define a variable  $P$  such as (#define P methodIdRef == 309

`&& objectIdRef == 8047)` where 309 is the start methodId and 8047 is the objectIdRef for that particular Start Methodentry

We formalize in LTL formula as  $\langle \rangle (p \& \& X \langle \rangle p)$ , means that eventually in the model P will be preceded by P. In this model we verify this LTL formula for every thread

## 7.6 Model Based Approach's Results

To start the Spin model checker, click on RUN: (Re) Run-Verification in the Spin window. Spin will compile the PROMELA program into a C program, which when executed will do the model checking. While the compilation into C takes place, a small window pops up with the text: "Please wait until compilation of the executable produced by spin completes". When this window disappears, the model checker starts executing (the now generated C program). When this terminates, a window with the verification result appears as shown in Appendix B. In our case it says (top line): "property violated", and further down it states: "errors: 1". If there are no errors "errors: 0" is printed.

This message sequence chart (in the appendix A) can be explained as follows. The Thread 6 sends a message (No 1) to the thread 7 and thread 7 send the message (No 2) to the thread 6. The send message is label as message! (No 1, 2, 3, 4) Similarly the Thread 9 sends a message (No 4) to the thread 5 and thread 5 sends the message (No 5) to the thread 9. The message is send from one thread to another threads if the consecutive events in the PROMELA are on the same object (referred by the ObjectIdRef in the PROMELA model) but on the different thread. Similarly for the receiving, if the consecutive events refer to the same objectId but different threadId, the message is send by the previous event and received by the consecutive event.

The experiments were performed on the following the system configuration:

## System Configuration:

Operating System: Window 2000

CPU: AMD Athlon 900 MHz

RAM: 512 Mbytes

The following technologies were used for the model based trace analysis are:

## Technologies used:

Model Checker: Spin 4.1

Compiler: gcc (C compiler)

### 7.6.1 Antipattern: Double Call of Start () method

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

Application: It is a fragment of Java multi-threaded platform Guest [Mag02]

Trace size	Promela model size	Execution time	Total Time(Execution + Compile Time )
63.6 KB	3.25 KB	9s( approx)	34seconds

**Table 7.1: Analysis time for premature join () detection**

Promela model size	Pan.c build time	Pan.exe build time	Verification time
3.25 KB	1s( approx)	2s (approx)	1s( approx)

**Table 7.2: Verification & Compilation Time (double start ())**

#### 7.6.1.1 Verification Data

State Vector Size: 1592 bytes  
Depth Reached: 35  
No of transitions: 20 (visited + matched)  
No of matched states: 1  
No of states visited: 19  
No of errors: 1

## 7.6.2 Antipattern: Premature Call of Join () Method

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 [TR2].

Application: Custom Race program [JPROBE]

Trace size	Promela model size	Execution time	Total Time(Execution + Compile Time )
29.3 Kb	2.23 KB	5s (approx)	27seconds

**Table 7.3: Model Generation Time (premature join)**

Promela model size	Pan.c build time	Pan.exe build time	Verification time
2.23 KB	1s( approx)	2.5 s (approx)	1s (approx)

**Table 7.4: Verification & Compilation Time (premature join)**

### 7.6.2.1 Verification Data

State Vector Size: 1592 bytes  
Depth Reached: 21  
No of transitions: 17 (visited + matched)  
No of matched states: 1  
No of states visited: 16  
No of errors: 1

## 7.7 Open Problems and alternatives for trace modeling

First we list the possible future work or open problems in the model-based trace analysis approach:

1. As described before we verify the generated PROMELA model against the properties/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 property verification approach is that we implement the antipattern

- detectors in C. Then SPIN version 4.0 or later, support 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 using the SPIN model checker, the embedded code provide guidance to the precise location of such errors
2. We used object-oriented paradigm (OOP) for PROMELA model generation and extraction from XML based Java trace. An alternative to OOP, we can possibly use aspect-oriented programming (AOP) based tool such as Aspect J for the model generation and extraction. AOP based model extraction will be particularly useful when the XML trace is quite large with many events interleaving. In such a large trace, AOP proves to be very versatile, because in AOP we can divide the functionality of the code as concerns and code these concerns independently. In the exception cases where there are cross-cutting concerns
  3. Extend the model-trace analysis to verify other concurrency related antipatterns
  4. Extended the current error detection approach to error correction/error location finder in the target program. To extend it warnings emitted (antipattern violations) by the model checker, after trace analysis can be given as feedback to the target program. For this, we can write code which read/interpret the content of the warning and based on the content of the warnings can possibly locate the error in the target program. Once the error is located, we can code it further to correct the error, writing such a code will be quite trivial task. But using AOP it can be made possible to write error correcting code.

Another option is correcting the code at bytecode level, rather than at source code level. The correction at the bytecode level will save the software development cost. This type of automated error correction system, can possibly be used in those autonomous systems used in deep space missions where there is very little human interaction. Those autonomous systems are designed to be such that they have the capability to adapt and evolve themselves according to the environmental conditions. In such autonomous systems, we will need such an automated error correction system we can locate the errors and correct them, without very less or almost no human interaction

### **7.7.1 Alternatives to Trace Modelling**

There are few other alternatives to trace modelling, and here we list few of them.

- 1) We can possibly use mathematical techniques to create a model of the Java trace (in XML format), and then verify against the antipatterns/properties of the generated model using the Theorem Proving techniques.
- 2) We can experiment with other model checkers and possibly translate the trace to other model checking languages
- 3) We can implement algorithms for antipattern detection using Maude. Maude [1] is a modularized specification and verification system that very efficiently implements rewriting logic. Maude was developed at the computer science department, university of Illinois at Urbana Champaign. The automated software engineering group at NASA have already implemented such verification in Maude.



## **8 Chapter 8 Comparison**

### **8.1 Introduction and Motivation**

Here we will make a detail comparison between two approaches namely model-based trace analysis and custom based trace analysis approach. The comparison will be based on the experimental results, easy of usage, scalability, program complexity, quality of analysis and resource consumption. In section 2 we first explain the auxiliary details such as how the experiment was performed, its environment, technologies used and other related technical details. In the section 3 a detailed comparison is made based on the criteria such as time usage, quality of analysis, resource consumption, scalability and easy of usage etc.

Motivation to conduct such a comparative study is to analyze the feasibility of formal approaches in runtime analysis. The model based approach is formal and custom based approach in a semi-formal one.

### **8.2 Experiments Results:**

We used custom detectors to analyze large programs, such as SAP Vending Machine Server. The size of SAP vending machine Server trace is of the about 80 megabytes or so. Analyzing such a large trace will cause memory overflow problem. To reduce the trace size, we used fine-tuned filters. To counter such problem, probably more scalable XML tools could be used for XML parsing such as SAX parser.

As detailed before the model checking approach is further based on two approaches namely variable based and message based approach. Modeling the trace based on message passing approach does not scale to very large trace size, because of the inherent

limitation of the PROMELA language. Because of this reason we plan to completely replace message passing with shared variables for scalability purposes.

Nevertheless, model-checking approach could hardly outperform custom analysis in terms of scalability, since parsing of the XML trace file is required anyway to generate the model. Thus comparison is performed on middle size programs.

For comparison we performed experiments on two applications and two antipatterns using both custom and model-checking approaches. The first application is a fragment of Java multi-threaded platform Guest [Mag02]. The second is a toy demo program (borrowed from JProbe), both with injected faults and 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. For the model based trace analysis, technologies used were Model Checker- Spin 4.1, C compiler- gcc and for custom based trace analysis technologies used were:

- 1) Compiler: Java 1.4
- 2) IDE: Eclipse
- 3) Java - XML Tool: JAXB (Java Architecture for XML Binding)

**Table 8.1: Experimental Results**

	Bug	Trace Size	PROMELA Model Size	Custom analyzer	Model building and Verification
App1	Double start	64 K	3.25 KB	4	13
App2	Premature join	29.3 KB	2.3 KB	4	10
App3	Double start	667 KB	22.0 KB	6	20

### **8.3 Comparison**

The comparison between these two approaches is based on the criteria such as time usage, complexity of analysis, scalability, quality of analysis and easy of usage etc.

#### **Time Usage:**

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

#### **Complexity of Analysis:**

The model-based trace analysis is more cumbersome and complex than the custom based analysis approach because in former the PROMELA model of the trace is required to be generated, whereas in latter we directly analyze the XML trace.

#### **Scalability:**

Both the approaches can equally scalable, but model approach fares better than custom approach in analyzing large traces. When analyzing large traces, the custom approach gave the memory overflow exception whereas there is no problem in analyzing large trace with model based approach

By using the filters in the model generation code, the size of the PROMELA model (in Megabytes) is less as compared to the size of the execution trace used for custom based trace analysis approach.

#### **Quality of Analysis:**

Regarding the quality of analysis, model based approach is rated better than the custom based approach. Custom based approach cannot completely guarantee that the program

property/antipattern is satisfied because only one run is examined. In order to achieve higher assurance the model based trace analysis approach is used. Model based approach allows predictive trace analysis, in a sense that we could analyze several events interleaving.

Also model based approach can verify the parallel compositions of the trace and thus the analysis can be made more versatile.

We can experiment with others logics such as interval logic, real time logic can be better using model based approach than the custom based approach.

#### **Resource consumption –memory problems:**

Regarding the resource consumption (memory), model based approach is better than custom based approach because when the model is generated extra memory of consumed Whereas in the custom based analysis approach we directly analyze the execution trace.

#### **False positives:**

The custom based analysis may emit false warnings. By making the model based approach as guided model checking we can eliminate the false positives generated by the custom based analysis approach. The guided model checking the help focus the search of the model checker

Moreover in the model-based trace analysis approach we can set the search options in the model checker to cover parts of the state-space that weren't covered – e.g. to cover “deep” paths.

#### **Future research requirements:**

The future research is being undertaken as to influence the program behavior, when the property is violated. For the future research on this aspect the model based trace analysis

is better suited for such research projects because it being a formal approach it is more possible to give feedback and correct the errors in the target program.

Such as future research is undertaken for the autonomous applications used in deep space mission. In such an autonomous application there is little or very less human interaction.

## 9 Chapter 9: Conclusion and Future Work

We have presented two approaches for the runtime analysis, namely custom based and model based trace analysis approach. Motivation of this work is to present runtime analysis approach, with combine testing and formal methods, while avoiding the pitfalls of ad hoc testing and complexity of full-blown theorem proving and model checking.

Both these approaches provide an integrated environment to integrate two instrumentation approaches (profiling interface tool with bytecode instrumentation) to obtain an improved Java trace in XML format with the relevant events logged. These two runtime analysis approaches different in the antipatterns detection approach.

In custom based approach the Java trace is analyzed for antipatterns using Java detectors. Whereas in model based analysis approach the Java trace is first translated to PROMELA model, and then the extracted model is verified against the antipatterns specified in LTL using SPIN model checker. We have implemented algorithm for the model extraction from the Java trace.

We have designed the architecture as a modular architecture consisting of several components for the increasing flexibility. The approach being modular can be extended further to suit the future needs of the runtime analysis.

Then we made a detailed comparison between these two runtime analysis approaches based on the following criteria:

- 1) Quality of Analysis
- 2) Resource consumption
- 3) Time Usage

We divide our future work as long term and short term agenda. In short term future work will be:

1. Experiments with other real cases studies and compare the results with other runtime approaches such as JMPAX, JPAX and JavaMac
2. Experiment with other MT antipatterns.
3. To analyze the execution trace against high level requirement specification and also to experiment with new kinds of logics such as interval and real time logics
4. Secondly we are interested to experiment with this logic implemented in maude rather than in LTL
5. Implement GUI for both the approach. This will help us to better visualize the results
6. Use aspect Oriented programming (AOP) for model extraction, bytecode instrumentation.
7. Extended model based analysis approach using AOP to build autonomous software, which are adaptive and self evolving. In such an autonomous software testing could be automated to an extent that error detection and correction is automatic, with very less human interaction.
8. Guide execution via aspect based code instrumentation to explore the possible interleavings of a non-deterministic concurrent program during testing.

Our long term agenda for the future work will be to combine static analysis and runtime analysis.

