Visual Analytics Tool for Java Virtual Machine Execution Traces

MONALI PANDE
Visual Analytics Tool for Java Virtual Machine Execution Traces

MONALI PANDE
Abstract

The usage of multithreaded programs is continuously increasing, which leads to various concurrency issues. The non-deterministic approach of the thread scheduler makes the analysis of such programs complex. Thread-based visualization of the concurrent events helps to analyze a concurrent program efficiently. The extension of such visual analytics jpf-visual tool for regular JVM traces will help Java programmers to better understand and analyze the runtime execution of concurrent programs. AspectJ instrumentation with its lock() and unlock() pointcut extension makes it possible to capture important runtime events information in order to generate the JVM event trace. A successful integration of the JVM trace into the jpf-visual tool is achieved through code refactoring and the use of adapter classes. In this thesis, the implementation of such an approach is shown to analyze the concurrent events using the regular JVM. Such implementation can help to provide a generic approach for the concurrency issue analysis.
Sammanfattning

Acknowledgement

It wasn’t that easy! Thank you the time and situations, for coming hard, testing me and bringing better out of me! It was great learning.

I would like to express my deep gratitude to my supervisor, Dr. Cyrille Artho, for his valuable guidance, consistent encouragement and useful critiques throughout the thesis project. I wish to thank my husband who played an important role during my master studies. His unconditional love, caring, support gave me positive energy to always move on with enthusiasm. I want to thank my parents and in-laws for all their support. I couldn’t have achieved it without their blessings. I can’t forget motivation from my sisters, Sweety and Priya and keeping me positive all the time. Special thanks to my dearest people Nilesh and Sneha, for their consistent motivation and support.

I want to thank all of friends in Stockholm and in India, for their encouragement throughout my study. Moreover, a great help from my dear friends Swapnali and Saranya (also for fruitful technical discussions) for proofreading the thesis will be remembered.

Finally, I would like to thank my little princess Gargi, whose smile always keep me lively and happy.
# Contents

1 Introduction 1
   1.1 Motivation 1
   1.2 Research question 2
   1.3 Methodology 3
   1.4 Contributions 3
   1.5 Scope 3
   1.6 Ethics and sustainability 4
   1.7 Outline 4

2 Background 5
   2.1 Concurrency 5
      2.1.1 Synchronized blocks 6
      2.1.2 Wait/notify/notifyAll 6
      2.1.3 Concurrency issues 6
   2.2 Bytecode instrumentation 7
      2.2.1 Instrumentation using native interfaces 8
      2.2.2 Bytecode manipulation libraries 9
      2.2.3 Aspect-oriented programming (AOP) framework 10
   2.3 jpf-visual 12
      2.3.1 Java Pathfinder 12
      2.3.2 Panels in jpf-visual 13
   2.4 Event ordering 15
      2.4.1 The Java Memory Model 15
      2.4.2 Happens-before relation 15
      2.4.3 Event capture 19
   2.5 Summary 19

3 Related Work 20
   3.1 JBInsTrace 20
Chapter 1

Introduction

Multithreading has become a common programming technique and is widely used for modern software. It allows the programmer to interact with multiple clients and to run multiple computations in parallel. Multithreading enhances the performance of an application by allowing the multiple threads running concurrently. Unfortunately, the debugging of a multithreaded program is often difficult, and simple synchronization errors can lead to various concurrent issues, such as data races and deadlocks. Concurrency errors like deadlocks, starvation, and race conditions can bring down any system in which they occur, and it can take a couple of days or weeks to track them down. Hence, there is a need for an efficient tool to analyze multithreaded programs.

In multithreaded interactive programs, unpredictable interleaving can cause errors. One way to analyze such program is by using print statements and reviewing the text logs. However, debugging and testing of concurrent programs using this traditional approach is difficult due to many threads that can interleave. In addition, text logs are not easily understandable.

1.1 Motivation

The analysis of large concurrent programs is sometimes difficult and time taking due to huge number of thread interleavings. Hence, thread based analysis of events in the execution trace collected at runtime is useful in this context. However, the event trace can be a large text with a lot of condensed information which is not easy to understand. Therefore, visualization of such events along with thread interleaving can be useful to show patterns in the large trace.

The complex visualizations do not always scale well to large datasets. Hence, a visualization tool which provides various events and thread-based
filtering would be helpful to get a better understanding of the program. Such analysis helps software engineers to extract important information on the properties and characteristics of the system. The \texttt{jpf-visual} tool is a visual analytics tool that highlights the events and provides thread-based functionality in the execution trace. In addition, it allows the user to provide interactive visualization.

The \texttt{jpf-visual} tool uses error traces obtained from Java Pathfinder (JPF) for runtime event visualization. JPF produces a comprehensive error traces to analyze concurrency issues. However, it is susceptible to combinatorial explosion, the configuration system for JPF modules is complex, and JPF runs into memory limitations for large state spaces. Additionally, its overhead is exponential, hence, JPF cannot be used for large program verification. On the other hand, large traces are also generated in programs running on the Java Virtual Machine (JVM). To provide an alternative when given overhead of the state space exploration and make the \texttt{jpf-visual} tool applicable to any Java program, there is a need to generate error traces with the help of regular JVM.

1.2 Research question

The project mainly focuses on integration of execution trace from the regular JVM into the \texttt{jpf-visual} tool in order to visualize the following runtime events:

1. thread creation/death
2. lock/unlock
3. wait/notify
4. field access
5. method call
6. line-by-line display of execution trace

In the current project following research questions are addressed:

- To what extent can data from a regular virtual machine be mapped on a trace generated by the software model checker JPF?
• To what extent the information with respect to the \textit{happens-before} relation is obtained correctly? It includes thread start, join, wait/notify, locking.

1.3 Methodology

The above-mentioned research questions will be explored by the following methodology.

On the front-end side, the code is well structured, therefore refactoring of the code in \textit{jpf-visual} and use of unified interfaces provides different events data structures. Implementing this common interface using adapter classes provides all the events data used in the \textit{jpf-visual}, both from the current JPF error traces and JVM error traces.

In order to generate a JPF compatible error trace for the regular JVM, the AspectJ instrumentation tool is used on the back-end side. AspectJ captures the runtime event information during program execution in the JVM. The captured data is temporarily stored in a singleton instance. Further, these global variables are used to process captured event data into data structures that are used by \textit{jpf-visual} for visualization.

1.4 Contributions

The \textit{jpf-visual} has been shown to be useful to understand the concurrent issues in detail by highlighting only key data in a large JPF trace. The contribution of this thesis is to provide the JVM trace as an alternative to a JPF trace in the \textit{jpf-visual} tool for thread-based visualization of the runtime events. In addition, bytecode instrumentation such as AspectJ is used to obtain these runtime events using regular JVM. An additional contribution is creating a mirror data structure like JPF using adapter classes for successful integration of JVM trace into a \textit{jpf-visual} tool. The visualization of the happens-before relation in the \textit{VA4JVM} is provided along with its limitations. Finally, the report addresses limitations of the visual analytics tool for JVM after its evaluation, along with its future scope.

1.5 Scope

The thesis focuses on the successful integration of the JVM trace in the \textit{jpf-visual} tool. The thesis evaluates, to what extent is it possible to generate
such JPF compatible trace through AspectJ instrumentation in JVM. Such that, the extended tool highlights the different runtime features along with the actions belonging a particular thread at a particular time. The visualization of the happens-before relation to some extent is provided along with its limitations in the report.

This thesis work addresses the problems of recording JVM data in a way that can be presented visually in a framework that was originally designed for JPF, and also provides a graphical view of such data. The data record may not in all cases accurately reflect low-level ordering of operations in the code, due to inherent limitations of event recording with code instrumentation (see section 2.4.3).

1.6 Ethics and sustainability

The different runtime features of a JVM trace captured using AspectJ give a holistic understanding of concurrent program executions. From an ethical perspective, visualization of the runtime events, tracking thread activities and monitoring the acquisition and release of lock events on a different thread helps software developers in program analysis. Such a visual analytics tool for JVM trace helps the software research community to understand quickly why a specific error was reported. In addition, it gives a better understanding of concurrent issues and inspires, the research community to use the AspectJ as an instrumentation tool to capture the concurrent event in regular JVM.

1.7 Outline

The remainder of this report is organized as follows: Chapter 2 is the background section which goes through the most important concepts in order to understand the method and the results; Chapter 3 discusses the literature studies of different visualization tools for execution trace; Chapter 4 gives information about the different runtime features captured using AspectJ; Chapter 5 describes the processing of these events to create the data structure and Chapter 6 provides evaluation of the tool by analyzing various concurrent programs. Finally, the results are presented, discussed, and a conclusion is stated.
Chapter 2

Background

This section provides background information about the important concepts required to understand the visual analytics tool for JVM, which is an extension of the existing jpf-visual tool. In the report, the visual analytics tool for the JVM is referred to VA4JVM. This section starts with the importance of the concurrency issues in software analysis followed by the description of different instrumentation techniques, which help to extract interesting event information through the regular JVM. Further, this section introduces the jpf-visual tool for the visualization of concurrent events. Understanding this tool is essential for the successful integration of JVM trace with existing jpf-visual tool. Finally, this section explains the different rules of happens-before relation that gives a partial ordering to the concurrent events.

2.1 Concurrency

The use of multithreaded programming is increasing nowadays, which may lead to many concurrent issues such as race condition, deadlocks etc. The analysis of concurrent events is important, as its execution order may change the output of the program. However, the analysis of these events is expensive due to the huge number of thread interleavings. It is not feasible to try all of the interleavings, as only a few of them cause a fault. The non-deterministic thread scheduler and the interleaving runs under different environmental conditions, making the analysis of these multithreaded programmes difficult.

In concurrent programming, threads are allowed to run multiple events at the same time. Furthermore, additional language features given below are required to operate the thread safely on shared data at the same time.
2.1.1 Synchronized blocks

Synchronized blocks can be used to prevent multiple threads from accessing shared resources concurrently. When a currently running thread reaches the synchronized block, then that running thread acquires a monitor. If a monitor is already used by the other thread, then it blocks the running thread. The acquired monitor will be released when the thread leaves synchronized block.

2.1.2 Wait/notify/notifyAll

The wait/notify/notifyAll methods are defined in `java.lang.Object` library and help to interact with monitors. The `wait()` method puts the current thread to sleep which is wake up by `notify()` method, and `notifyAll()` method wakes up all threads. However, waiting thread sometimes may be woken up for no reason, such a spurious wakeup can occasionally be a source of bugs [4]. The `wait()` method temporarily releases the monitor which is held by a thread on which `wait()` method is called. The `wait()` and `notify()` methods are used when a thread needs to wait for a certain condition to become true.

Copty and Ur [5] mentioned the locations at which most of the runtime concurrent events, described in the Section 1.2 occur are as follow:

1. Synchronization functions, such as `join()`, `start()`, `wait()` and `notify()` primitives.
2. Synchronization block or method.
3. Shared variable accesses.
4. At the entry to every method.

The different concurrency issues occur while multiple threads accessing these concurrent events are mentioned below.

2.1.3 Concurrency issues

Thread interleavings in the multithreaded program make it difficult to analyze various concurrent issues. These issues bring down the system and require a longer time to fix. Thus, it is essential to analyze the following concurrent issues [6].

- A **deadlock** occurs when threads are waiting for each other forever which results in infinite waiting.
• A **starvation** occurs when a thread never gains access to resources. It results in, thread is unable to make progress and never allowing the program to finish.

• A **race condition** occurs when the correctness of the program computation depends on interleaving of the multiple threads by the runtime. It can cause concurrent programs to fail in unpredictable ways.

• A **data race** occurs when two threads access the same memory location and at least one of the accesses is a write. In addition, there is no restriction provided on ordering between the accesses. A data race may result from a programming error which may be difficult to debug because it can exhibit different behaviors when executed repeatedly with the same set of inputs [7].

• A **livelock** occurs when two threads are engaged in responding to each others actions. In this situation, the program is unable to proceed as threads can cause a loop similar to something between a deadlock and starvation.

Analyzing programs typically rely on abstraction of the program execution such as trace, which contains operations and events that a program executes. It allows to reason about possible executions and serves to check for concurrency properties. In order to get all the concurrent event information in the JVM execution trace, it is necessary to modify bytecode of the program using a code instrumentation tool. This extracted information further can be used to analyze the program efficiently.