## 10 References:

- [AntiPattern] An antipattern book [Online] <http://www.antipatterns.com/thebook.htm>
- [AntiPattern2] Wikipedia, Antipatterns [Online] <https://c2.com/cgi/wiki/Antipattern>
- [ABH03] C. Artho, A. Biere and K. Havelund. “*High-Level Data Races*”  
VVEIS'03, First International Workshop on Verification and Validation of  
Enterprise Information Systems Angers, France, April 22, 2003
- [ALD99] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, Susan J. Eggers:  
Static Analyses for Eliminating Unnecessary Synchronization from Java  
Programs. Agostino Cortesi, Gilberto Filé (Eds.): Static Analysis, 6th  
International Symposium, SAS '99, Venice, Italy, September 22–24, 1999,  
Proceedings. LNCS 1694 Springer 1999: 19-38.
- [ART01] Cyrille Artho: “*Finding faults in multi-threaded programs*”. Master  
Thesis. Institute of Computer Systems, Federal Institute of Technology,  
Zurich/Austin. 2001.
- [AS85] Bowen Alpern and Fred B. Schneider. “*Defining Liveness*” Information  
Processing Letters, October 1985
- [AH03] Allen Goldberg and Klaus Havelund. “*Instrumentation of Java Bytecode  
for Runtime Analysis*” Fifth ECOOP Workshop on Formal Techniques for  
Java-like Programs, Darmstadt, Germany, July 21, 2003
- [ARM98] Eric Armstrong “*Hotspot: A new breed of virtual machine*” 1998



<http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html>

- [BAU03] M. C. Baur. “*Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs*” Diploma Thesis, Computer Systems Institute, Swiss Federal Institute of Technology (Zurich), April 9, 2003
- [BER97] P. Bertelsen. “*Semantics of Java bytecode*” A Technical Report. Department of mathematics and physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997, [Online] <http://www.dina.kvl.dk/~pmb>.
- [Caffeine] Y.-G. Gueheneuc. Caffeine. [Online] <http://www.yann-gael.gueheneuc.net/Work/Research/Caffeine>
- [CAI03] Andrew Cain, Jean-Guy Schneider, Doug Grant and Tsong Yueh Chen “*Run-time Data Analysis for Java Programs*” 1st Workshop on ASARTI, Darmstadt, Germany, July 21, 2003
- [CKC98] G. A. Cohen, D. L. Kaminsky and J. S. Chase “*Automatic Program Transformation with JOIE*” In Proceedings USENIX Annual Technical Symposium, 1998
- [CHO02] J.-D. Choi and A. Zeller “*Isolating Failure-Inducing Thread Schedules*” International Symposium on Software Testing and Analysis (ISSTA2002), Via di Ripetta, Italy, July 22-24, 2002.
- [COH98] G. A. Cohen, D. L. Kaminsky and J. S. Chase. “*Automatic Program Transformation with JOIE*” In Proceedings USENIX Annual Technical Symposium, 1998

- [CS98] Jong-Deok Choi and Harini Srinivasan, “Deterministic Replay of Java Multithreaded Applications”, ACM SIGMETRICS Symposium on Parallel and Distributed TOOLS (SPDT), ACM Press, August 1998
- [CSGC03] Andrew Cain, Jean-Guy Schneider, Doug Grant and Tsong Yueh Chen. “*Runtime Data Analysis for Java Programs*” 1st Workshop on ASARTI, Darmstadt, Germany, July 21, 2003
- [DAH99] M Dahm. “*Byte Code Engineering*” In JIT 99 Proceeding
- [DAH01] M. Dahm. “*Byte Code Engineering with the BCEL API*” Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, April 3, 2001
- [DC01] M. Deters and Ron K. Cytron “*Introduction of Program Instrumentation using Aspects.*” OOPSLA 2001 Workshop on Advanced Separation of Concerns, October 2001
- [DY04] Laura K. Dillon and Qing Yu. “*Specification and Testing of Temporal Logic Properties of Concurrent System Designs*” Technical Report, March 2004
- [EFB01] T. Elrad, R. E. Filman and A. Bader. “*Aspect-Oriented Programming*” Communications of the ACM, Volume 44 Issue 10, October 2001.
- [ENG00] Editors of The American Heritage Dictionaries, The American Heritage Dictionary of the English Language, 4<sup>th</sup> edition, published by Houghton Mifflin Co, January 2000

- [FH02] R. E. Filman and K. Havelund “*Source-Code Instrumentation and Quantification of Events*” Foundations of Aspect-Oriented Languages (FOAL'02), Enschede, Netherlands, April 22, 2002
- [FZ03] Fancong Zeng “*An Initial Study of Common Coding Pitfalls in Java Programs*” MASPLAS '03 Mid-Atlantic Student Workshop on Programming Languages and Systems, April 26th, 2003
- [GEI01] M.C.W. Geilen. “*On the construction of monitors for temporal logic properties*” RV'01 - First Workshop on Runtime Verification, July 23, 2001 Paris, France
- [GH03] Allen Goldberg and Klaus Havelund “*Instrumentation of Java Bytecode for Runtime Analysis*” Fifth ECOOP Workshop on Formal Techniques for Java-like Programs, Darmstadt, Germany, July 21, 2003
- [GJSB00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha “*The Java Language Specification*” Addison-Wesley, second edition, 2000
- [HAG01] Peter Hagger “*Understanding bytecode makes you a better programmer*” Developer Work July 2001, [Online]  
[http://www-106.ibm.com/developerworks/ibm/library/it-haggar\\_bytecode/](http://www-106.ibm.com/developerworks/ibm/library/it-haggar_bytecode/)
- [HAL03] Hallal, H., Boroday, S., Ulrich, A. and Petrenko, A. “*An Automata-based Approach to Property Testing in Event Traces*” In Proceedings of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003), pp. 180-196. Sophia Antipolis, France, May 26-29, 2003.

- [HAL04] H. H. Hallal, E. Alikacem, P. Tunney, S. Boroday and A. Petrenko  
“*Antipattern-Based Detection of Deficiencies in Java Multithreaded Software*” Quality Software, Fourth International Conference on (QSIC'04) September 08 - 10, 2004, Braunschweig, Germany
- [HAV01] K. Havelund, Scott Johnson and G. Rosu. “*Specification and Error Pattern Based Program Monitoring*”. European Space Agency Workshop on On-Board Autonomy, Noordwijk, Holland, October 2001.
- [HAV02] Klaus Havelund “*Dynamic Program Analysis*” A talk at NASA Ames Research Center, October 2002.
- [HAV03] K. Havelund, C. Artho, D. Drusinsky, A. Goldberg, M. Lowry, C. Pasareanu, G. Rosu and W. Visser. “*Experiments with Test Case Generation and Runtime Analysis*” ASM 2003, 10th International Workshop on Abstract State Machines Taormina, Italy, March, 2003
- [HAV04] K. Havelund “*Using Runtime Analysis to Guide Model Checking of Java Programs*” The 7th International SPIN Workshop, Stanford, California, August-September, 2000
- [HOV04] David Hovemeyer and William Pugh: “*Finding Bugs is Easy*” JavaOne 4Sun’s 2004 Worldwide Java Developer Conference
- [HJR01] K. Havelund, Scott Johnson and G. Rosu. “*Specification and Error Pattern Based Program Monitoring*”. European Space Agency Workshop on On-Board Autonomy, Noordwijk, Holland, October 2001

- [HR01] K. Havelund and G. Rosu. “*Monitoring Java Programs with Java PathExplorer*” First Workshop on Runtime Verification (RV'01), Paris, France, 23 July 2001
- [HR02] K. Havelund and G. Rosu “*Synthesizing Monitors for Safety Properties*” International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France, April 14, 2002
- [HR04] Klaus Havelund and Grigore Rosu “*Efficient Monitoring of Safety Properties*” In Software Tools for Technology Transfer, 2004
- [HUA79] J. C. Huang. “*Detection of Data Flow Anomaly Through Program Instrumentation*” IEEE Transactions on Software Engineering, Volume 5, January 1979
- [HEU03] Dirk Heuzeroth, Thomas Holl, Gustav Högström and Welf Löwe “*Automatic Design Pattern Detection*” 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA
- [Hyades] Hyades Project, Eclipse platform [Online] <http://www.eclipse.org/hyades/>
- [IBM00] IBM AlphaWorks. *CFParse*, September 2000  
<http://www.alphaworks.ibm.com/tech/cfparse>
- [IBM002] IBM AlphaWorks. “*Jikes ByteCode Toolkit*” March 2000  
<http://www.alphaworks.ibm.com/tech/jikesbt>
- [IBM01] IBM Writing Multithreaded Java applications “Learn to avoid problems common in concurrent programming” February 2001 [Online]  
<http://www-106.ibm.com/developerworks/java/library/j-thread.html>

- [JCON] Java implementation of Design by Contract by apache open source community, [Online] <http://jcontractor.sourceforge.net/>
- [JFluid] A profiling tool for Java from Sun Microsystems, [Online] <http://research.sun.com/projects/jfluid/>
- [JPAX] Ron LeMaster and David Leberknight. *JPaX*  
<http://ic.arc.nasa.gov/researchinfusion/materials/JPaX/talk.pdf>
- [JPDA] Java Platform Debugger Architecture from Sun Microsystems available online at <http://java.sun.com/j2se/1.4.1/docs/guide/jpda/>
- [JProbe] A tool by Quest Software JProbe [Online] <http://www.quest.com/jprobe>
- [JVM] Tim Lindholm and Frank Yellin “*The Java Virtual Machine Specification*” Second edition [Online]  
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>
- [JVMPI] Sun Microsystems “*Java Virtual Machine Profiling Architecture*”  
<http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>
- [JRAT] Java Runtime Analysis Tool [Online] <http://jrat.sourceforge.net/>
- [J2SE 5.0] Sun Microsystems “*Java 2 Platform Standard Edition (J2SE) 5.0 ("Tiger")*” [Online]  
<http://java.sun.com/developer/technicalArticles/releases/j2se15>
- [KH98] Ralph Keller and Urs Holzle “*Binary Component Adaptation*”  
Proceedings of ECOOP, Brussels, July 1998

- [KHB98] Murat Karaorman, Urs Holzle and John Bruno “*jContractor: A Reflective Java Library to Support Design By Contract*”. A Technical Report 1998
- [KIM01] Moonjoo Kim “*Information extraction for run-time formal analysis*”  
Ph.D Thesis, CIS Department, University of Pennsylvania, September 2001
- [LAM77] Leslie Lamport “*Proving the Correctness of Multiprocess Programs*”  
IEEE Transactions on Software Engineering SE-3 March 1977
- [LEE03] Jooyong Lee, Ki-Seok and Jin-Young Choi “*Model Checking in Java Program Debugger*” 10th Annual International Static Analysis  
Symposium(SAS'03), San Diego, California, USA, June 11-13, 2003
- [LEW03] Bil Lewis “*Debugging Backward in Time*” Proceedings of the Fifth  
International Workshop on Automated Debugging (AADEBUG 2003),  
September 2003
- [LOG4j] Log4j Project [jakarta.apache.org/log4j](http://jakarta.apache.org/log4j)
- [LZ97] H. B. Lee and B. G. Zorn “*BIT: A tool for instrumenting Java bytecodes*”  
In USENIX Symposium on Internet Technologies and Systems, 1998
- [MYE97] G.J Myers. “*The art of Software Testing*” John Wiley and Sons, 1978
- [MT Java] Bil Lewis and Daniel J. Berg. “*Multithreaded Programming with Java Technology*” Sun Microsystems Press, 2000
- [ODB] Bil Lewis “*Omniscient Debugger (ODB)*” [Online]  
<http://www.lambdacs.com/debugger/ODBDescription.html>

- [PU03] Pietschker, A. Ulrich, A. “*A Light-weight Method for Trace Analysis to Support Fault Diagnosis in Concurrent Systems*”. Journal of Systemics, Cybernetics and Informatics, Vol 1. no 2. 2003.
- [PW02] Paul Pazandak and David Wells “*ProbeMeister: Distributed Runtime Software Instrumentation.*” USE 2002 Spain, June 2002
- [RC03] Grigore Rosu and Feng Chen “*Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*”. Third International Workshop on Run-time Verification (RV'03). Boulder, Colorado, U.S.A, July 13, 2003
- [RW02] Grigore Rosu and Jonathan Whittle. “*Towards Certifying Domain-Specific Properties of Synthesized Code*” Verification and Computational Logic (VCL'02), Pittsburgh, PA, USA, 5 October 2002
- [SBN97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “*Eraser: A Dynamic Data Race Detector for Multithreaded Program*” ACM Transactions on Computer Systems, ACM Press, NY, 1997
- [SEN03] Koushik Sen, Grigore Rosu and Gul Agha “*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, pages 337-346, Helsinki, Finland, 2003
- [SNO88] R. Snodgrass. “*A relational approach to monitoring complex systems*” In ACM Transaction on Computer Systems, May 1988



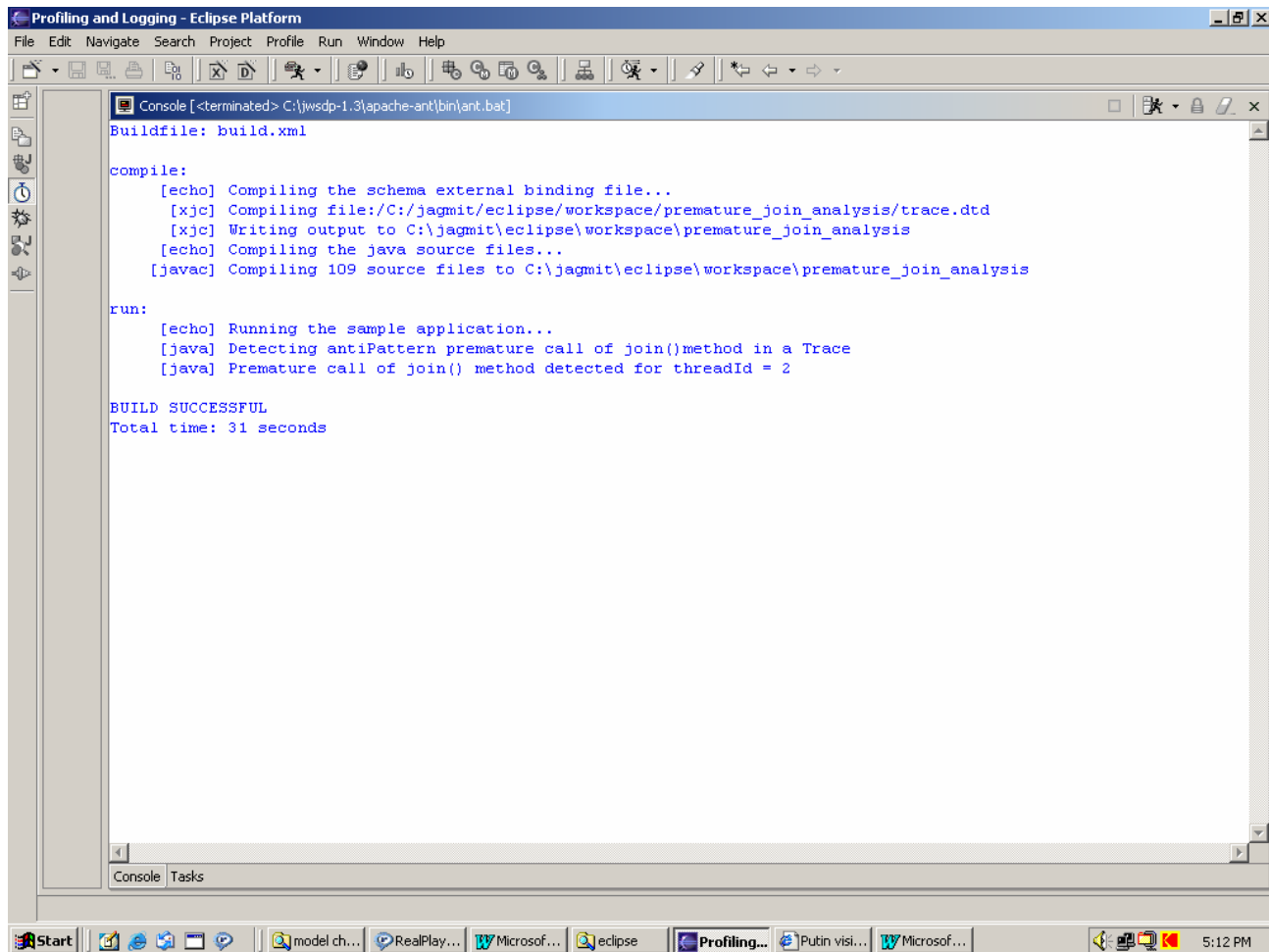
- [SMI00] Connie U. Smith and Lloyd G. Williams, “*Software performance antipatterns*”, In Proc. of Workshop on Software and Performance, Ottawa, 2000, pp. 127-136.
- [SSP] Software Surveyor Project, [Online]  
<http://www.objs.com/DASADA/index.html>
- [MAG99] J. Magee and J. Kramer: *Concurrency: State Models & Java Programs*: John Wiley & Sons, 1999.
- [MEI03] Rob van der Mei, Marcel Harkema, Dick Quartel, and Bart Gijzen “*JPMT: a Java Performance Monitoring Tool*” In Proc. of TOOLS 2003, Urbana, Illinois, USA, 2003
- [MS03] N Markey and Ph Schnoebelen “*Model Checking a Path*” 14th international conference on concurrency theory (CONCUR 2003), Marseille, September 3-5
- [NASA02] Stacy Nelson and Charles Pecheur. “*V&V OF ADVANCED SYSTEMS AT NASA*” Technical Report for NASA January 25, 2004 [Online]  
<http://ase.arc.nasa.gov/vvvhm/reports/FinalNASAREport2.pdf>
- [TR1] Technical Report 1, First intermediate report on formal analysis of MT java applications. Montreal, August 2003.
- [TR2] Technical Report 2, Second intermediate report on formal analysis of MT java applications. Montreal, November 2003.
- [ULR03] Ulrich, A., Hallal, H., Petrenko, A. and Boroday, S. “*Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-*

*file Analysis*” In Proceedings of the IEEE 36th Hawaii International Conference on System Sciences (HICSS-36), Hawaii, USA, January 6-9, 2003.

[RIN01] Martin Rinard, “*Analysis of Multithreaded Programs*”, In Proc of 8<sup>th</sup> International Static Analysis Symposium, Paris, France, July 2001

[JPF] Java Pathfinder “A Model Checker for Java Programs”, [Online]  
<http://ase.arc.nasa.gov/visser/jpf/>

## Appendix A: A snapshot of the console output for “premature join” verification based on custom based detection Approach



The screenshot shows the Eclipse Platform console window titled "Profiling and Logging - Eclipse Platform". The console output is as follows:

```
Console [<terminated> C:\jwsdp-1.3\apache-ant\bin\ant.bat]
Buildfile: build.xml

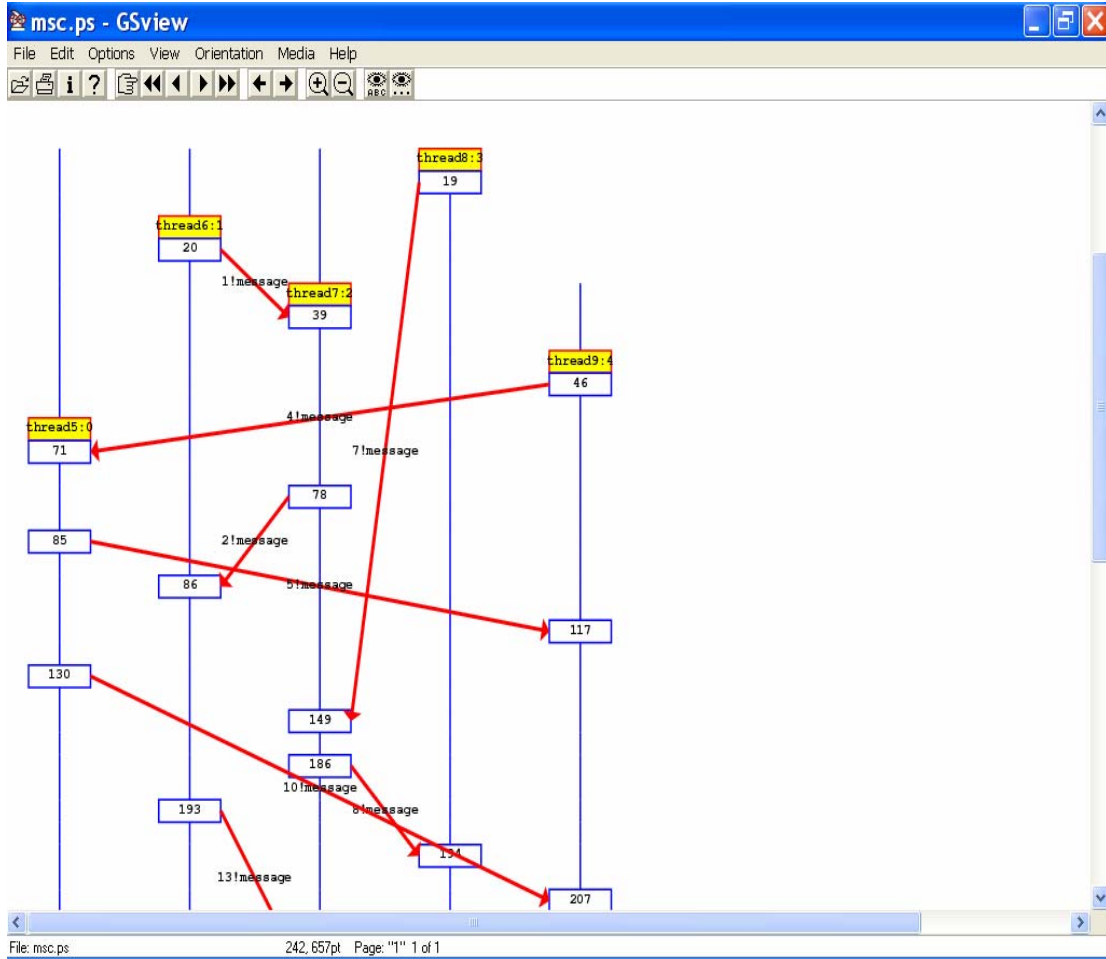
compile:
[echo] Compiling the schema external binding file...
[xjc] Compiling file:/C:/jagmit/eclipse/workspace/premature_join_analysis/trace.dtd
[xjc] Writing output to C:\jagmit\eclipse\workspace\premature_join_analysis
[echo] Compiling the java source files...
[javac] Compiling 109 source files to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

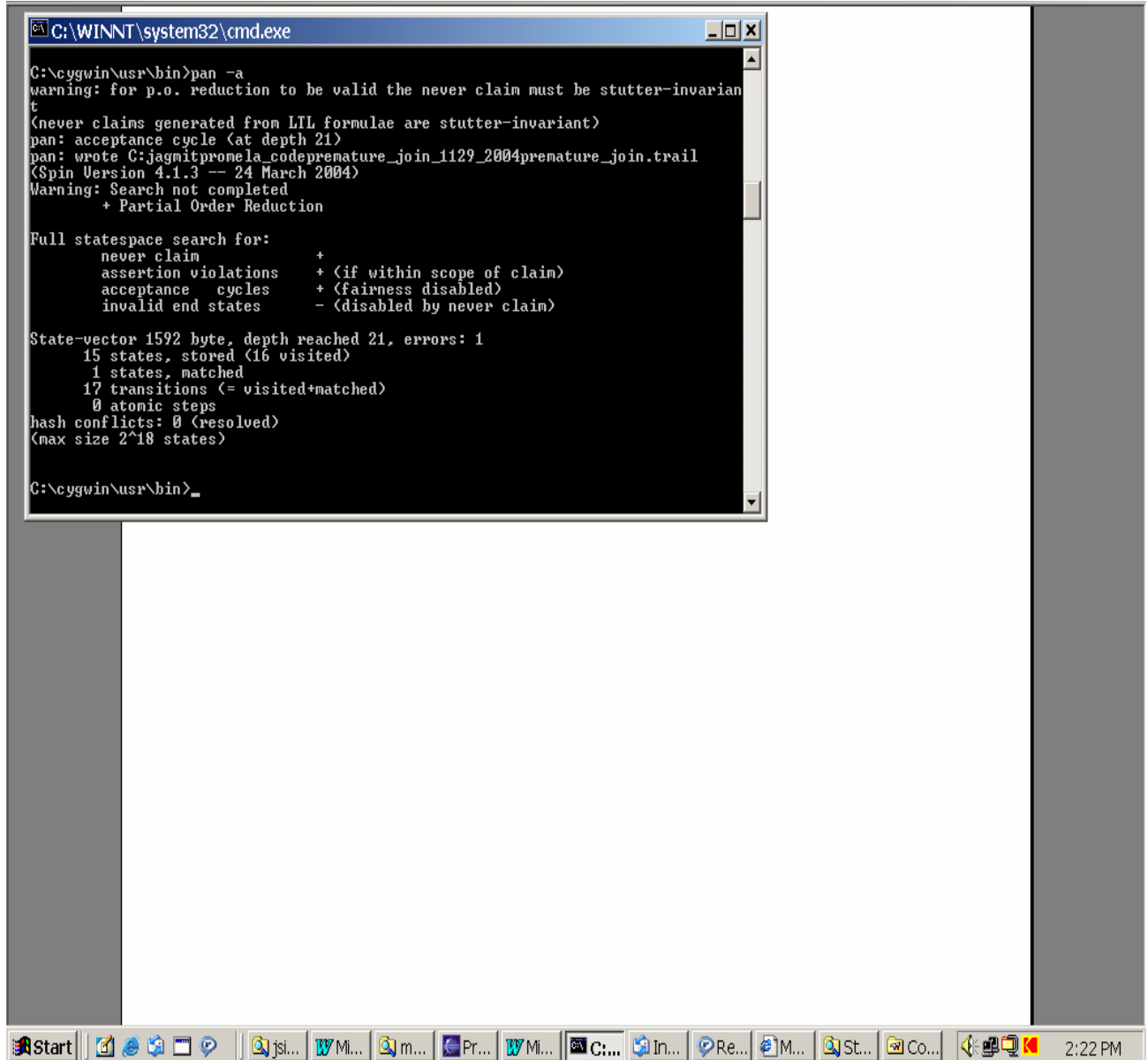
BUILD SUCCESSFUL
Total time: 31 seconds
```