### 2.2 Bytecode instrumentation

Compiled Java classes are stored in a `class` file format that contains bytecodes, i.e. instructions that are interpreted by the JVM. Bytecode instrumentation[8] is a process which allows adding new functionality to the program by modifying the bytecode of classes before they are loaded by the JVM. The bytecode instrumentation technique permits capturing application, specific events and gives the data regarding critical sections of the application[8]. Hence, it is useful in dynamic program analysis to find and understand the concurrency related errors by observing and monitoring interesting events information in the program execution. The different instrumentation techniques are given below:
2.2.1 Instrumentation using native interfaces

- **Java Virtual Machine Debug Interface (JVMDI)**
  JVMDI [9] is used by debuggers which give a way to examine the execution state and also control the program execution in JVM. JVMDI client can be informed important occurrences through events and it uses different functions to query and control the application described in JVMTI [2.2.1] which is an extension.

- **Java Virtual Machine tool interface (JVMTI)**
  JVMTI [10] is a native programming interface which enables development of portable profiling and debugging tools through an extension of JVMDI. Using event callbacks from JVMTI it is possible to capture most of the runtime events. It has the advantage that it does not need its own code to generate these runtime events as mentioned in Section [1.2] except last runtime event *line-by-line display of execution trace*. To obtain all these events JVMTI [10] can signal a variety of profiling events as given below:

  - **ThreadStart(Thread t):** Triggered when a thread t is started.
  - **ThreadEnd(Thread t):** Triggered when a thread t is terminated.
  - **MethodEntry(Thread t, Method m):** Triggered at the entry of method and m.isNative() method checks whether m is a native method.
  - **MethodExit(Thread t, Method m):** Triggered at the exit from methods.
  - **FieldAccessed:** triggered when a predetermined field is accessed.
  - **VMDeath():** It gives the notification about the termination of JVM which ensures that no JVMTI events will occur after that.

The JVMTI is conceptually the simplest way of getting events, but events are processed as native code, so it may be more complicated to write event handlers. It is not portable across different operating systems and may not be available in all JVM implementations. It cannot give information about the control flow, because branch instructions do not trigger events. To know which part of the code has been executed in the presence of branches, the execution of the beginning of each basic block...
needs to be tracked. JVMTI is not suitable for this unless a breakpoint is set at each possible branch. However, it is probably impossible to set breakpoints for the entire application.

2.2.2 Bytecode manipulation libraries

BCEL (Byte Code Engineering Library), ASM, and Serp are open source bytecode libraries in Java available for bytecode instrumentation purposes.

- **BCEL**
  BCEL [11] gives users an efficient way to analyze, create, and modify bytecode of a set of Java class files. The information about fields, methods and bytecode instruction can be extracted from these classes. JustIce [11] is a bytecode verifier written with BCEL, which gives information about what is wrong with program code than the standard JVM message.

- **SERP**
  SERP [12] is an open source Java bytecode manipulation framework. A set of high-level APIs is provided by the SERP framework, which allows to manipulate bytecode of class member fields and methods. However, users need to have knowledge of the class file format and JVM instructions to manipulate bytecode. In addition, SERP is slow as it fully parses loaded class files and its toolkit is not thread-safe [12].

- **ASM**
  ASM [13] is Java bytecode manipulation framework and performed efficient analysis by modifying bytecodes of a set of classes directly in binary form. ASM uses the visitor design pattern. A class visitor is defined by interface `ClassVisitor` [14] and is used to override the components including superclass, interface, attributes, fields, and methods. ASM provides an object called `ClassReader` to parse the class and generate events. ASM class `ClassReader` [13] reads the bytecodes of an existing class and call corresponding visitor component on a class visitor. Once the class has been parsed, `ClassWriter` is used to consume the events and convert them back to the class byte array. ASM library is small in size and more efficient than BCEL and SERP for bytecode instrumentation purposes [14]. Bytecode instrumentation using ASM can capture all kinds of concurrent events and is fully portable. However, use of ASM for complex programs is challenging as it requires the knowledge of bytecode syntax.
2.2.3 Aspect-oriented programming (AOP) framework

AOP [14] allows the creation of generic instrumentation rather than other instrumentation technology which requires expertise of bytecode syntax. The core problem that AOP tries to solve are multiple concerns (known as crosscutting concerns) such as security, logging, and transaction handling in modern object-oriented systems. AOP extracts and separates these cross-cutting concerns into reusable aspects that are implemented in one place and then used all around the code. AOP supports such dynamic crosscutting concern by keeping the original functionality of the targeted program. It captures crosscutting structures in a modular way [5]. In AOP, as a program executes, it triggers certain event which is referred as a join point and allows to introduce new code called as advice immediately before or after these points [15]. Following are the different instrumentation tools based on AOP framework.

- **AspectJ**
  AspectJ [16] is an extension to the Java programming language hence, every valid Java program is also valid in AspectJ program and it is not that difficult for the developer to understand. AspectJ provides a modular implementation of a range of crosscutting concerns and adds a new concept to Java, a join point. AspectJ join points designate the specific points during program execution to get interesting information about the program [17]. It deals with crosscutting concerns with the following constructs: pointcuts, advice, aspects, and inter-type declarations which affect programs class hierarchy. Aspects encapsulate these new constructs and define a behavior that provides tracing, profiling within the application. AspectJ allows to enable and disable the functionality when desired.

AspectJ has the following core concepts

- **Join Point** is a well-defined point in the execution program.
- **A Pointcut** is a regular expression that matches the join point and certain values at those join points.
- **Advice** is a small pieces of code that is executed at a particular join point. It is like a Java method but cannot be called directly from the application.
- **Aspect** is a class which contains pointcuts, advice, and inter-type declarations. The pointcut can be evaluated as boolean expressions (AND, OR and NOT).
AspectJ has three kinds of advices:

- **Before advice** is executed before the join points that trigger it.
- **Around advice**: is executed around the join points that trigger it.
- **After advice**: is executed after a specific join point that trigger it.

The crosscutting behavior is implemented using an advice which consists of pointcut and body of code. AspectJ provides `thisJoinPoint` variable which contains both static (method name, class name and type information) and dynamic information (parameter values, virtual call targets and field values) unique to the join point [15]. In AspectJ, a piece of advice is defined to execute most of the runtime events mentioned in the above introduction section. However, traditional AspectJ did not capture the acquisition and release of locks caused by synchronized block. Bodden and Havelund [18] describe the extended AspectJ which enable use of AspectJ to analyze most of the concurrency related error by providing `lock()` and `unlock()` pointcuts.

- **AspectWerkz**

  AspectWerkz [19] weaves classes at load time using load time bytecode modification. The classes loaded by any class loader except the bootstrap class loader is hooks in and weaves by AspectWerkz. It has a rich join point model and introductions are implemented using the concept of Mixins, which is a way of faking multiple inheritances, methods, and fields that can be added to the class. It uses plain Java to write aspects, advice, and introductions. Moreover, the target classes can be regular Java objects. AspectWerkz is dynamic as it is possible to add, remove and restructure advice and to swap the implementation of introductions at runtime. Aspects can be defined using runtime attributes.

- **Frama-C**

  Frama-C [20] is a software analysis platform which analyzes a source code of C program, to conduct program verification of industrial size. It has a set of plugins for safety and security-critical software analysis. Frama-C enables collaborative verification across cooperating plugins by integrating a shared kernel, data structures and their compliance with a common specification language. Frama-C [20] has shown its adaptability within the research community.

The JVMTI and ASM bytecode instrumentation tools are more flexible in capturing the runtime concurrent events, whereas, their implementation complex-
ity is high. The developer must be highly skilled in the low-level syntax of bytecode in order to implement it effectively without corrupting the original application. However, AspectJ helps developers to focus on the concurrent issue and allows to concentrate on the instrumentation at the point of interest [8]. Hence, AspectJ is selected as an instrumentation tool in this thesis work, rather than using complicated instrumentation techniques which requires lots of expertise. In addition, AspectJ is an open source project, hence its implementation is widely accepted. As the code is available to all, it is easy to find and fix a bug, if any.

2.3 jpf-visual

The jpf-visual [1] tool is a visual analytics tool for JPF execution traces which provides information about concurrent events. jpf-visual helps to highlight important event information in the execution trace, which is helpful for the analysis. The developers can work efficiently by focusing on the analysis of a particular event with the help of different filtering options. Furthermore, it helps in the analysis of concurrent issues such as deadlock, since JPF provides program trace leading to a property violation such as Assertion-Errors, DeadlockedProperty, and UncaughtExceptions [2].

2.3.1 Java Pathfinder

JPF [21] is an explicit state model checker for Java program and built on top of the unmodified JVM. It is an extensible software analysis framework for Java bytecode which gives the program trace. JPF perform such analysis, by an efficient analysis through the program state space, different thread interleaving, backtracking, and by providing partial order reduction. JPF checks for concurrency defects such as deadlocks, and unhandled exceptions like Null-PointerExceptions and Assertion Errors [2]. JPF-core is the basis for all JPF projects. It contains the underlying custom JVM and model checking infrastructure. As the number of threads increases, the output of the JPF which is in the textual format increases. The analysis of a defect using such a large textual data is time-consuming and inefficient. The solution to this problem is an use of Graphical user interface (GUI) such as jpf-visual which provides a basis for the development of GUI’s for JPF.

The jpf-visual tool is built upon JPF-shell by creating a new Error Trace panel which visualizes JPF program trace. It helps to understand the nature of failures and properties of the program. The jpf-visual tool
uses JPF-core to get the error traces and JPF-shell for visualization of that trace in the newly created panel Error Trace.

The work flow of the existing jpf-visual tool is given below:

- The jpf-core module loads jpf-shell, which is the main interface for the jpf-visual tool. Then the trace generation of the whole events is started by jpf-core.

- The Reporter in JPF, which is the data collector, uses these events and gives the error path. Once the error path is generated, a new Error Trace panel is created on the rightmost side of the JPF-shell for visualization of the trace, which is shown in the Figure 2.1.

- The method postCommand created in the jpf-visual tool, processes all the trace data and draws all the panels using JGraphX library. JGraphX displays graphs using Swing framework components.

2.3.2 Panels in jpf-visual

The jpf-visual tool consists of following panels:

1. **Main Panel**: It is the central-most panel of the tool which shows the transition information of the error trace.

2. **Navigation panel**: The navigation panel is on the left side, which has Collapse all and Expand all buttons. It allows the user to expand/collapse all transitions in the main panel. In addition, it has checkboxes for wait/notify, thread start/join, and, (un)lock which highlight corresponding parts of the transition. A drop-down list for field access and method calls allows to highlight the transition related to certain field access and method call in the main panel.

3. **World Map**: The panel on the right can show two views: an outline of the main tabular panel and a thread state view.
Figure 2.1: An overview of jpf-visual

4. **Thread state view** shown in Figure 2.1 is in the tabular layout with different colored bars, which indicate a thread state change in the error trace. A green bar means either the new thread is started or a lock is released. The bar in yellow color indicates that state changes to a lock. A red colored bar, which is not shown in the example indicates that the thread state changes to wait. It also provides an option of clicking on the transition range in the tabular layout which will bring the same transition range in the main panel.

A comprehensive JPF error trace with concurrent event information and visualization of these events with the above panels in the jpf-visual tool, helps to analyze concurrency related defects. However, JPF is slower as it does more than a normal bytecode interpreter and it might not support all the Java libraries such as java.io and java.net used by the system under test [2]. These limitations restrict the use of JPF for larger program verification. Hence, it will be helpful to extract such important concurrent event information using AspectJ instrumentation for regular JVM.

In a concurrent program, some errors occur under specific thread schedules. The lack of repeatability of such thread schedules and the non-deterministic nature of thread scheduler makes it difficult to analyze concurrent errors. Since JPF is built on the top of modified JVM, it provides a partial order to the events in the JPF execution trace, and the jpf-visual tool is able to visualize this event ordering. The ordering of events in the regular JVM execution trace can
be obtained using the happens-before relation, which is described in the next section.

### 2.4 Event ordering

In a non-deterministic concurrent program, program code may be re-ordered as a part of code optimization. The output of a program depends on the order in which events occur. However, this is difficult in the context of concurrency in which multiple threads may access shared memory simultaneously. The result of an unsynchronized program is unpredictable, in general.