The console window is part of the Eclipse IDE interface, with a taskbar at the bottom showing various open applications like "model ch...", "RealPlay...", "Microsof...", "eclipse", "Profiling...", "Putin visi...", and "Microsof...". The system clock in the bottom right corner shows "5:12 PM".

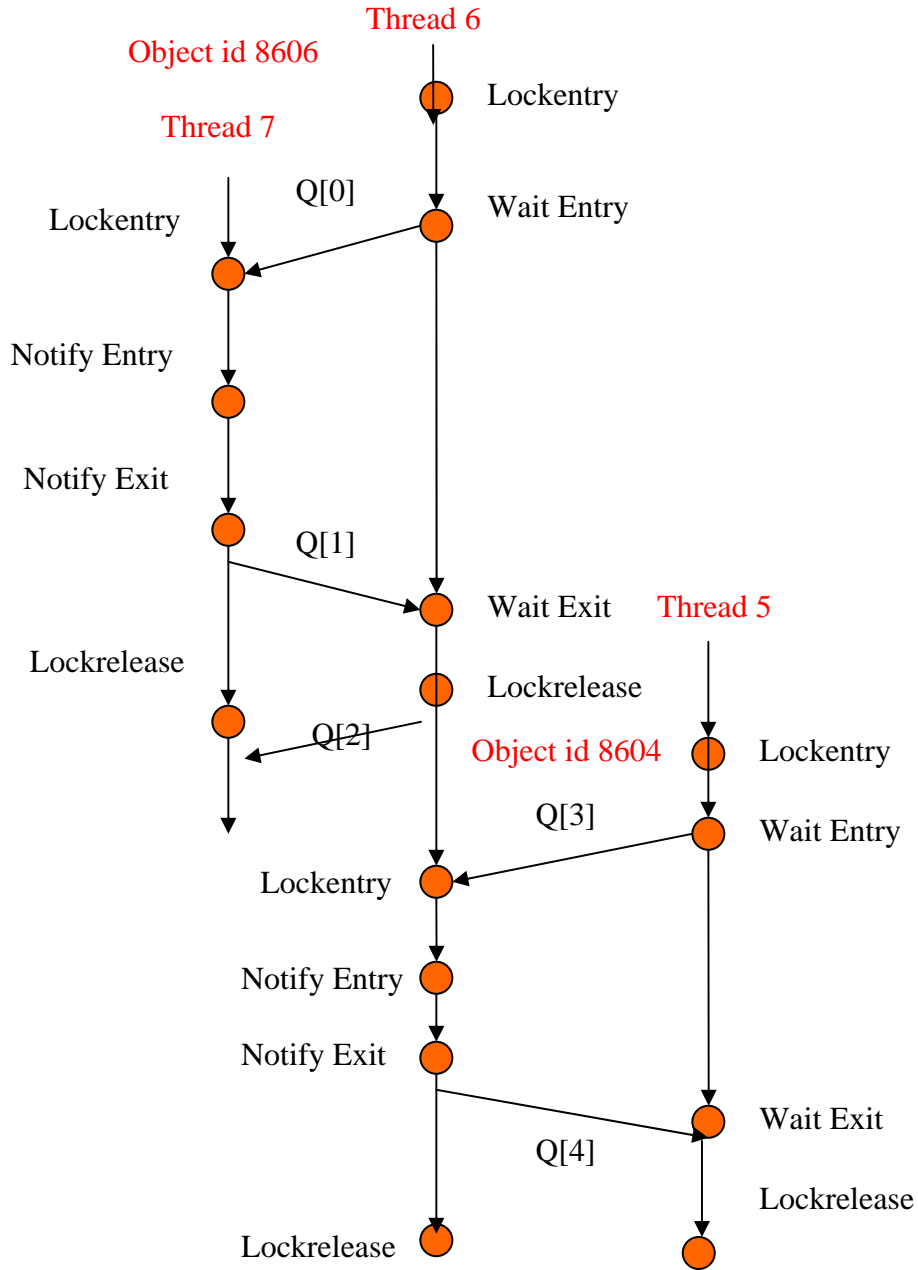
## Appendix B: A Snapshot of the MSC (Message Sequence Chart)



**Appendix C: A snapshot of console output for “premature join” verification based on model checker based approach**



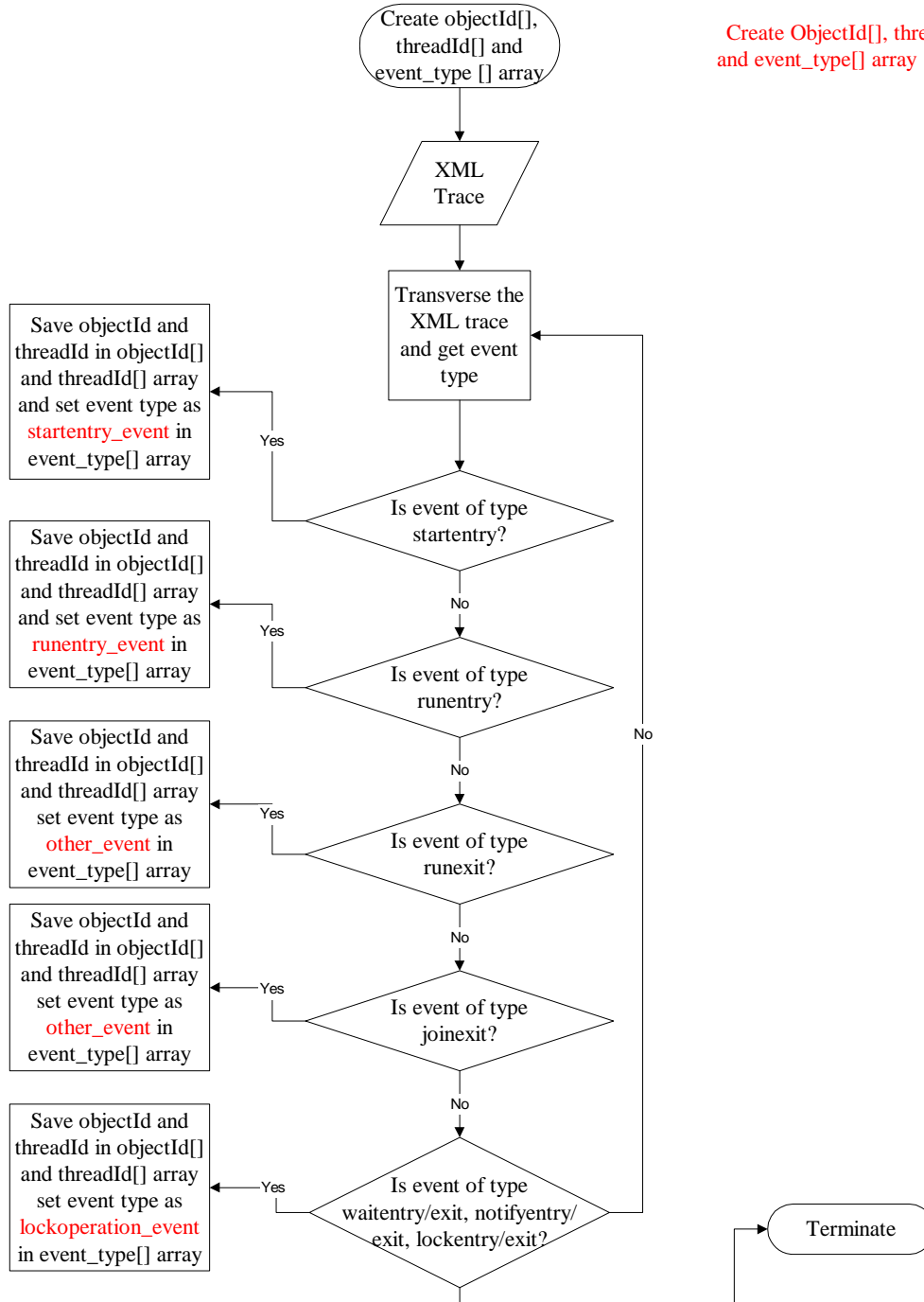
## Appendix D: Ordering of the events on the same Object



## Appendix E Flowchart for the XML-Promela model

The XML trace is transversed and objectIds and threadIds of the events e.g. (waitentry/exit, notifyentry/exit, lockAcquire entry/exit, lockrelease entry/exit), startentry, runentry,runexit, joinexit are saved in objectId[] and threadId[] arrays and there event types lockoperation\_event, startentry\_event,runentry\_event,other\_event,other\_event are saved in event\_type[] array respectively.

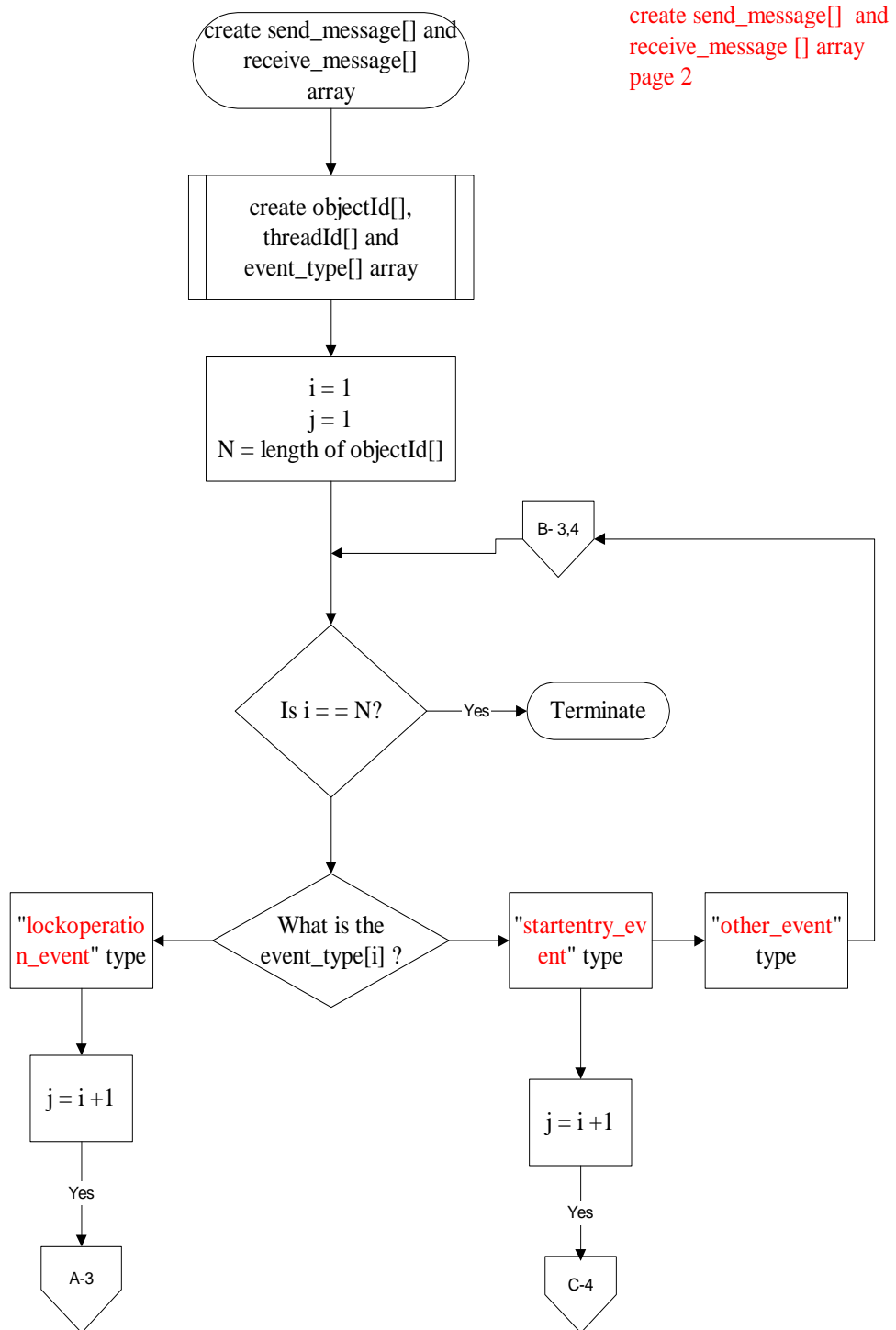
Create ObjectId[], threadId[] and event\_type[] array page 1



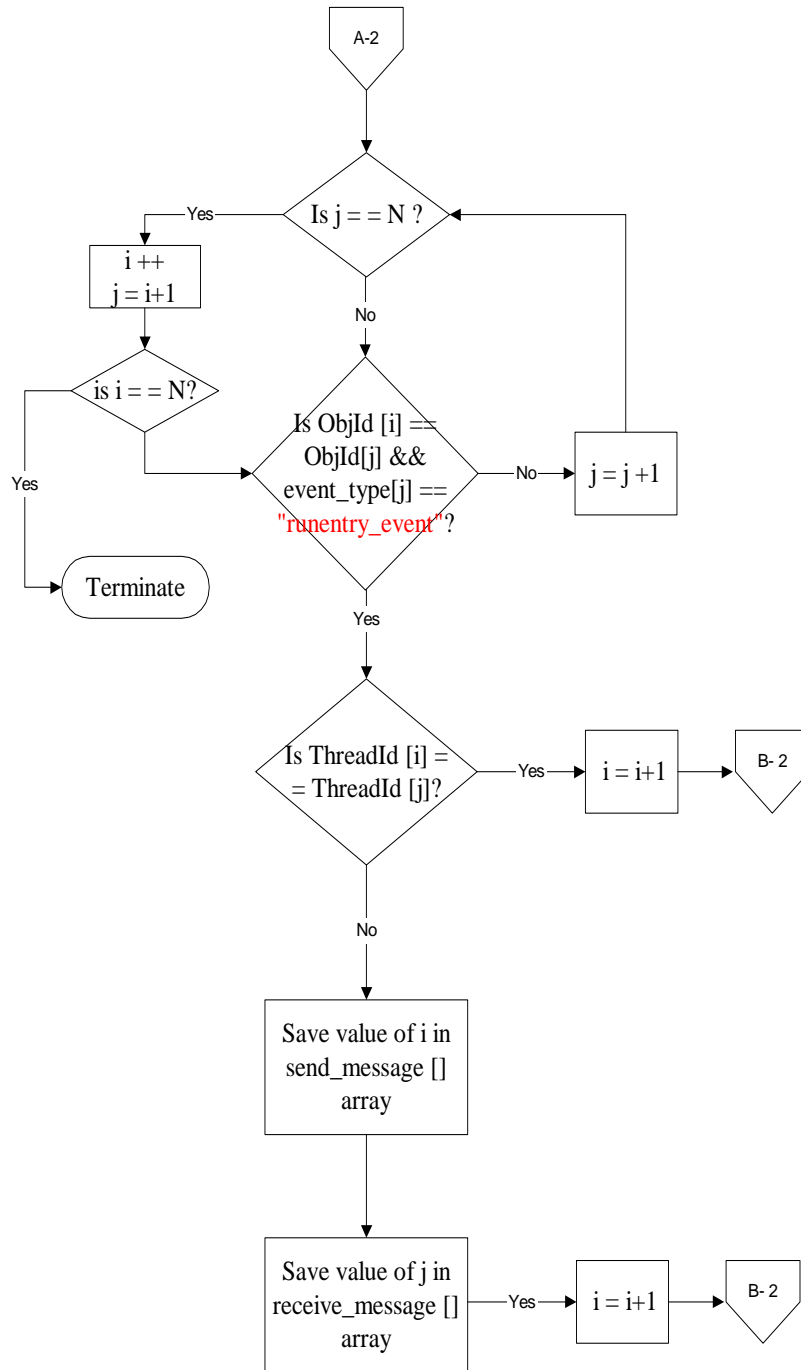




This subroutine find those objectIds and threadIds values in the arrays whose preceding objectId and event type is same ( but in the case of start and run startentry should be preceded by runentry) but different threadId (for sending messages) and save there index values in send\_message[] array and save those index values from which they are different in receive\_message[] array.