#### 2.4.1 The Java Memory Model

The Java Memory Model (JMM) [23] sets boundaries on how the code is allowed to be transformed and describes which re-orderings are possible. In multi-threaded programs, the JMM describes how threads may interact through memory and what behaviors are legal. The JMM provides ordering guarantees with the use of volatile and synchronization. The change made by synchronized block is visible to other synchronized blocks with the same lock [24].

The JMM also defines a partial ordering of actions of the program which is called happens-before relation. This is used so that a write action performed by one thread is visible to another action in a different thread. Hence, providing partial order to the runtime events allows analysis of concurrent issues, which may occur when multiple threads operate simultaneously on the shared objects or resources. Happens-before between two actions has a very specific meaning in the JMM such that it is not necessarily imply that they have to occur in that specific order in an implementation [25].

#### 2.4.2 Happens-before relation

In concurrent programming, it is not possible to order the events based on the local time. Lamport [3] mentioned that the order in which events happen is more important than the absolute time of the events. Happens-before relation assures that action of the particular thread is visible to the other thread by giving partial ordering to the events in the program. Lamport [3] proposed the happens-before relation, which captures dependencies between the events as a partial order and helps to give the assurance of ordering of read and write to memory. If a write operation happens-before the read operation then results of
a write by one thread are guaranteed to be visible to a read by another thread [25]. This guarantee of visibility helps to avoid memory consistency error. Bijlsma [4], described that, "When two actions are in a happens-before relation, x happens-before y, the effects of action x are observable by action y."

Several thread actions such as lock/unlock, thread start, and thread join helps to create a happens-before relation. Rules to establish the happens-before relation mentioned by [26][4] are given below:

1. If x and y are two events in the same thread and x precedes y in program order, then x happens-before y. This rule ensures, each event in a single thread happens-before every other event in that thread, that comes later in the program order.

2. The volatile keyword is used for the shared variables among multiple threads. Accessing a volatile variable establishes a happens-before relationship between program statements. It ensures that a write operation to the volatile variable happens-before a subsequent read of the same variable in the other threads. This is used to avoid memory inconsistency errors.

![Volatile variable rule](image)

Figure 2.2: Volatile variable rule

This rule suggests that, when the read and write operation on the volatile
variable is performed by threads, it is possible to visualize these changes in data consistently. The Figure 2.2 [27] shows that write operation on volatile variable var statement by thread A will happens-before the read operation on that variable of thread B. It ensure that thread B can access the updated value of the shared variable among them. The arrows in the figures indicate the effects of actions of the thread visible to action in remaining threads.

3. An unlock() event on object happens-before all consequent lock events on that object by any thread. Figure 2.3 [27] shows that thread A and B use the monitor on same object lock. Hence, the happens-before relation ensures, the monitor on lock object in Thread A is only used when this lock object is released by thread B and vice-versa.

4. A call to Thread.start() happens before all the events in that started thread is as shown in Figure 2.4 [27].
5. All the events of a terminated thread are happens-before any other thread which successfully returns from a join on that thread is as shown in Figure 2.5 [27].

6. The happens-before relation is transitive.
Visualization of the happens-before relation shows actions performed by one thread to be visible to another action in a different thread. A change in the order of event execution may lead to the occurrence of data consistency errors and data race issues. Therefore, visualization of the happens-before relation helps in analysis of concurrent issues. The visualization of the happens-before relation in the VA4JVM tool is described in detail further in Section 6.7.

2.4.3 Event capture

The important event information is captured using instrumentation. However, capturing instruction-level events with instrumentation inserts extra instructions. Hence, this event capture is never atomic. This also applies while using locks as there could be a context switch, just before the lock is acquired. This can result in a scenario where the recorded events are not in the order of their occurrence. Higher-level cases of the happens-before relation such as thread.start, and thread.join, are exceptions to this case. However, this problem is common for any trace-based analysis and not limited to the VA4JVM tool. This thesis work focuses on visualizing the data that is present, without trying to modify or restrict the execution environment to allow a more precise event capture.

2.5 Summary

Faulty concurrent programs may exhibit failures that depend on the execution schedule. Different instrumentation tools can capture information about concurrent events. An approach of visualization of JPF trace analysis with the jpf-visual tool shown its usefulness for concurrency related defect analysis. Finally, discusses the happens-before relation rule, which provides partial ordering to concurrent events.
Chapter 3

Related Work

This section gives an overview of the different existing visualization tools for dynamic analysis approaches. It further explains different approaches to facilitate the analysis and understanding of concurrent executions. Finally, it describes why an approach of extension to the jpf-visual tool is selected for the visualization of JVM trace along with its useful features that will help to analyze the concurrent event.

3.1 JBInsTrace

Analysis of program execution trace reveals important information, which helps users to understand the program and its behavior. Caserta and Zendra [28] proposed JBInsTrace analysis tool that instruments Java bytecode to provide a fine-grained trace of Java software execution along with static information of source code. The tool does this in two stages: Firstly, runtime control flow is extracted and static information of the program is saved. Further, offline analysis of the execution trace is performed with the help of saved static information to visualize the call graph. JBInsTrace instruments program classes as well as JRE classes to get the full coverage of the runtime program. They use ASM instrumentation tool for analysis which adds extra bytecode and parses Java classes, with minimal memory requirements. JBInsTrace traces three kinds of events: basic block execution, start and end of a method. When classes are loaded, JBInsTrace parses them and performs three operations

• Unique identifiers will be assigned to each class, method and basic block with the help of recorded event integer.
• Static information is extracted from the source program.

• Performed instrumentation of the class by manipulating bytecodes. The instrumentation is done based on event number such as 01 for method start, 10 for method end, and 11 for basic block start.

The information obtained from instrumentation is further used for the analysis with the help of unique identifiers and static information. The tool also captures the thread number of events (method start/end, and basic block start). Tracer uses a singleton [29] class (which will have only single instance) to buffer the list of observed events (integers) before writing them to the disk. JBInsTrace trace analyzer is built on this information by relating each event of the trace with the static information saved. However, their result showed that a tracer has a reasonable performance penalty. Instrumentation of the event such as thread start/end and thread-based visualization is not taken into consideration in this tool. In addition, JBInsTrace is mainly a research-oriented tool, not used in software industries due to its scalability limits.

3.2 Zipkin

Zipkin [30] is a distributed tracing system which provides troubleshooting latency problems in microservice architectures. It does so by managing the collection and lookup of gather timing data. It filters or sorts all traces based on the application, length of trace, annotation, or timestamp. The percentage of the total trace time each span takes, allows identification of the problem application. Tracing information is collected on each host using the instrumented libraries. The library is instrumented by creating the following elements [30]:

• Core data structures - the information that is collected and sent to Zipkin

• Trace identifiers - the tag information are needed so it can be reassembled in a logical order by Zipkin. It consists of:
  
  – Generating identifiers: It generate IDs and selects which of these should be inherited.
  
  – Communicating trace information: Zipkin receives this additional information with the traces and their respective IDs.

• Timestamps and duration - records information on the timing of an operation.
Zipkin mainly focuses on troubleshooting of latency problems. Other issues of the application is not taken into consideration.

### 3.3 JThreadSpy

Malnati, Cuva, and Barberis \[31\] proposed JThreadSpy, an educational tool which aims to provide a detailed level execution trace of each thread of dynamically instrumented Java programs. JThreadSpy uses the ASM framework for bytecode instrumentation. The graphical analysis of execution flow makes it easier to understand the scheduler's non-determinism nature, effects of synchronization, and other concurrent issues.

JThreadSpy has some limitations, an increase in the number of objects leads to increase in the diagram complexity, which makes an analysis of whole representation difficult. Whereas, the jpf-visual tool filtering option helps in the analysis even with a number of object increases. The JThreadSpy tool is not applicable to the reverse engineering of a large system. The disturbances on the time measurement are introduced due to the incorporation of objects to record trace events. Addition of the code as instrumentation may result in different thread scheduling, this also changes the temporal relation between events. Moreover, the program analysis is made difficult due to the relation between JThreadSpy and the non-instrumentation of core Java classes. These limitations restrict the use of JThreadSpy for larger program analysis.

### 3.4 EXecution TRAce VIStualizer (Extravis)

Cornelissen et al. \[32\] proposed two trace visualization techniques based on the massive sequence and circular bundle view (modification of circular charts). These techniques are implemented in a tool called Extravis which helps to navigate successful through dynamic data without getting lost. The circular bundle reveals information about classes, packages and the dynamic calling relationships of the system. The massive sequence view provides an interactive overview of an entire trace.

The application of this tool are: traces exploration, feature detection, and top-down analysis to check the suitability of approaches. The current approach has not investigated the role of threads in the visualization tool. The technique that effectively displays both the threads and their interactions is not taken into consideration in the current Extravis tool.
3.5 SDExplorer

Reverse-engineered sequence diagrams help for debugging real-world execution traces by capturing runtime bindings and thread interactions. However, the main challenge is size explosion as the real world software execution traces results in too large sequence diagram which is hard to read. Lyu, Noda, and Kobayashi [33] proposed generic toolkit SDExplorer which effectively explores an extensive sequence diagram in a highly scalable manner. In addition, it supports the search, filter and grouping features of existing tools. They found that the resulting diagram is still large although the compression ratio is very high. Hence, the features for smooth exploration provided by SDExplorer are necessary. Thread-based visualization of the events in the program is given in the extensive sequence diagram of SDExplorer.

3.6 Java VisualVM (jvisualvm)

Java VisualVM [34] is GUI that provides monitoring, troubleshooting, and profiling utilities for Java application while running on JVM. It helps developers to improve the performance of the application along with troubleshooting applications and to monitor. The jvisualvm analyzes heap dumps, tracks down memory leaks and performs lightweight memory and CPU profiling. In addition, it monitors an important aspect of JVM runtime garbage collection (GC), as poorly performing GC can have adverse impacts on the performance of your Java application. Analysis of the concurrent issue is not taken into account by jvisualvm.

3.7 ConTest

ConTest [5], a tool used by IBM developers helps to find the bugs caused by concurrency. Copty and Ur [5] investigated the suitability of the Aspect-Oriented Programming (AOP) for ConTest testing tool in order to get entire runtime features of the tool. It has different coverage models such as synchronization coverage, shared variable coverage, method and branch coverage which helps to capture the concurrent events through instrumentation. Copty and Ur [5] noticed that AspectJ instrumentation is easy to use and suitable for ConTest. However, during research, they found that basic block coverage and identifying synchronization block, were the two missing features for complete implementation. They suggest that AspectJ can be very important
for implementing testing tool in the domain of data race detection, coverage analysis, trace analysis and many more.

However, the above-mentioned missing feature in the traditional AspectJ did not allow programmers to intercept events that capture the acquisition and release of locks caused by synchronized block. Therefore, it was unable to capture concurrency related errors fully. Hence, the programmer has to depend on the low-level bytecode instrumentation for that. Bodden and Havelund [18] proposed lock() and unlock() pointcuts by extending AspectJ. They implemented the lock() pointcut by matching object which is currently locked by another thread and the unlock() pointcut by matching on unlocking of locked object. It enables the AspectJ instrumentation tool available to use for analysis of concurrency related issues (data race). In addition, the – ShowWeaveInfo option in the AspectJ helps to issue informational messages whenever the weaver touches a class file [35]. These new features allow AspectJ to be used for concurrent program analysis.

3.8 Summary

In this chapter, a literature study on the different ways to obtain the execution trace for analysis using the instrumentation tool, is carried out. Further, various visualization tool for the execution trace along with its advantages and limitations are discussed. The above-mentioned visualization tools for the execution trace are helpful for getting an understanding of the concurrent program. However, every tool has some limitations. Hence the present thesis proposes a generic approach which may help the developer to understand and analyze issues in a concurrent program, through visualization.
Chapter 4

Event capture

Present chapter covers the use of AspectJ instrumentation tool for capturing the runtime events. It starts with method access, field access, thread start/end, wait/notify, ReentrantLock lock/unlock and synchronized method, followed by capturing synchronized block using lock()/unlock() pointcuts, proposed by Bodden and Havelund [18]. Further, covers different approaches of starting the VA4JVM for visualization at the end of main method along with an explanation of why a shutdown hook is not applicable for this tool. Finally, it discusses the approach for instrumentation of a standard Java library using bootstrap classpath option.

All the relevant runtime events need to be captured for successful integration of an execution trace information into the jpf-visual tool. AspectJ and its extended implementation of lock(), unlock() pointcuts [18] are able to capture the runtime events information required in the jpf-visual tool. Pointcuts and its advice type give information about location of the instrumentation, and the advice body determines what will actually be instrumented.

Pointcut information is written in AspectJ file HelloAspect and RuntimeData is the class which has data structures required for the visualization. Scope of the AspectJ pointcut is defined below in the listing 4.1 pointcut:

```java
pointcut instrumentationScope() : !within(HelloAspect) 
   && !within(RuntimeData) && 
   !cflow(within(HelloAspect));
```

Listing 4.1: Instrumentation scope pointcut

All runtime events with the AspectJ instrumentation is captured in the following way:
4.1 Method access

Instrumentation at the entry and exit of different method is important to analyze a concurrent program. Method calls with specific signature are captured using `call(Signature)` or `execution(Signature)` pointcut and syntax shown in listing 4.2.

```
pointcut <pointcut name>({any values to be picked up}) :
call({optional modifier} {return type}
    {class}.<method>({parameter types}));
```

Listing 4.2: Syntax for method call

A Signature can include wildcard characters which are used to select a range of join points on different classes and methods. This pointcut is also able to capture various interesting information at the time of method call as given below:

- The `call(Signature)` pointcut along with the `args([Type-Patterns | Identifiers])` pointcut captures the parameter values passed to the method call.

- The `call(Signature)` pointcut along with `target(Type | Identifier)` pointcut captures target object for the method call.

- The `execution(Signature)` pointcut along with `this(Type | Identifier)` pointcut captures the value of this Reference when a method is executing.

Various wildcard options within method signature along with the `args([Type-Patterns | Identifiers])` and `target(Type | Identifier)` pointcuts are given in the Table 4.1 [36].
**Table 4.1: Poincut signature with wildcard options**

<table>
<thead>
<tr>
<th>Signature with wild-cards</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execution(* *(..))</code></td>
<td>Captures the join points on any method which has any modifier, return type, class, or method and any number of parameters.</td>
</tr>
<tr>
<td><code>execution(*.new(..))</code></td>
<td>Captures join points on a constructor regardless of the modifier, return type, or class.</td>
</tr>
<tr>
<td><code>call(Thread+.new(..))</code></td>
<td>Captures join points on new() method on the class Thread and its any subclass.</td>
</tr>
<tr>
<td><code>(call(* Thread+.start()) &amp;&amp; target(childThread))</code></td>
<td>Capture start() method of the child thread.</td>
</tr>
</tbody>
</table>

In AspectJ, the `call(Signature)` and `execution(Signature)` pointcuts can behave differently based on the static and dynamic type of the target method.

**Difference between call() and execution() pointcuts**

The use of `call()` and `execution()` pointcut depends on the place in which the code is woven. `Call()` pointcut is used when the caller can be weaved itself and an `execution()` pointcut is used when an executor can be weaved. In the `execution()` pointcut we can intercept the call themselves, irrelevant of its called location. The `execution()` pointcut is used when user wants to pick a join point during execution of actual piece of code. This is because a `call()` join point does not capture the calls to non-static methods and also those made with `super`, or this constructor. `Call()` pointcut is picked out within the caller and `execution()` pointcut is picked out where the corresponding method is actually defined. Selection of the `call()` or `execution()` pointcut depends on which parts of the source code is under the user's control while applying aspect weaving. Consider the following example, shown in Listing 4.3, for better understanding the difference between `call()` and `execution()` pointcuts.

```java
class Producer extends Thread {
    private Buffer buffer;
```
public Producer(Buffer b) {
    buffer = b;
    this.start();
}

class Buffer{
    protected int SIZE;
    protected int usedSlots = 0;
    int m =3;
    int n =4;
    public Buffer(int b) {
        SIZE = b;
        array = new Object[b];
    }
    public synchronized void put(Object x) {
        while (usedSlots == SIZE) {
            try {
                System.out.println("Data information"+addValue(m,n));
                wait();
            } catch (InterruptedException ex) {
            }
        }
    }
    int dataInfo(int m, int n) {
        return m+n;
    }
}

Listing 4.3: Producer example for the better understanding the difference between call() and execution() pointcuts.

Consider the following scenario for a better understanding.

1. Application code calls (uninstrumented) Java library function
   In AspectJ, a third-party jar and JRE/JDK methods are not modified with advice. If the JRE/JDK method or any third-party method is called from the user written code, the method itself cannot be weaved. This will
not be possible, unless woven JDK and third-party jar are created as a preparatory step and passed them on the path during load time weaving (LTW) or compile time weaving (CTW). Therefore, in this case, executor such as JRE/JDK or third-party methods are not possible to weave with AspectJ advice. Hence execution() pointcut does not capture the event at all. Whereas, a call() pointcut is used, if user’s wants advice invocation of the code in a third-party jar or JRE/JDK methods. This is because the calling code is under the users control.

As shown in the above code, wait() method is called from method java.lang.Object.wait(). However, is not possible to weave the method itself unless creating a woven java.lang.Object.wait() method i as preparatory step. Whereas, the caller buffer (which is an instance of the Buffer class) can be weaved as it is the user’s code from which the java.lang.Object.wait() method is called. Hence, the call() pointcut is used for capturing the wait() method.

2. Calls within the application

If user has control over the calling as well as the called code, then both pointcuts can be used. But, execution() pointcut is just woven in one place, i.e into the method itself. Whereas, call() pointcut is woven into many places where a method is called. The selection of execution() pointcut depends on whether the user has control over the called code and can weave aspects into it. In addition, the user will select execution() pointcut, if the user wants to pick a join point that runs while an actual piece of code runs. Whereas, if the user wants to pick join point that runs when a particular signature is called, use call() pointcut. As shown in the code in Listing 4.3, public synchronized void put(Object x) defined in the Buffer class, is written in our own code which can be weaved using AspectJ advice. Hence, execution() pointcut will give a better result which is shown in Table 4.2.

In a method pointcut, there can be situations where either of them is needed. Therefore, it is realistic to have both of them for such situations. However, this results in capturing duplicate events where two events are generated for one action. Among these two call-execute events for the same method in a row, the second captured event is redundant. In order to resolve this problem of duplicate events, there is a need to remember the current method call, at each captured event using call() pointcut. Each event captured using execution() pointcut, will be
checked with its last event if it is also captured by call() pointcut. In such a case, the second captured event will be ignored. Importantly, for anything that happens after the event is captured using call() pointcut, it is essential to forget the last call event again, to account for self-recursion. It ensures that all events in a chain of execute-execute-execute are not lost.

3. An uninstrumented library calls the application code
When an uninstrumented library calls the application code in a call-back, it can be woven using AspectJ advice. However, it is not possible to instrument the library itself using advice. Hence, an execution() pointcut will give the proper output in this situation and call() pointcut does not capture the event at all. As program described in 4.3, println() is a method of java.io.PrintStream which is not under the user control. Whereas, this println() method, internally calls user-defined function addValue(m, n) which can be woven using advice. Therefore, the caller cannot be weaved in this case. An execution() pointcut gives the proper output in this case.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution(public synchronized* <em>,</em>(..))</td>
<td>As the synchronization method is written in the code and it can be accessed directly so that its instrumentation is possible. Hence execution() pointcut will provide the better result</td>
</tr>
<tr>
<td>call(* java.lang.Object.<em>,</em>(..))</td>
<td>It is Java library code which cannot be instrumented directly as it is outside the scope of the given class. Hence, a call() pointcut will provide the better result</td>
</tr>
</tbody>
</table>

Table 4.2: Difference between execution() and call() pointcuts

4.2 Constructor and initialization:
Information of new object creation can be obtained with the help of instrumentation in the constructor. A constructor call is captured using the execution(signature) pointcut along with the additional new keyword as part of the signature. The syntax is given in Listing 4.4.
pointcut getConstructor():execution(*.new(..)) ||
   initialization(*.new(..)) && instrumentationScope();
after(): getConstructor{}

Listing 4.4: Pointcut to capture constructor call

4.3 Field access

Concurrent program allows multiple threads to access the same variables where ordering between the events affects the output. Therefore, capturing of information during read and write operations to the variable by different threads can be helpful. The get(Signature) or set(Signature) pointcuts triggers the advice where the attribute is directly accessed and assigned. This pointcut captures both static and non-static accessed members as long as they are not declared as final shown in Listing 4.5.

pointcut getObject() : (get(* *.*) || set(* *.*)) &&
   cflow(traceMethod()) && instrumentationScope();
after(): getObject{}

Listing 4.5: Pointcut to capture field access

4.4 Thread start and join

Threads are important in a concurrent program because they allow running multiple tasks simultaneously. A thread is an independent unit of execution and defines a separate path of execution with its own program counter and stack frame. In Java, a multithreaded program is built upon the Thread class, its methods, and its associated interface, Runnable. The program will create a new thread either by extending thread class or implementing the Runnable interface. Implementing the Runnable interface is preferred than extending thread class since, using Runnable interface, it is possible to use inherit from another class in Java, which is not the case with extending thread class. The start() method starts a new thread by creating a separate call stack for the thread. Joining a thread with the join() method blocks the calling thread until the referenced thread terminates. Hence, it is useful to capture these thread-related events using the call(Signature) pointcut on the Thread.start() and Thread.join() methods shown in Listing 4.6.
4.5 Wait/notify

The `wait()` method suspends the thread on which it is invoked. In order to start the suspended thread and continue its execution, another thread has to invoke the `notify()` or `notifyAll()` method. These are used to coordinate activities of multiple threads in Java to reduce data consistency issues. Enclosed `wait()` in a while loop ensures that the thread is woken up with the satisfied condition and avoid spurious wake ups [38]. The `wait()`, `notify()`, and `notifyAll()` methods are captured by using `call(Signature)` `pointcut` on the Java library `java.lang.Object` for object and `java.lang.Thread` for a thread shown in Listing 4.7.

```
pointcut waitNotifyPoint(): call(* Thread+.wait()) ||
call(* Thread+.notify())|| call(*
Thread+.notifyAll()) || call(*
java.lang.Object.**(..))) && instrumentationScope();
after(): waitNotifyPoint()
```

Listing 4.7: Pointcut to capture wait/notify/notifyAll methods

4.6 Lock/Unlock

In a multithreaded program, when multiple threads try to operate on a Java object simultaneously, a data inconsistency issue may arise. To overcome this problem, a synchronized method or a block or, a ReentrantLock are used which allow only one thread to execute that method or a block on given object. Simple errors in the synchronization and lock acquisition can lead to data races which are important concurrency error and extremely hard to detect.
4.6.1 Synchronized method and block

Internally, the synchronization concept is implemented using the existing lock. A thread enters into a synchronized method or a block when it acquires a lock on the particular object and releases it while exiting. Acquisition and release of lock events in a synchronized method and block captured using AspectJ is given below:

**Synchronized method**

synchronized method is having a `synchronized` modifier which is invoked after successfully acquiring a lock on a particular object. It is captured using a pointcut that matches on the synchronized method modifier and `target(Type|Identifier)` pointcut. The target pointcut employed to capture the object on which a method is invoked which is shown in Listing 4.8.

```java
private static final Object lock = new Object();
pointcut syncPoint(): execution(synchronized* .*{..})
    && instrumentationScope();
after(Object lock): syncPoint() && target(lock){}
```

Listing 4.8: Pointcut to capture synchronized method access

**Synchronized block**

Bodden and Havelund [18] proposed a language extension to the AspectJ by providing `lock()` and `unlock()` pointcut. These pointcuts allow monitoring the events where the lock is acquired and released in a synchronized block. To use these newly implemented AspectJ pointcuts, a non-standard compiler option has to be set to `-Xjoinpoints:synchronization` in Eclipse and Maven AspectJ plugin configuration. The `lock()` and `unlock()` pointcuts are used along with `args([TypePatterns|Identifiers])` pointcut which bind the required identifiers to extract lock/unlock object. The pointcut to capture before lock acquisition and after lock release on object in the synchronized block, is shown in listing 4.9.

```java
before(Object l): (lock()) && args(l) && instrumentationScope(){}
after(Object l): (unlock()) && args(l) && instrumentationScope(){}
```