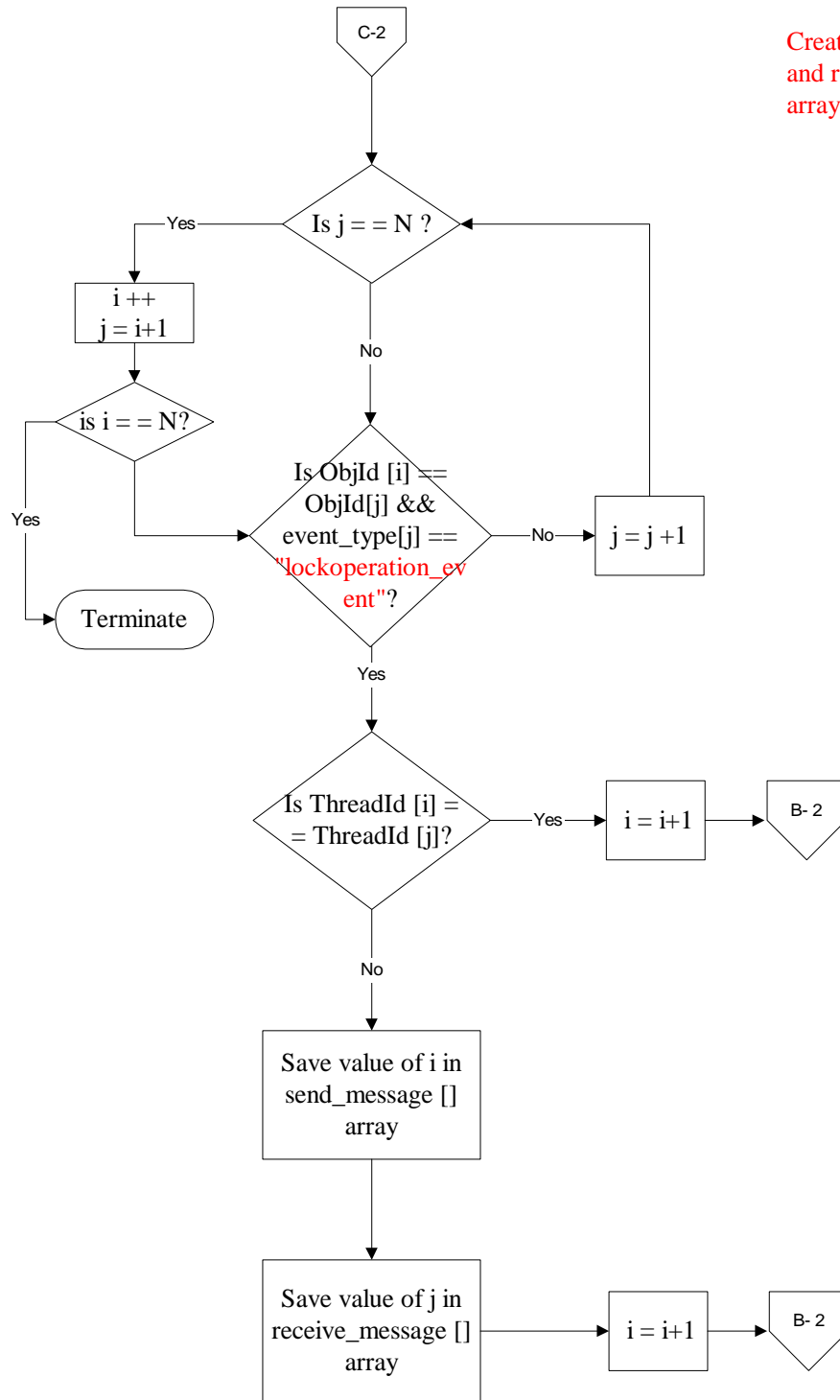


Create send\_message[]  
and receive\_message[]  
array- page 3

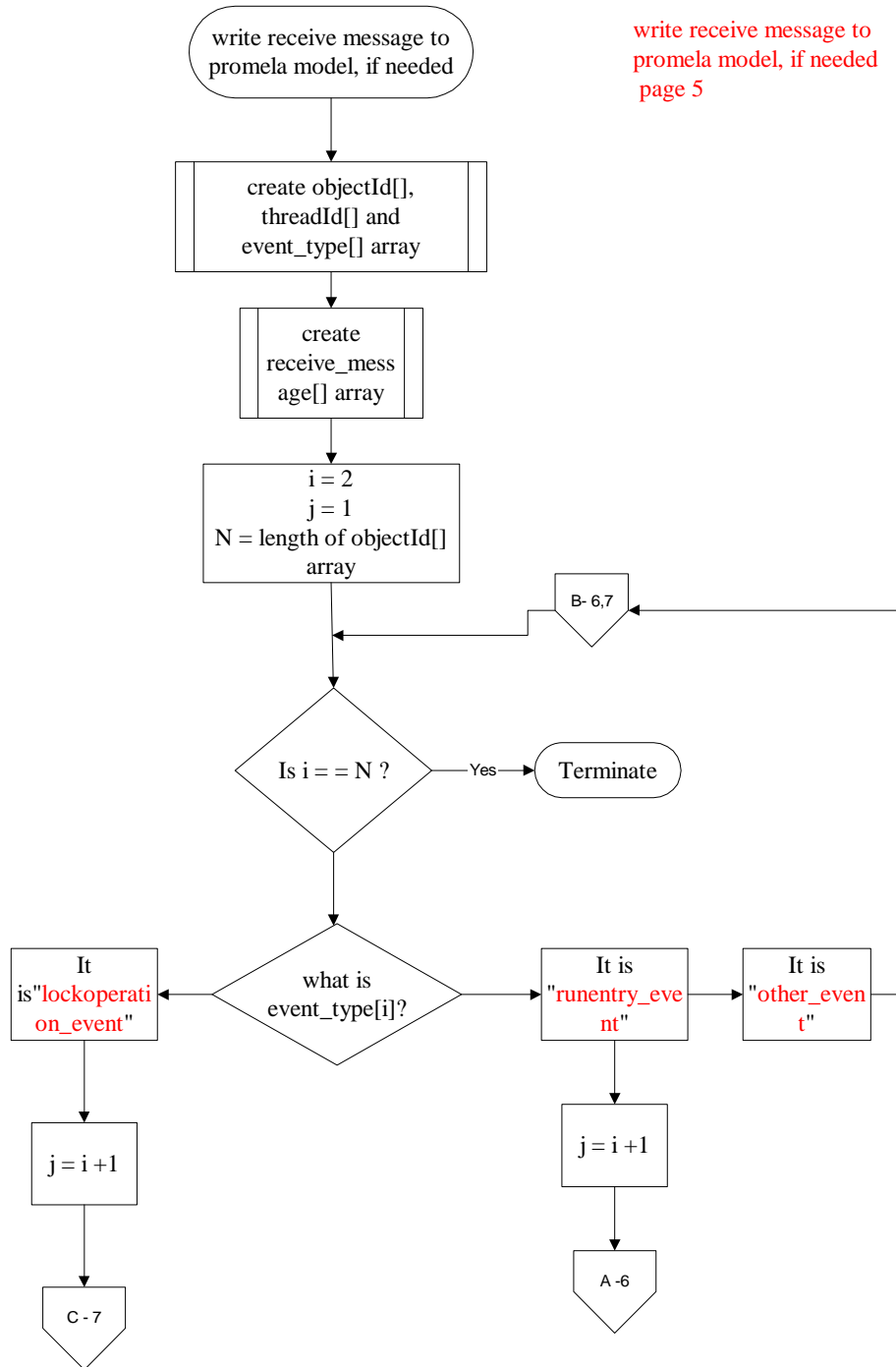




Create send\_message[]  
and receive\_message[]  
array- page 4

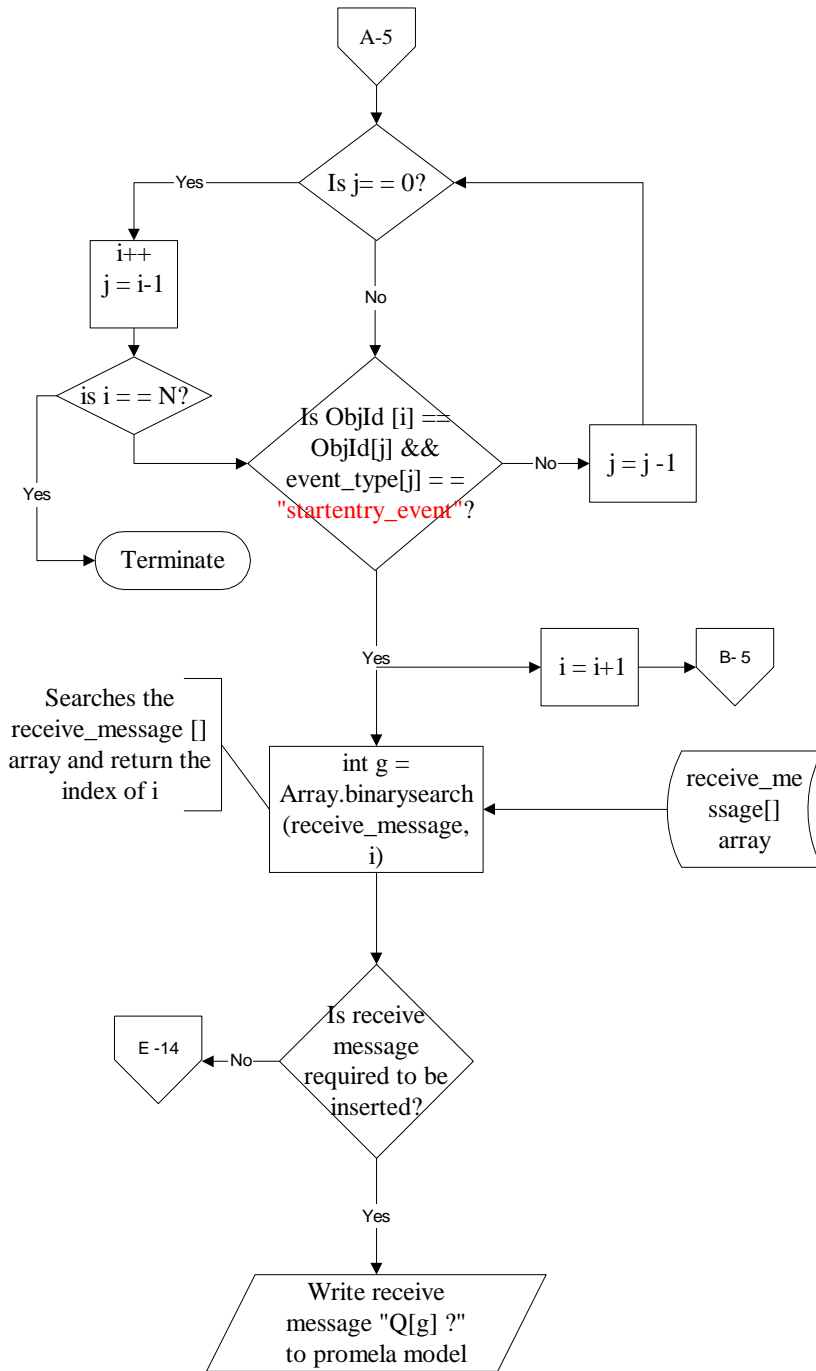


The algorithm of this subroutines is the same as followed before for to create receive\_message[] (page 2, 3 & 4), except that the indexes are not stored in arrays but rather there index position (second index values) are found in receive\_message array. While writing events to the promela model, the receive messages are numbered according to the second index values.





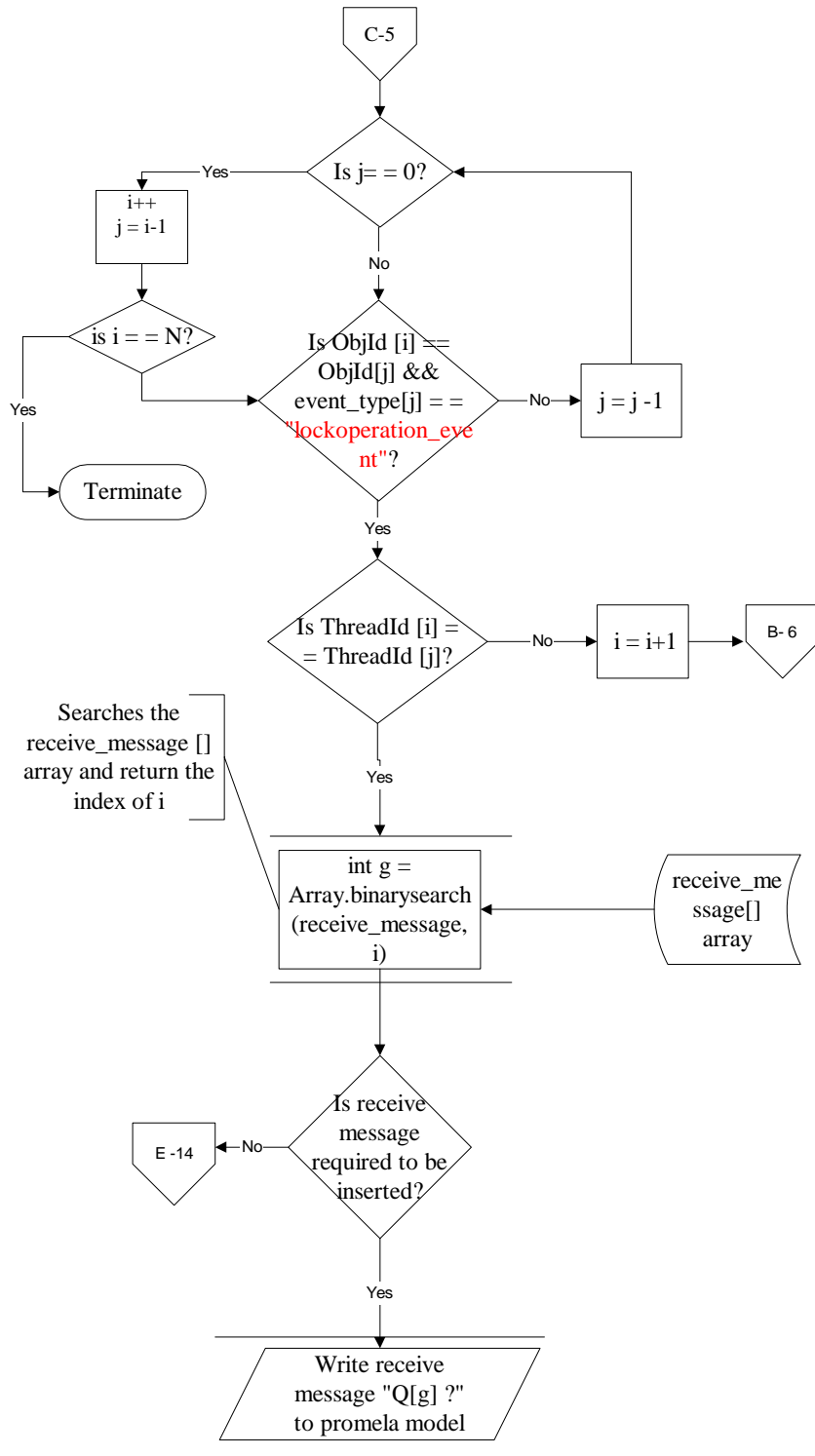
write receive message to promela model, if needed  
page 6