Listing 4.9: Pointcut to capture synchronized block
4.6.2 ReentrantLock

ReentrantLock class implements the Lock interface and provides same functionality as a synchronized keyword along with extended capabilities such as to check whether the object is locked or not. It provides the tryLock() method which acquires the lock, only if it is available or not held by any other thread and asymmetric lock/unlock nesting. The ReentrantLock lock() and unlock() methods are converted into lock/unlock events. Pointcut along with target(Type|Identifier) pointcut are used to capture the lock/unlock object using ReentrantLock class is shown below in Listing 4.10.

```java
before(ReentrantLock l): call (* ReentrantLock.lock ()) && target(l ) && instrumentationScope()

after(ReentrantLock l): call (* ReentrantLock.unlock ()) && target(l ) && instrumentationScope()
```

Listing 4.10: Pointcut to capture ReentrantLock lock/unlock events

4.7 End of main method

jpf-visual uses the method postCommand which processes the error trace data and draws all the panels with the help of jpf-shell. However, this approach does not work for VA4JVM as, it is independent of JPF and jpf-shell. In a multi-threaded program, the main thread is the first thread which begins the program execution and creates a different child thread. It is often the last thread to finish the execution if it performs shutdown action. Hence, in order to start visualization in VA4JVM, end of the main method is used. This is achieved by instrumentation of the end of main() method using after() advice, given below in Listing 4.11.

```java
pointcut mainMethod() : execution(public static void main(String[]));
  after() : mainMethod() {}
```

Listing 4.11: Pointcut to capture end of the main method

If the shutdown action is not performed in the main method, we cannot guarantee that the main thread is the last thread to finish. A main thread may exit before any threads it has created. In order to get a proper visualization, the main method needs to be the last one to exit. If the main thread will wait for all the child-threads to finish, it will be the last thread. In order to achieve this,
the following approaches are used.

### 4.7.1 Thread.join() method

To ensure that, the visualization will not be started too early, it is necessary to wait on all threads that were created by an application. This is achieved by maintaining a list of threads and wait on all of them at the end unless a third-party library creates threads. If the third-party library creates a thread, it is required to instrument third-party library to capture that thread. Instrumentation of the third party library is mentioned in the Section 4.8.

`Thread.join()` allows a thread to wait until the other thread has completed its execution. The thread list is created and a started thread is added to it when a new thread is captured by the `Thread.start()` instrumentation. Once the complete thread list is obtained, applying `Thread.join()` to each thread in the thread list allows the main thread to wait until all its child threads finish the execution. It is important to use a try/catch block for `Thread.join()` method to capture the `InterruptedException`, since every join() method interrupted an exception and `InterruptedException` is checked exception. This exception is either handled by using a try-catch block or by using a keyword throw, otherwise a compile time exception will occur.

### 4.7.2 ShutdownHook

A shutdown hook is a non-started thread which executes when the JVM is shutting down. It is used for the graceful termination of all the running threads. A shutdown hook is another way which ensures that the main thread is the last thread to terminate. In addition, it is used in case of user termination request (Ctrl+C), standard kill a SIGTERM without kill-9 or kill-SIGKILL and when operating system is shutting down. However, JavaDoc mentions following limitations of the shutdown hook:

- The shutdown hook stops the Abstract Window Toolkit (AWT) event dispatch thread which doesn’t allow further GUI interaction.
- It finishes the operation quickly.
- The use of shutdown hook in case of AWT event dispatch thread which is thread-based service may lead to deadlocks.
CHAPTER 4. EVENT CAPTURE

However, the jpf-visual GUI runs on top of AWT and its GUI may take lot of time to finish. Hence, the above-mentioned limitation makes it impossible to use a shutdown hook in case of user termination request (Ctrl+C). A shutdown hook would work nicely if the events were processed and visualized in an external process, as it could be then possible to just notify the external process that no more events will arrive. However, in the current version of the jpf-visual tool, visualization is not an external process. Therefore, to use a shutdown hook, it would require a different architecture of the jpf-visual. Also, there is the possibility that the application itself needs its own shutdown handler, where this option induces problems.

4.8 Java standard library instrumentation

The standard Java library has to be instrumented to track threads including internal threads. AspectJ is unable to instrument the standard Java library directly using pointcuts, because, it requires AspectJ library to be loaded in the JVM, and not available when Java standard library is loaded. Hence, build time weaving of the JRE classes will be helpful in this aspect. The Java standard library is defined in rt.jar and, using AspectJ compiler it is possible to weave this rt.jar. rt.jar is weaved by instrumenting thread start method. The below command used to weave the rt.jar into newrt.jar with the help of org.aspectj/aspectjtools:

```
"$JAVA_HOME/bin/java"
-classpath "<path to aspectjtools.jar>:<path to aspectjrt.jar>:$JAVA_HOME/lib/tools.jar:$CLASSPATH"
-Xms256M -Xmx2048M org.aspectj.tools.ajc.Main -8
-inpath $JAVA_HOME/jre/lib/rt.jar
[<AspectJ code path>...] -outjar <path to newrt.jar>
```

Listing 4.12: Command to instrument rt.jar into newrt.jar

This command weaves the JRE jars and puts the weaved classes into a newrt.jar file. Using the -Xbootclasspath/p option preloads this weaved class(newrt.jar) before the regular run-time library. This option helps to prepend only the modified Thread class and omits the remaining unmodified runtime library. The weaved JRE classes can be used by passing the following VM argument in the run configuration:

```
-verbose:class -Xbootclasspath/p:path_to/newrt.jar
```


Eclipse launch configuration:

$ -verbose:class
-Xbootclasspath/p:${resource_loc:/project_name/newrt.jar}.

Running the program with this VM arguments may give an error regarding the failure of VM initialization. The error may be solved by increasing NewSize using the following command:

$ java -XX:NewSize=1 -XX:MaxNewSize=1023k -version

However, the same approach cannot be used further for the instrumentation of the standard Java library due to a change in Java features for JDK enhancement. The bootstrap class-path option is removed in the JPE 261 [43] and, therefore, a user will be unable to prepend the weaved class with this option.

### 4.9 Summary

The chapter describes how the concurrent events information such as field access, thread start/end, method access are obtained using AspectJ. The extended AspectJ helps to capture lock/unlock pointcut. Furthermore, the chapter mentioned that, why the shutdown hook approach is not applicable in this scenario. Finally, it mentioned an approach of standard java library instrumentation.
Chapter 5

Event processing

The jpf-visual is built on top of JPF error traces, and it uses its error path for visualization. In order to use the same tool for the JVM error traces, it is necessary to create a similar error path structure from JVM trace. Hence, it is good to understand the data structure of the error path in the jpf-visual.

This chapter starts with a description of the adapter approach to generate event data structures used in the jpf-visual tool. In addition, it introduces the artificial creation of a ChoiceGenerator and instruction data structure from captured events with the help of global variables. Further, the chapter mentions the creation of transition information at the time of a thread change and when an important thread actions mentioned in the ChoiceGenerator are captured. Finally, it describes the creation of the error path object with the help of this transition information. The error path object is then passed to the VA4JVM tool to highlight the important events using JFrame.

5.1 Structure of the path instance

The error trace path is an important data structure of the jpf-visual tool. This path object consists of application name and stack of transitions. Each transition is collections of Steps, ThreadInfo and ChoiceGenerator information, which is obtained in the following way:

- **Step**: Each Step contains information of a different type of instruction. Such an Instruction provides information about the class, field, and method name of the particular captured event.

- **ThreadInfo**: A ThreadInfo instance record thread related information of the current running thread on which particular event occurs.
• **ChoiceGenerator**: A ChoiceGenerator consists of information about the particular state of the thread (e.g., ROOT, LOCK, WAIT, START, JOIN, RELEASE, TERMINATE) along with its thread Id. Detailed information is given in Section 5.2.4.

An overview of the error path instance is shown in Figure 5.1. It shows that Step instance is created from the right type of Instruction data structure. Further, this Step object along with ThreadInfo and ChoiceGenerator data structures is used to generate transition information. Finally, transition information along with application name creates a path instance.

![Figure 5.1: An overview of error path instance of the jpf-visual](image)

The JVM event trace obtained using AspectJ needs to be processed further to convert it into JPF compatible trace. This is achieved by code refactoring and implementing common interfaces using adapter classes.

## 5.2 Code refactoring and adapter approach

JPF was originally developed by NASA, and jpf-visual is built on top of the JPF error traces. Therefore, the jpf-visual tool has NASA packages dependencies to process the JPF error trace path for visualization. To achieve the goal of successful integration of JVM traces into jpf-visual and make tool compatible with the JVM and JPF error traces, the jpf-visual tool code needs to be refactored. Code refactoring helps to separate JPF specific code and generic code which contains Java GUI elements. It bridges data with the GUI of the jpf-visual tool and creates a common interface. The interfaces are created in such a way that they are very close to the one used by jpf-visual, except that, there are no direct NASA package dependencies.
The interfaces are implemented using adapter classes, as adapters allow two incompatible interfaces to work together. It does that by allowing the interface of an existing class to be used as another interface. Adapters wrap the JPF classes effectively by implementing only those functions that are actually used by the jpf-visual. The adapters are created for only those classes whose objects are used while iterating over the error path instance as described in the Section 5.1. Implementation of the common interfaces for JPF and JVM adapters allows that only JPF adapters can use the NASA packages for the JPF specific implementation. In addition, NASA packages are not compiled against VA4JVM. The JVM adapters help to convert the JVM trace into a data structure such that it mirrors the data structure used in the jpf-visual. These data structures are created by providing all required information to its adapter class.

5.2.1 Path adapter

path is an important data structure from which all other data structures information can be retrieved. The Path adapter creates the path object by providing application name and stack of transition information to its constructor. Furthermore, the adapter implements all essential function on path object which are used in the jpf-visual, in order to process error trace. As the path consists of a stack of transition information, it is essential to create the transition data structure by creating its adapter.

5.2.2 Transition adapter

The transition data structure is a collection of information such as the Choice-Generator, ThreadInfo, and Step which consist of instructions. The class adapter pattern for Transition interface is as shown in the Figure 5.2. The figure shows the extraction of transition information with the help of class adapter pattern from JPF and JVM execution traces. A TransitionJpfAdapter class calls all the JPF-specific transition functions with the help of jpfTransition object which is passed to its constructor. The actual implementation of these JPF specific functions is in the class gov.nasa.jpf.vm.Transition. Whereas, implementation of JVM specific transition functions is done in the TransitionJvmAdapter itself by passing required JVM specific information to its constructor. Hence, this adapter pattern for class shows that the Transition adapter interface has two implementations(JPF and JVM) and each implementation internally uses
a specific data type.

![UML diagram of class adapter design pattern for Transition](image)

Figure 5.2: UML diagram of class adapter design pattern for Transition

However, low-level informations of ChoiceGenerator and Instruction are not obtained directly from AspectJ. Whereas, these pieces of information are used to get the thread state view and identify different runtime concurrent events in the existing jpf-visual tool. Hence, this information needs to create artificially with the help of their respective adapter classes.

### 5.2.3 Creation of instructions of the right type

In the jpf-visual tool, error trace information is processed based on the event instruction of particular type in the transition. The instruction information cannot be directly captured in JVM trace, as AspectJ does not provide such a low-level data. Hence, the synthetic instruction records are created in such way that high-level information like methods, fields, class, etc., can be accessed.

Every time when the event of interest is captured by AspectJ advice, right type of instruction is created. Different pointcuts advice captures the
event information simultaneously, resulting in continuous addition of new information. In order to create the right type of instruction, it is important to have detailed event information. Singleton pattern helps to update the continuously captured information without getting lost. A single instance of the singleton pattern allows temporary storage of captured event information required for instruction creation. The instruction type depends on the type of event advice captured. Further, instruction objects are created by passing this required event information to the right type of Instruction adapters as shown below.

- An instruction of type VirtualInvocation is created when advice captures wait(), notify(), and notifyAll() method event.
- A JVMInvokeInstruction is created when advice captures an event for method call other than above-mentioned method.
- A FieldInstruction is created whenever advice captures an event where a field is getting accessed.
- A LockInstruction is created when advice captures the lock(), unlock(), and synchronized block events.
- A JVMReturnInstruction is created when advice captures the synchronized method event.