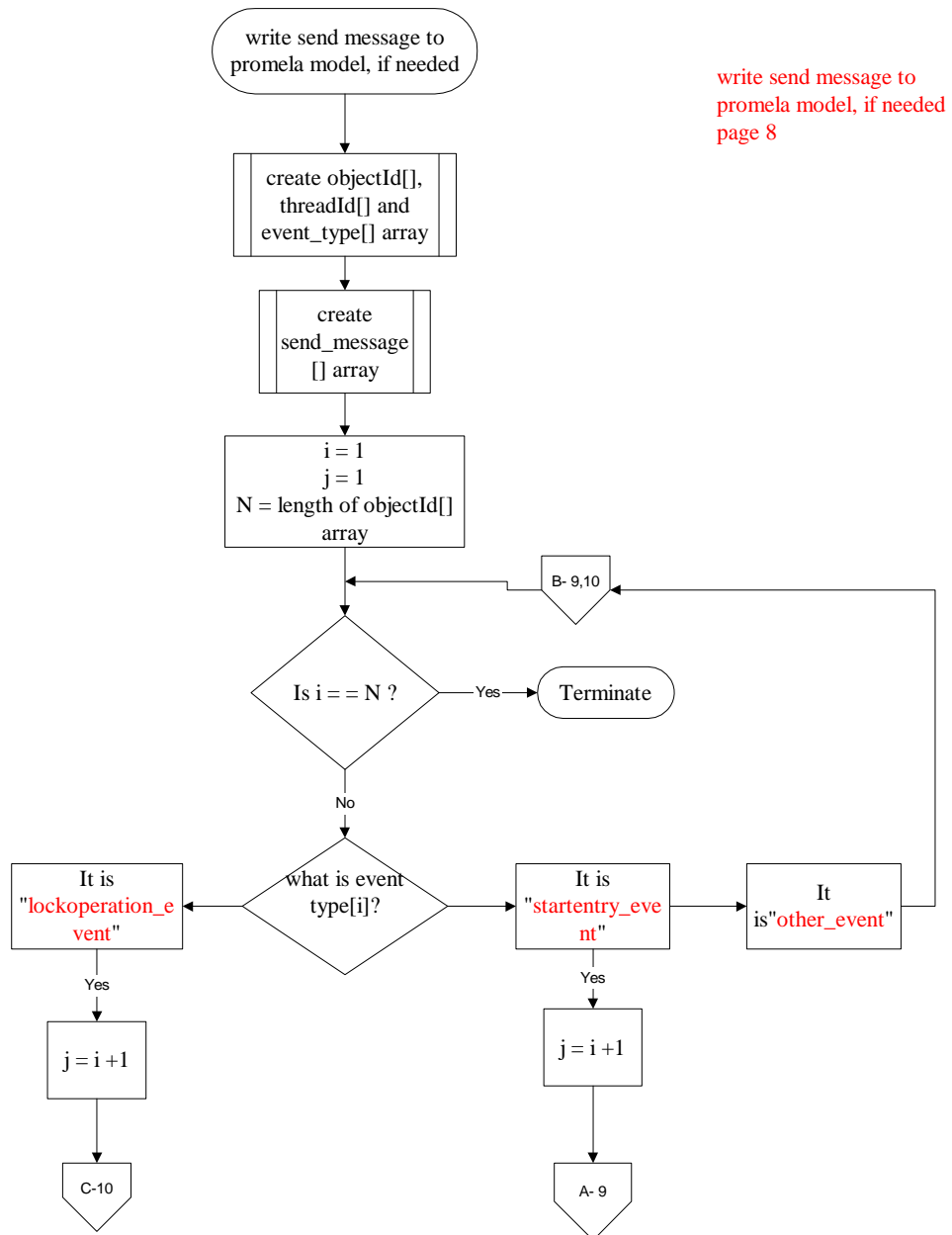




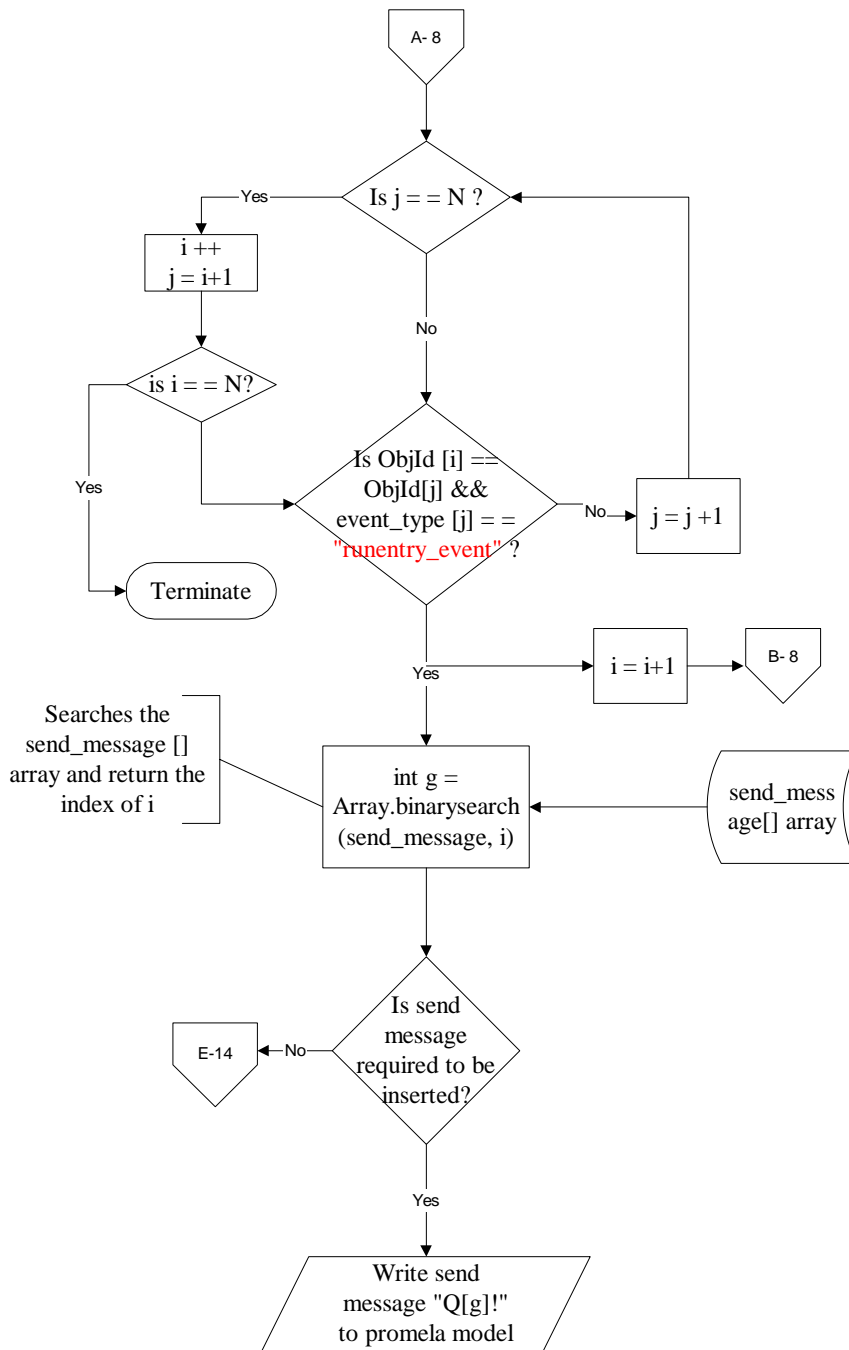
write receive message to promela model, if needed  
page 7



The algorithm of this subroutines is the same as followed before to create send\_message[] (page 2,3 & 4), except that the indexes are not stored in arrays but rather there index position (second index values) are found in send\_message array. While writing events to the promela model, the send messages are numbered according to the second index values.

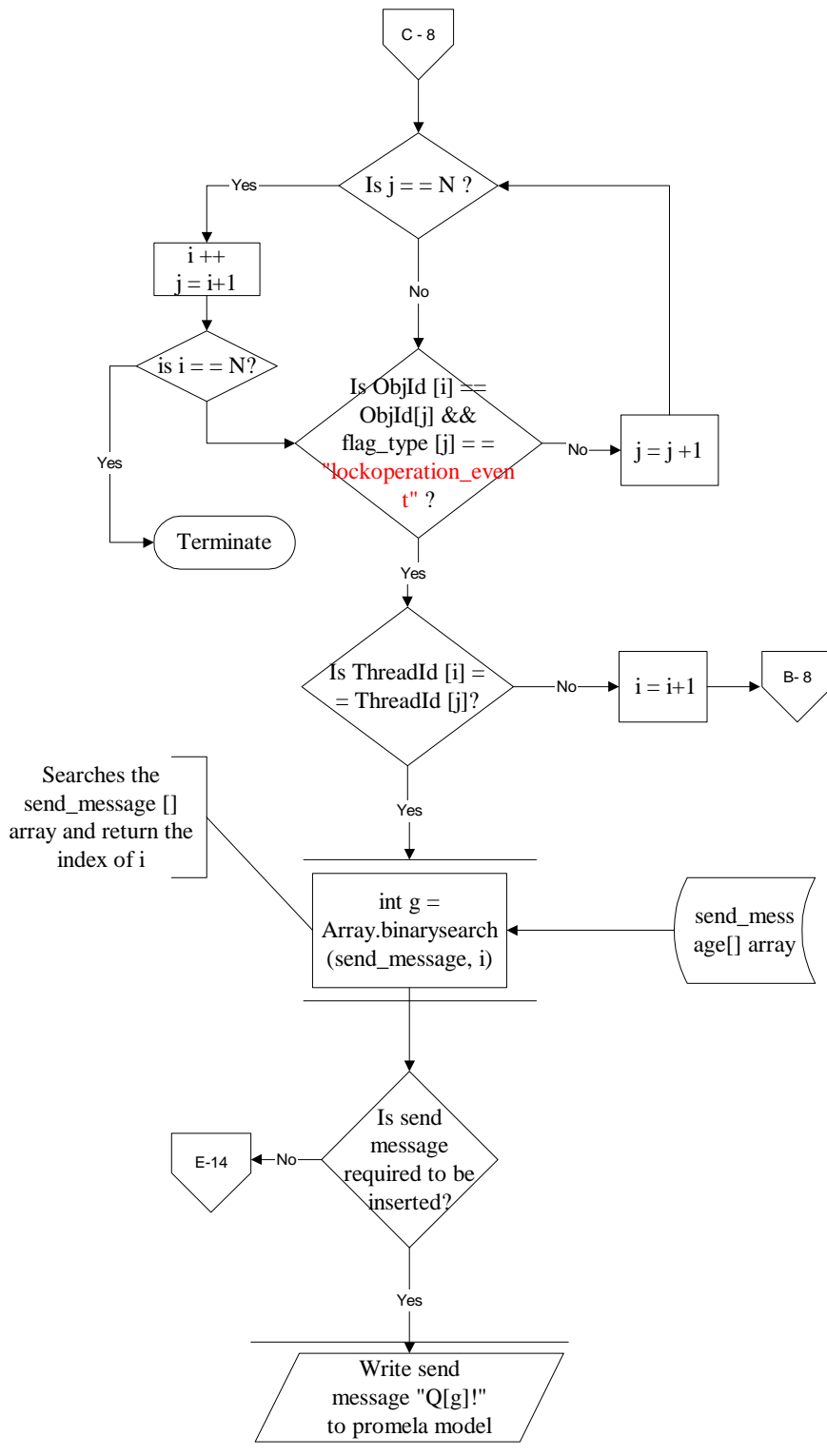






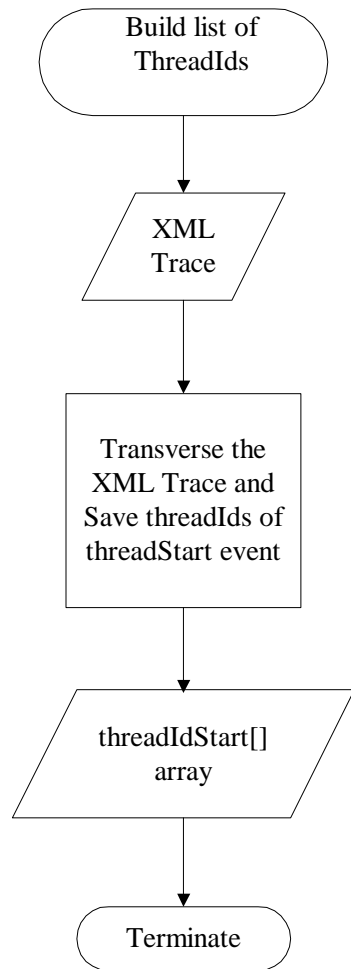
write send message to promela model, if needed page 9





write send message to promela model, if needed page 10

This subrouine build threadIdStart [] array, containing the list of all the threadIds