5.2.4 ChoiceGenerator

The jpf-visual tool uses the ChoiceGenerator for the Thread-State diagram. In the ChoiceGenerator, the information of thread id and type of action of that thread are required to highlight the different events on the threads. The synthetic ChoiceGenerator data structure is created by providing the information of thread id and type of action of thread to its constructor. In addition, all required functions on this data structure are implemented in its adapter class. Information about the type of action of the thread depends on the following captured event:

1. ROOT: For the first event in main (when main is started).
2. START: When Thread.start is called.
3. JOIN: When Thread.join is called.
4. TERMINATE: When a thread terminates (when its main() or run() method finishes.

5. WAIT: Object.wait()

6. LOCK: monitorenter (when a lock is acquired by thread)

7. RELEASE: monitorexit (when a lock is released by thread)

8. Running: Created this type of action for the rest of the events.

The ChoiceGenerator information is created when an above-mentioned event is captured by thread. These captured actions of the thread in ChoiceGenerator also helps to define and visualize happens-before relation in the VA4JVM, which will be discussed further in Section 6.7. The main goal of creating this synthetic instruction and ChoiceGenerator is to eventually have one version of the jpf-visual that can handle both types of trace data. The ThreadInfo created by capturing thread related information when a particular event is captured by advice. All the required pieces of information are now available to create the transition object.

5.3 Thread transition

For a better understanding of multithreaded program, it is useful to get all events information based on the thread transition. Hence, the new transition information is created every time when the following situation occurs:

1. When a thread change occurs as detected by the updated thread id of each captured event.

2. When an important thread action mentioned in Section 5.2.4 is captured. Such transition information is created, since path instance is processed based on transitions in jpf-visual. This separate transition information helps to get a proper visualization of the thread state view with an existing tool. In addition, such important thread actions often define happens-before relations between events described in the Subsection 2.4.2.

Transition information is created with the help of thread information and ChoiceGenerator information. After the first transition, if one of the above-mentioned situation occurs, the previous transition information is added to the stack and new transition is created. The above-mentioned situation does not
add the last transition to the stack, which is added further, separately. Once, all the events in the application are processed, the added stack information along with the application name is used for creating the error path. This error path then passes to the jpf-visual tool the important event. In addition, it provides the action of a particular thread at a particular event and thread state view in JFrame. The pseudocode for event processing to create an error path from the right type of instruction, ChoiceGenerator, and threadInfo information is given in below Appendix A[A].

Once the path instance is created, it is passed to the VA4JVM tool. However, thread IDs are not identical across the platforms and may be large; the current table-based layout of jpf-visual tool relies on minimal, consecutive thread IDs. In order to properly visualize the captured events based on threads, its IDs are converted to small numbers during a pre-processing step. It is achieved by creating thread list of unique thread id and index of that thread list is passed to the existing code for visualization. These changes help for the successful visualization of the events for the particular thread in the VA4JVM.

For successful visualization of the event trace, the VA4JVM tool needs to compile only GUI specific file and exclude JPF dependent class of the jpf-visual during compilation. In addition, it is required to set the AspectJ configuration option to enable lock() unlock() pointcuts. The complete source code along with build and run script to start the VA4JVM tool is present in the GitHub repository[1].

### 5.4 Summary

The chapter shows that code refactoring, use of adapter classes and artificially created data (ChoiceGenerator and instruction) make the JVM trace JPF compatible. The path instance is further generated by processes through the instruction, transition and ChoiceGenerator data structures. The next chapter shows that how well the tool is able to visualize the captured events and the happens-before relation.

[1]https://github.com/monalip/VA4JVM.git
Chapter 6

Experiments and results

This chapter demonstrates the visualization of different concurrent programs in the VA4JVM tool to show how well the tool works. In addition, it provides the visualization of the happens-before relation for the concurrent event in the VA4JVM tool. The different concurrent programs source code are present in the GitHub repository[1]. The tool is tested against several multithread programs in order to check to what extent the VA4JVM tool is able to highlight the runtime features, happens-before relation, its limitations, and what still has to be done for efficient analysis. Finally, we discuss the runtime overhead of the program.

6.1 Fibonacci

The program finds the \(n\)th fibonacci number using two techniques: memoization or bottom up. This program is used here to test features such as field and method access of the VA4JVM. The existing jpf-visual tool highlights the access of different runtime features using a different color in the main panel for JPF trace. Figure 6.1 reveals that it is possible to highlight the method and field access for JVM trace as well. As a result, the execution trace in the main panel is highlighted with a pink color represents access of map field access. Similarly, fibMemo() and fibBotUp() methods access is represented by bluish green and orange color respectively.

[1] https://github.com/monalip/VA4JVM.git
6.2 Racer

Racer is the concurrent program that illustrates the execution of two threads. These two threads share a common variable and perform a certain operation on that variable. This program is used to test how the VA4JVM tool can be used to analyze the change in an output of the program depending on the sequence in which threads access the shared variable.

The figure highlights access to shared variable \( d \) represented by pink color by main thread and thread-1. In the trace \( \text{doSomething}(\text{int } n) \) is a method that is called by both threads. Calls to method \( \text{doSomething}(\text{int } n) \) is represented by bluish green. As figure 6.2 shows that call-
ing dosomething(int n) method by thread-1 results in changing the value of d to zero. Further, if this changed shared variable is used by the main thread to perform a division operation, this leads to java.lang.ArithmeticException: / by zero exception in the main thread. Hence, from Figure 6.2 it is revealed that the VA4JVM tool can help developers to analyze the cause of exception the main thread.

6.3 Readers-writers problem

The reader-writer concurrent program is about accessing a shared variable among the reader and writer thread with certain restrictions [45] as follow:

- If a writer thread is ready, then at that time, only the writer thread can write to the file.
- No other thread can read when a writer thread is writing.
- If a thread is reading no other thread can write.
- Multiple threads can read at a time.

This program is selected to test, how the VA4JVM tool and its filtering options help to analyze these conditions.

As shown in Figure 6.3 writer thread-2 performing write() operation with following three steps:

1. **Before write:**
   The beforeWrite() method shown in sky-blue color in writer thread-2 checking if writer thread-2 is allowed to write or not, using allowWriter() method in corn color. As shown in the Figure 6.3 the reading thread-1 is still active, hence writer thread-2 is in waiting state shown with wait() method in light pink color till reader thread is finished reading. Once the reader thread is done with reading represented by afterRead() method in the olive color, it will notify writer thread. It is shown by notifyAll() method in light pink color in the thread-1 which wakes up the writer thread-1. Again, when writer thread check for the allowWriter() method and this time it does not find any reader thread-1 active, so it will proceed further.

2. **Perform writing:**
   As there is no active reader thread-1, writer thread-2 performs writing to file represented by doWrite() method in turquoise color. Till
this time, reader thread-1 is in waiting state shown by light pink color wait() method in reader thread-1 column.

3. **After write:** Once the writer thread-2 is done with writing, it uses afterWrite() function highlighted by light violet color to notify reader thread using notifyAll() represented in light pink color.

![Screenshot of the VA4JVM for Readers-writers problem](image)

Figure 6.3: Screenshot of the VA4JVM for Readers-writers problem

From the visualization of these events in the VA4JVM in Figure 6.3, it is revealed that the VA4JVM tool makes it possible for developers to analyze a particular condition in the program, to check if the program is behaving properly or not. The different filtering options help to focus on a particular method and block in such a complex concurrent program. Similarly, the thread state
view on the right-hand side of the Figure 6.3 also helps to see the different state of a thread. A red colored bar represents a thread that is in waiting state. In addition, the yellow bar represents the acquisition of a lock and a green colored bar represents the release of a lock. The short green line of the main thread shows that the main thread terminates before its child thread. Hence, this thread state diagram view of the VA4JVM is also helpful to track lock/unlock events. Hence, visualization of different runtime events, thread activities, and dynamic filtering options in the VA4JVM tool help software developers to analyze the program. In addition, it gives information that may be related to the failure.

6.4 Dining philosophers problem

The problem is about five philosophers, and there are five forks which are placed between each pair of adjacent philosophers. A philosopher may eat by picking up two adjacent chopsticks to him. One chopstick may be picked up by any one of its adjacent followers but not both. The problem is about the use of synchronization in order to design a solution so that the philosophers achieve their goal of eating and thinking without getting starved.

There exist two deadlock situations when all five philosophers are holding a left fork and waiting for a right fork and vice-versa. When two or more threads are waiting for each other forever, such a cyclic dependency may lead to a deadlock situation. The inappropriate use of the synchronization on multiple objects can lead to the possibility of a deadlock [24]. Hence, pieces of information such as, a list of the waiting threads on a given lock and the thread holding a lock, are useful for getting an understanding of the deadlock in a given program [46]. This Dining philosophers problem is selected for testing to check, to what extent the VA4JVM tool is able to demonstrate synchronization issues and challenges of avoiding deadlock.

A deadlock results in indefinite waiting for the lock and may hang the current application. However, the current version of the VA4JVM tool starts the visualization after the end of the main method. In addition, a shutdown hook is not appropriate for GUI which runs on top of AWT and the way to capture the signal for the termination of JVM is not well documented. Its functionality is not in the official API and it is also not portable [47][48]. Hence, in order to start the visualization of such a hanged application to understand the deadlock, there is a need to rewrite the program using the ReentrantLock. The ReentrantLock has the functionality which enables checking the deadlock condition. This helps to call the visualizer when the deadlock is detected in
order to analyze deadlock issue in the program.

**Deadlock analysis using ReentrantLock**

ReentrantLock has the functionality to check if another thread holds a lock. This feature is not available for conventional Java locks. A program is modified using ReentrantLock and the lock acquisition is made atomic by using global lock such that it involves an unsafe check and use of ReentrantLock. The deadlock detection function is written by considering the following points which may result in deadlock:

1. Every time before a lock is about to be obtained, check if there is at least one other thread that is in RUNNABLE state.

2. If there is no such thread, and the lock about to be taken is owned by another thread, then the current thread is the last thread. If this last thread tries to obtain a lock, this action will lead to a deadlock.

The thread list and the isLocked operation help to check the above-mentioned condition. There are two main aspects behind the rewriting the Dining philosophers example:

1. Make lock acquisition visible to the checker, which is achieved by checking the isLocked condition before acquiring the second lock.

2. Make the checker call and lock acquisition atomic. It is done by using the global lock around call to lock()/unlock() operation of ReentrantLock shown in 6.1.

```java
public void run() {
    while (true) {
        synchronized (globalLock) {
            rightFork.lock();
        }
        synchronized (globalLock) {
            while (leftFork.isLocked()) {
                try {globalLock.wait();}
                catch (InterruptedException e) {
                    e.printStackTrace();}
            }
            leftFork.lock();
```

synchronized (globalLock) {
    leftFork.unlock();
    globalLock.notifyAll();
} 
synchronized (globalLock) {
    rightFork.unlock();
    globalLock.notifyAll();
}

Listing 6.1: Dining philosophers problem using ReentrantLock

The complete modified program along with deadlock detection code is shown in Appendix Section B.1. In this example, the deadlock condition is checked in two places:

- after acquiring the first lock.
- after acquiring both the locks, since sometimes threads are not releasing the grabbed lock. This results in a deadlock situation as other threads are waiting for the lock forever.

This deadlock condition check may still miss some cases due to race conditions in the check and use pattern. In practice, though, this will catch many deadlocks, and it does not cause false positives, as if there is only one thread left, the deadlock would indeed be real.

The modified program uses a synchronized block on variable globalLock around lock/unlock event of forks. This synchronized block is represented by turquoise color `synchronized(globalLock)` shown in Figure 6.4. Figure 6.4 illustrates the deadlock condition, as all five threads acquired right fork represented by pink color `rightFork.lock()` are blocked. When thread-4 tries to acquire leftFork it check for the condition if the left fork is already acquired by other thread using `leftFork.isLocked()`. As in this case, thread-4 left fork is acquired by thread-3, hence it will check for the deadlock condition. All threads acquired rightFork and in order to get left fork all thread acquired a lock on `globalLock` variable. This results in deadlock situation because threads try to acquire the left fork which was already acquired by other threads. All threads will be waiting for the left
fork forever. It also highlights the synchronized block on globalLock to show the lock/unlock event on it.

Figure 6.4: Screenshot of the VA4JVM for dining philosophers problem

However, the lock() and unlock() pointcuts on the synchronized block is captured by the same synchronized block location string. As an existing tool uses the same color to represent the lock/unlock event, such a visualization may sometimes confuse the viewer. Whereas, thread state view diagram helps to overcome this confusion as the yellow bar in tabular layout representing lock event on a thread and a green bar representing lock is released. Combination of both visualization format helps to analyze the deadlock condition. This example shows that by providing the necessary infrastructure, the VA4JVM tool works in principle for deadlock analysis. This modified program may miss some deadlock situation, but deadlock indeed is real.

6.5 Server-client program

In a server-client program, two separate programs communicates with each other through network protocol. Consider one program as a server and the other as a client, in which the client connects to a server port, while the server
is listening to possible connecting clients through a socket. The socket is a class which belongs to package `java.net` and is used to establish communication between two machines. Whenever the socket is used to connect with server, it is important to provide the IP address of the server and the TCP port number to identify the particular process. In this server-client program, the client connects, sends messages to the server and the server shows them using socket connection.

This program is selected for testing to analyze the communication between client and server which is not possible in the `jpf-visual`. This is because when a server-client program runs on the JPF to get the execution trace, it gives the error `UnsatisfiedLinkError` which is shown in the Figure 6.5 with a pink colored box, since JPF do not support the Java libraries such as `java.net` described in the Section 2.3.2. Hence, the `jpf-visual` tool cannot provide the visualization of this program as it uses JPF error trace. Whereas, visualization of this program in the `VA4JVM` is shown in the Figures 6.6, 6.7.

![Screenshot of JPF error output for server-client program.](image)

Figure 6.5: Screenshot of JPF error output for server-client program.

The visualization of the `ServerSocket` program is shown in Figure 6.6, it highlights the code where server accepts the client connection shown in turquoise color. The code `br.readLine` highlight in pink color is used to read the message from the client. Finally, the code `server.close()` shown in an orange color represents that server closing the client connection. The visualization of the `ClientSocket` program is shown in the Figure 6.7, it highlights the code where the client can send the message to the server with
the help of OutstreamWriter and socket.getOutputStream is shown in orange color. This message is write to the OutstreamWriter highlight in pink color. Finally, flush the OutstreamWriter shown in turquoise color. Such server-client program communication in the VA4JVM ensure that analysis of communication through a network protocol is possible in the VA4JVM whereas, it is not possible in jpf-visual. However, as the standard libraries are not instrumented in AspectJ, some of the code such as the creation of the socket through its constructor is not getting captured in the JVM execution trace. Hence, such an event is not visualized in the VA4JVM for efficient analysis there is a need to instrument the standard library as well.

Figure 6.6: Screenshot of the VA4JVM for ServerSocket program.

Figure 6.7: Screenshot of the VA4JVM for ClientSocket program.

6.6 Producer-consumer problem

The producer-consumer problem is a classical example of a multi-process synchronization problem that shares a common buffer between the producer
and consumer thread. The producer generates data into the buffer, and the consumer removes it simultaneously. It is important to make sure that, the producer will not add data into buffer when it is full and the consumer will not remove data from the empty buffer. Hence, the producer and consumer threads need to communicate with each other, so that they know when it is safe to attempt to generate data and remove it from the buffer. They use a counter variable usedSlots whose contents change reflect the amount of free buffer slot. As both the threads access shared data concurrently, an inadequate use of synchronized method and monitors could result in data race and another concurrent issue during execution.

The producer-consumer problem is selected for testing, to analyze communication of co-operating process and track the monitors for concurrent issue analysis. The program uses put(Object x) and get() methods for adding and removing data from buffers respectively. In order to make this communication thread-safe program uses a synchronized method for adding and removing data, which helps to monitor the shared buffer access. The program used four producer and five consumer threads in order to test the tool for larger thread communication.
As shown in Figure 6.8, the thread-id 14 represented consumer thread. When it is attempting to remove data from the buffer, it first checks if the buffer is empty, shown by the sky blue color while loop condition. As there is no data in the buffer to remove, the consumer thread of thread-id 14 is waiting for data, represented by light pink color `wait()` method. When consumer is in a wait state, the producer thread of thread-id 10 put the data into a buffer and notify all consumer thread using `notifyAll()` method highlighted with the light pink color. In addition, it increments the shared counter variable `usedSlots`. After notification, the consumer thread of thread-id 14 wakes up and again check for the availability of buffer data. As there is a data in the buffer, consumer thread of thread-id 14 removes it and decrements the counter `usedSlots`. Once the consumer thread of thread-id 14 removed data it releases the acquired lock on the `get()` method represented by pink color `return` statement.

The results shows that the VA4JVM tool helps developers to track the communication between concurrent process by visualizing `wait()` and `notifyAll()` methods. In addition, highlighting of synchronized methods and
unlock events helps to monitor lock acquisition and release events in concurrent communication. Hence, this analysis can further help to identify and understand the issue such as data race. Thread state view on the right-hand side of the panel helps to see the different state of the threads shown in 6.8. Differently color bar of the threads in thread state view helps to get the overall idea of the program execution. In addition, by clicking on a particular event bar of thread will bring the view to the corresponding transition range in the main panel. This feature allows to analyze an important action of thread with just one click. Such visualization helps for program comprehension of an enormous amount of data that is collected at a runtime without getting lost. However, runtime concurrent processing of events and initializing all the GUI widgets adding some overhead to the tool VA4JVM for the larger threads. This overhead results in the VA4JVM tool taking time during highlighting different events using different filtering options.

Next section mentions visualization of happens-before relation in the VA4JVM.

### 6.7 Visualization of happens-before relation

The partial order reduction in JPF helps to analyze concurrency issues, as it makes possible to determine the order of the event execution. In addition, it helps to group the related transition for visualization in the jpf-visual. Such a type of visualization of happens-before relation allows action performed by one thread to be visible to another action in a different thread. In addition, it helps to analyze the occurrence of data consistency errors and data race issues efficiently. Hence, it is essential to provide the partial event ordering in regular JVM trace to visualize it in the VA4JVM. Happens-before relation provides such partial events ordering in regular JVM as described in the subsection 2.4.2.

### 6.7.1 Happens-before relation in the VA4JVM

A ChoiceGenerator captures START, JOIN, LOCK, UNLOCK, WAIT, and RELEASE actions of the thread which are related to the happens-before relation. These captured actions helps to establish the happens-before relation in the JVM execution trace. However, the information is slightly hidden in the VA4JVM. In the VA4JVM tool, instrumentation inserts extra instructions to capture the runtime events for visualization. Hence event capture itself is never atomic and instrumentation may cause extra thread switches. This
means the order in which the events are recorded is not the order in which their corresponding instructions execute. For some events, the happens-before relationship is guaranteed. Other instructions can be reordered during code optimization. Visualization of the happens-before relation in the VA4JVM based on the rule described in the Subsection 2.4.2 along with its limitations are described further.