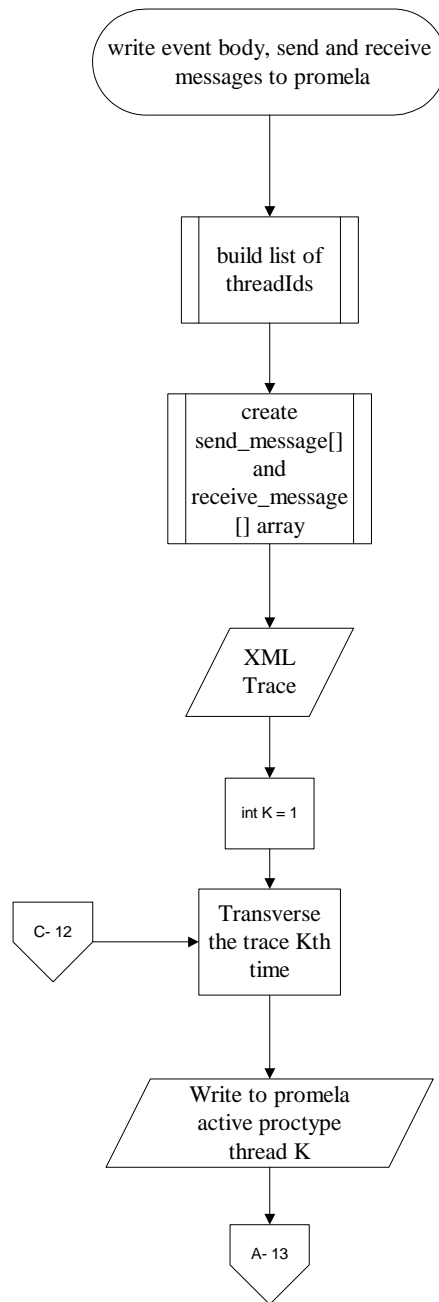


Build list of ThreadIds  
page 11

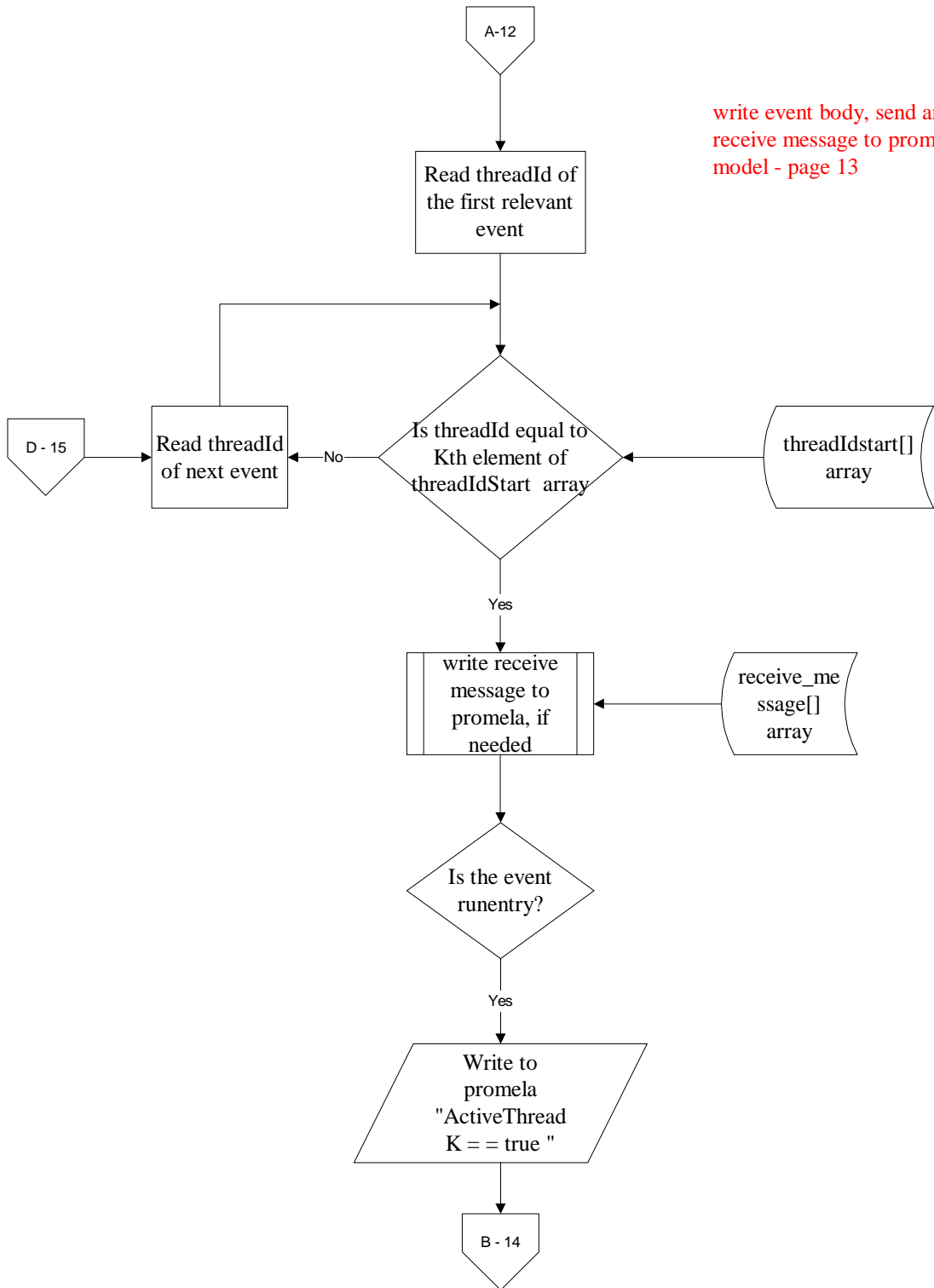




Events of the XML trace are not written (in the same order as in XML trace) to the promela model, but in different order. e.g. all the events of thread (e.g Id 4) are written first, then the events of thread (e.g Id 5) are written second and so on.



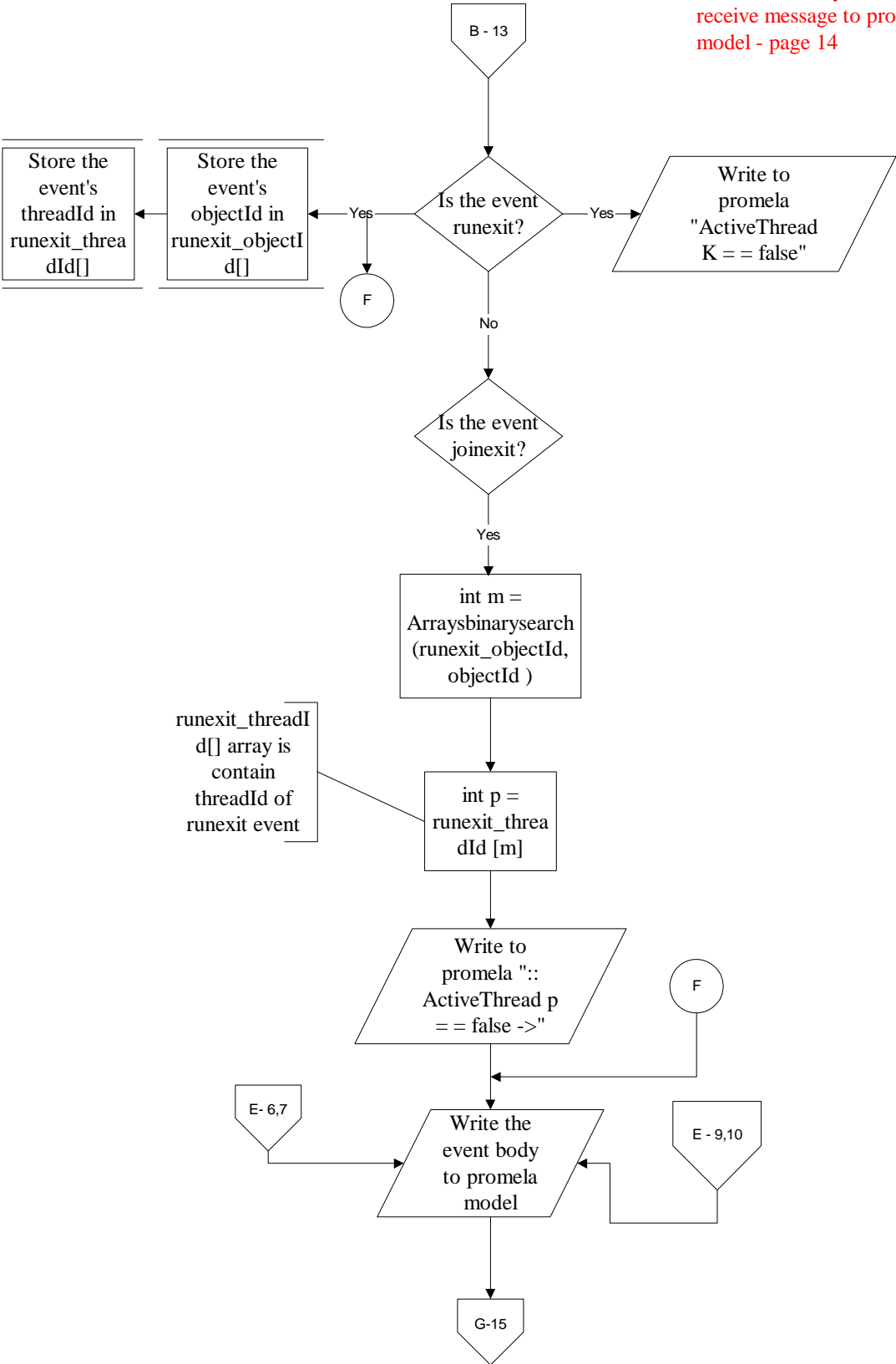
write event body, send and receive message to promela model - page 12



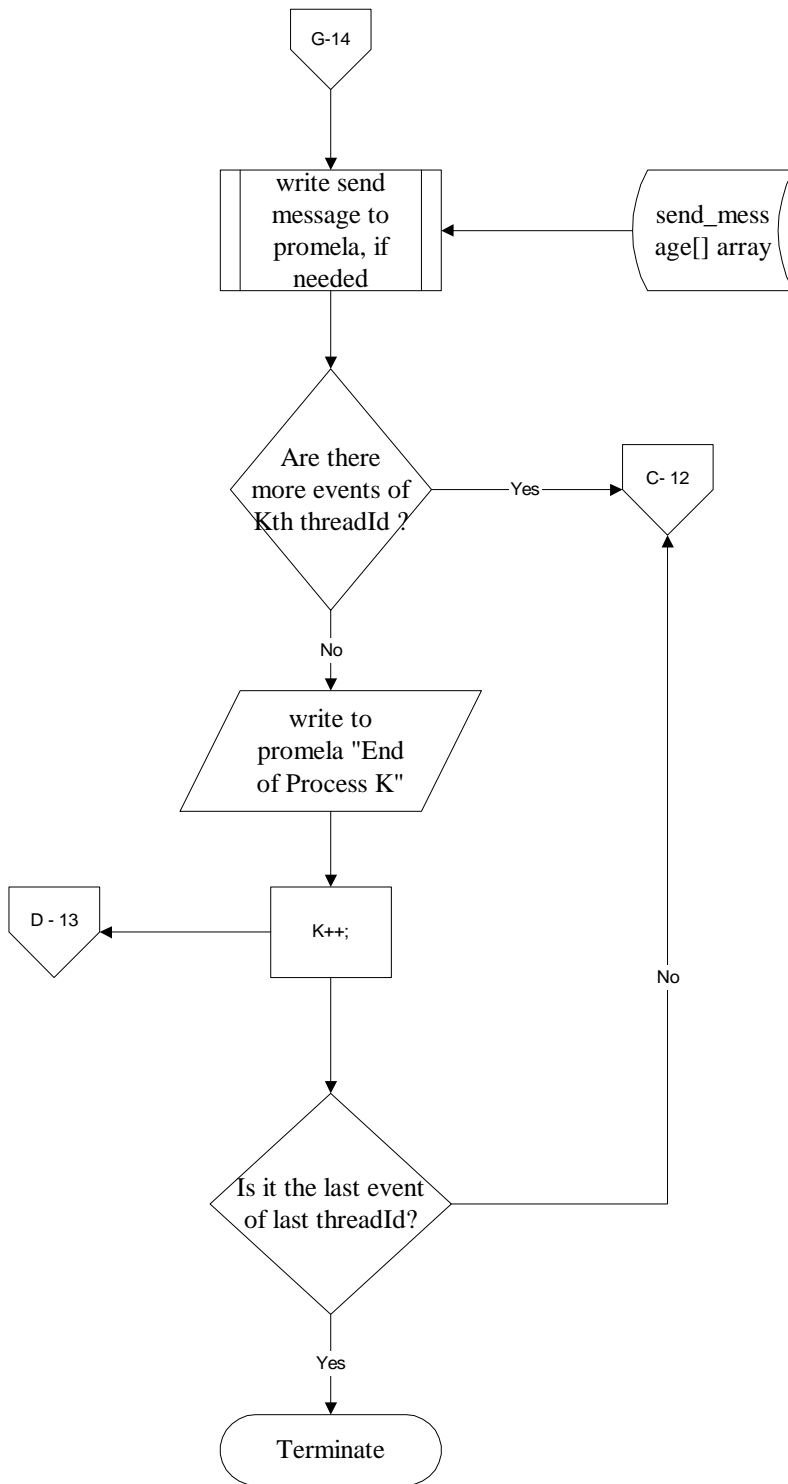
write event body, send and receive message to promela model - page 13



write event body, send and receive message to promela model - page 14



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



The promela model with send and receive messages inserted.  
Here before writing send or receive message to promela model, we check if they are required to be written, if required we insert receive or send message before or after the event body.

The main program - page 16