- **Single thread**

  According to the single thread rule for happens-before relation, each action in a single thread happens-before every other action that comes later in the program order. Visualization of happens-before relation for the single thread program is described using Fibonacci program in the Section 6.1. The program uses a single thread to print Fibonacci series using statements is shown in the Listing 6.2.

  ```java
  public static void main(String[] args) throws Exception {
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      int n = Integer.parseInt(br.readLine());
      System.out.println(fibMemo(n));
      System.out.println(fibBotUp(n));
  }
  
  Listing 6.2: Fibonacci program
  
  The program shows that event of printing Fibonacci series using fibBotUp(n) method is the last event. Visualization of the program using the VA4JVM tool is shown in the Figure 6.9. The arrow indicates, the effects of all preceding event actions are visible to action in the thread that comes later in the program.

- **Volatile variable**
As the volatile keyword is used for the shared variables, establishing the happens-before relationship between them ensures, any memory writes to a volatile variable by one statement are visible to another statement.

Figure 6.10: Happens-before relation for volatile variable rule

The happens-before relation for the volatile variable in the VA4JVM is visualized using Racer problem described in the Section 6.2 and marking shared variable d as a volatile. Visualization of this program in the VA4JVM is shown in the Figure 6.10 which ensures that writing to the shared volatile variable by Thread-1 happens before reading operation of the main thread. Establishing happens-before relation for volatile variable sometimes used to maintain data consistency across threads and helps to find the exception in the main thread error. Unfortunately, this is not the case always in the multithreaded program. Since the VA4JVM tool has true concurrency where two threads generate events at the same time, it often ends up with a lot of thread switches in the trace. Hence, it may be possible that event, where a shared variable is used by the main thread is captured before the writing operation of the thread-1. For such a case, it is not possible to visualize happens-before relation for a volatile variable in the VA4JVM.

- Thread start
  The rule to establish happens-before relation for thread start is described in the Subsection 2.4.2. It ensures that the Thread.start() method happened before all the events in the started thread. Visualization of the producer-consumer problem for two thread in VA4JVM is shown in the Figure 6.11. It highlights the start() method of threads, run() methods, and the first statement in the run() method with different colors (see Table 6.1). The visualization of these methods ensures that it is possible
to establish the happens-before relation for the thread start rule using the captured START action of the thread in ChoiceGenerator.

![Happens-before relation for thread start rule](image)

**Figure 6.11: Happens-before relation for thread start rule**

<table>
<thead>
<tr>
<th>Source code</th>
<th>color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>this.start()</td>
<td>Green</td>
<td>Thread-1 started.</td>
</tr>
<tr>
<td>public void run()</td>
<td>Olive</td>
<td>Started run() method of Thread-1.</td>
</tr>
<tr>
<td>for() loop</td>
<td>Sky blue</td>
<td>First statement in the started Thread-1.</td>
</tr>
</tbody>
</table>

**Table 6.1: Events in the execution trace**

The first green-colored bar of threads in the thread-state diagram is shown in the tabular layout, which helps to identify the started thread. The pink-colored arrow indicates that thread start() method happened before the first statement in the run() method. Hence, the combination of the thread-state diagram and thread-based events view in the main panel helps to visualize the happens-before relation for thread start rule in the VA4JVM.

- **Thread join**
  The rule to establish happens-before relation for thread join is described in the Subsection 2.4.2. It ensures that all events in the started thread, finish before all subsequent actions in a thread where thread.join() method is called. Visualization of the Racer program in the VA4JVM is shown in Figure 6.12. It highlights the t.join() method in the main thread with green color, end of the run() method in turquoise color, and the
subsequent statement after the `join()` method in the pink color, which is shown in the Figure 6.12. The pink colored arrow indicates that end of the `run()` method in Thread-1 happens before the subsequent statement after the `join()` method in the main thread. The visualization of these methods ensures that it is possible to establish the happens-before relation for the thread join rule using the captured thread termination action in `ChoiceGenerator`.

![Figure 6.12: Happens-before relation for thread join rule](image)

However, the start and end of the `run()` method are represented by the same location string in the VA4JVM which may confuse the viewer in complex program execution. In addition, in the current version of the VA4JVM tool, the `join()` method is visualized under the thread from where it is called and there is no direct way to identify the thread on which `join()` event is invoked in the thread state view. Whereas visualization of the `TERMINATE` action of the thread in the thread state view and an order in which the `join()` method is highlighted, it may possible to identify the terminated thread. In addition, the execution trace can find out the thread-id of its child thread on which `join()` method is invoked. Execution trace of the producer-consumer problem is used to visualize happens-before relation for thread join, which is shown in the Figure 6.13. Such visualization showed that the establishment of happens-before relation for thread join can be visualized using the extracted JVM trace.
Lock monitor

A lock monitor rule to establish happens-before relation for thread is described in the Subsection 2.4.2. It ensures, the unlock on monitor lock in synchronized block/method is happened before every subsequent acquiring of lock on the same object. The readers-writers problem is described in the Section 6.3, which uses a synchronized block to control access to a shared file among reader and writer threads.

The visualization of the Readers-writers problem using VA4JVM is shown in Figure 6.14, which is used to explain the visualization of the happens-before relationship for lock monitor rule. It shows, Thread-2 releases the acquired lock on RWPrinter before it is acquired by Thread-1. The lock acquisition and release of the lock on RWPrinter is highlighted by the pink color. Purple color arrow indicates that the unlock on monitor lock happened before subsequent acquisition of lock on the same object. This visualization ensures that it is possible to visualize happens-before relation for lock monitor rule in the VA4JVM. Whereas, highlighting of the lock/unlock events using same location string may confuse the viewer in complex program analysis. The visualization of lock/unlock events in thread state diagram help to identify these captured
Figure 6.14: Happens-before relation for lock monitor rule

Figure 6.15: Happens-before relation for lock monitor rule using execution trace

Visualization of this happens-before relation is also possible in the JVM execution trace, which is shown in the Figure 6.15. The logged information helps to identify the object on which lock is acquired or releases.
along with its thread id. However, the resulting execution trace with so many interleavings and a lot of condensed information makes it difficult to visualize this happen-before relation. Hence, for efficient visualization of this relation, there is a need of extensive program analysis.

- **Transitivity**
  
The transitivity rule to establish the happens-before relation among threads is described in the Subsection 2.4.2. It ensures, if event-A visualized the effect of event-B and event-B visualized the effect of event-C then event-A can be visualized the effect of event-C.

<table>
<thead>
<tr>
<th>Order of event</th>
<th>Event name</th>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>beforeWrite()</td>
<td>Sky blue</td>
<td>It is used to check if the writer thread is allowed to write or not based on checking if the reader thread is active or not.</td>
</tr>
<tr>
<td>2</td>
<td>doWrite()</td>
<td>turquoise</td>
<td>If no reader thread is active then the writer thread is allowed to write to file.</td>
</tr>
<tr>
<td>3</td>
<td>afterWrite()</td>
<td>Irish</td>
<td>Once the writer thread is done with writing it notify reader thread.</td>
</tr>
</tbody>
</table>

Table 6.2: Events used to perform a write operation.

A visualization of the Readers-writers problem using VA4JVM tool is shown in Figure 6.3, it shows the visualization of the happens-before relationship. As described in the program, write operation to the file is performed using three functions described in the preceding Table 6.2. In order to perform the proper read-write operation to file, it is important each event needs to visualize the effect of its proceeding events as described in Table 6.2. Highlighting of these functions in the VA4JVM ensure that, doWrite() method effect is visualized of the beforeWrite() method and afterWrite() method effect is visualized of doWrite() method. Hence, for this, it ensures that afterWrite() method visualized the effect of beforeWrite(), as at the time of the afterWrite() method writer thread know that reader thread is not active. Hence, visualization of the events of
the readers-writers problem using VA4JVM tool demonstrates that it is possible to visualize the rule of transitivity to establish happens-before relation.

6.7.2 Limitations

Even though captured actions in the ChoiceGenerator for the thread state diagram make it possible to visualize the happens-before relation to some extents in the VA4JVM tool. There are some limitations to this visualization. As the VA4JVM tool has true concurrency, it often ends up with a lot of thread switches in the trace, simply because, two or more threads generate events at the same time. There is no intrinsic need to read the resulting trace with so many interleavings, but as it is not easy to reduce the number of interleavings without an extensive program analysis. Currently, the tool shows the interleavings in the way they were recorded, as opposed to an ideal schedule with minimal interleavings. Hence, with so many interleavings, a representation of the lock/unlock events with same location string and difficult to identify the thread on which join() method is invoked makes it difficult to visualize the happens-relation in the VA4JVM for large programs.

6.8 Runtime overhead

The runtime overhead of programs as preliminary calculated using System.nanoTime(), which returns the current value of the precise available system timer, in nanoseconds. Further, it is converted into milliseconds as shown in Table 6.3. Execution time is measured from the start of the program till the end of the following situation given below.

1. End of the application program without instrumentation.
2. End of instrumentation using AspectJ.
3. The time required for the creation of path instance.
4. The time required to visualize JVM execution trace in the VA4JVM.
<table>
<thead>
<tr>
<th>Program</th>
<th>Program execution time in millisecond ( ms )</th>
<th>Trace extraction time in millisecond ( ms )</th>
<th>Path instance creation in millisecond ( ms )</th>
<th>Time for visualization of JVM trace in the VA4JVM in millisecond ( ms )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>2410</td>
<td>2</td>
<td>522</td>
<td>1524</td>
</tr>
<tr>
<td>Racer</td>
<td>1017</td>
<td>2</td>
<td>848</td>
<td>1748</td>
</tr>
<tr>
<td>Readers-writers</td>
<td>26</td>
<td>7</td>
<td>948</td>
<td>3831</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>416</td>
<td>2</td>
<td>788</td>
<td>36559</td>
</tr>
</tbody>
</table>

Table 6.3: Runtime overhead of program execution.

The table, it reveals that capturing of runtime information using AspectJ does not take much time. Hence, instrumentation using AspectJ adds little overhead to the program execution. In addition, table shows that initializing all the GUI widgets ahead of time, is very slow for large traces, so this clearly requires optimization in future work. The current version of the jpf-visual tool is optimized for a fast interactive display once it is fully initialized. This seems to work well on small traces, but already has problems on medium-sized traces. However, this calculation of execution time just gives preliminary information about the performance of the VA4JVM tool. An efficient and precise way of measuring runtime overhead can be investigated in the future.

### 6.9 Summary

The section revealed to what extent it is possible to visualize the different concurrent events with the VA4JVM. It also shows that by providing the necessary infrastructure it is possible to analyze the deadlock with the VA4JVM. Some issues with highlighting the same location string for lock/unlock pointcut are mentioned. In addition, the section also describes to what extents it is possible to visualize happens-before relation in the VA4JVM along with its limitations. Finally, addresses the runtime overhead of the program. The following section discusses these points in more detail and also provides some solution for better visualization.
Chapter 7

Discussion and evaluation

This section presents the evaluation of VA4JVM based on results obtained from the experiment section and also provides limitations of the tool. Firstly, it explains, how AspectJ is helpful to capture the concurrent events. It also shows the applicability of code refactoring and use of adapter classes for successful integration. Finally, it provides causes of the various issues in the VA4JVM tool and gives preliminary solution to mitigate these issues.

The program analysis typically relies on an abstraction of program execution, such as trace which contains operations and events that a program executes. The execution trace obtained through the instrumentation provides clarification about possible executions and serves to check for concurrency properties. To get a proper understanding of these concurrent runtime events in the JVM execution trace, it is helpful to visualize runtime events with the help of the selected jpf-visual visual analytics tool. Hence, the obtained JVM event trace is further processed to make it compatible with a JPF trace, using JVM adapters and code refactoring. Implementing common interfaces with the help of adapter classes wraps the JPF classes effectively.

The jpf-visual tool processes the error path based on transitions, which consists of the event instructions object, and uses ChoiceGenerator object for thread state visualization. Artificial creation of such low-level pieces of information which are not directly provided by AspectJ, helps to process the JVM trace. In addition, the code is refactored by separation of the JPF specific code and generic code which is related to GUI. This separation allows compiling only GUI specific classes for the VA4JVM. These changes make the JVM trace, JPF compatible and allows JVM trace to successfully integrate with the existing jpf-visual tool. Thread IDs are not identical across platforms. The current thread-based layout of jpf-visual relies on minimal,
consecutive thread IDs. Hence, thread IDs are converted to small numbers in a pre-processing step as described in the Section 5.3. These modifications proposed a generic approach for building the visual analytics tool VA4JVM for regular JVM trace.

7.1 Capabilities and limitations

The various figures in Chapter 6 show that the selection of the open source instrumentation tool AspectJ is capable of capturing most of the important runtime events. An extended version of AspectJ with lock() and unlock() point-cuts help to track the acquisition and release of the lock on an object of a particular thread.

The VA4JVM tool visualizes thread-based events of the JVM execution trace and allows software developers to monitor the acquisition and release of lock events on a different thread. In addition, it visualizes these runtime activities of the thread by a different color bar in tabular layout in thread state view. Visualization of different runtime events, thread activities, and dynamic filtering option help software developers to analyze programs. Furthermore, it gives information that may be related to the failure and also help developers in the analysis of lengthy trace transition information, collected at a runtime.

The line-by-line display of execution traces is not captured in the VA4JVM; this will lead to some statements not showing up in the execution trace. A possible way to overcome this is instrumentation of each basic block, using an extension of the AspectBench Compiler (abc)[50]. However, this instrumentation results in higher overhead and in addition, controlling overhead cost is a real concern in the program monitoring. VA4JVM visualizes complete source location string lines of the captured events in the main panel whereas, AspectJ is not able to capture location string line. Hence, the complete source code line is attained from the application file based on the line number and class name, obtained using AspectJ. An efficient way to capture basic block and location string of events need to be investigated in the future.

In the VA4JVM tool, the instrumentation code added to capture runtime events insert extra instructions. Hence, event capture itself is never atomic. There could be a context switch before the lock is acquired, and this means the order in which the events are recorded, is not the order in which their corresponding instructions execute. In general, AspectJ instrumentation does not show interleavings accurately, because information on the causes of extra thread switches are not known. However, this issue affects tracing in general and is not limited to the VA4JVM tool.
VA4JVM shows a thread switch when threadId is changed, change in activity of the particular thread (lock acquisition, the release of lock, wait/notify event, and start/end of the thread). As it has true concurrency, two or more threads generate events at the same time, which results in an excessive number of thread switches. In addition, the VA4JVM may show many thread interleavings that have no consequence on the global program state. This is due to events being captured by multiple concurrent threads on a modern CPU that can execute many threads at once. Conversely, the current VA4JVM tool can only show one observed interleaving in the execution and shows them in the way they were recorded, as opposed to an ideal schedule with minimal interleavings. If a particular interleaving that may cause a problem, is not captured in the execution trace for that thread schedule, the VA4JVM tool will not be able to visualize that interleaving. An evaluation of the VA4JVM based on the result of program visualization in Chapter 6 is discussed further.

7.2 Evaluation

The figures in Chapter 6 illustrated visualization of the programs. It also confirms that the code refactoring and use of adapter classes make it possible to visualize both the traces through the existing jpf-visual tool for analysis. Visualization of the Racer program using VA4JVM reveals that the tool is helpful to understand the cause of exceptions in the main thread by highlighting access to the shared variable by different threads. It is possible to analyze a particular condition on reader and writer threads by visualization the reader-writer concurrent program using the VA4JVM. Moreover, the VA4JVM tool helps to check if the program is behaving properly or not. The different filtering options help to focus on a particular method or block in such a complex concurrent program. In addition, the thread state view can help to give an overview of the different important actions of the threads. The visualization of server-client program using the VA4JVM ensures that analysis of communication through a network protocol is possible. However, this is not possible through jpf-visual since, JPF does not support the Java libraries such as java.net. For an efficient analysis, there is a need to instrument the standard library in VA4JVM to capture all important events related to this library.

The visualization of dining philosophers problem shows that with the necessary infrastructure, it is possible to analyze deadlock using the VA4JVM tool. The rewritten code using ReentrantLock reliably detects a deadlock if it occurs, it may miss some cases but the deadlock would indeed be real. The visualization gets cluttered with the injected source code. Hence,
the future implementation of an approach for extensive analysis of the execution trace can help to track the list of a pending monitor on each thread. This approach can be useful for a successful analysis of a deadlock program with an original code. Visualization of the producer-consumer program shows that it is possible to track the shared variable accessed by multiple threads with its lock/unlock events in the VA4JVM tool. However, with increasing threads, the overhead to the VA4JVM tool increases, which results in the tool taking more time in event processing and visualization. An approach of grouping of the related events in VA4JVM may help to analyze the data race problem more efficiently in such a complex program.

7.2.1 VA4JVM tool issues related to AspectJ

In AspectJ, advice used to captures the start and end of of the method or synchronized block, captures the same location string. The start and end of the main() method is required to capture the ROOT and TERMINATE actions of the main thread. Capturing the acquisition and release of the lock with a synchronized block and synchronized method is essential to trace monitor. Even though lock() and unlock() are two different pointcuts, they capture the same source location for the lock/unlock events, since it is developed to capture synchronized block. The existing tool provides the same checkbox to highlight lock/unlock events. Hence, distinguishing between these two events is difficult as lock/unlock pointcuts represent the same source string location and with the same color in the main panel. Such a visualization may confuse the viewer for a larger trace. However, yellow and green bars in the tabular layout helps to distinguish between these two events of the thread. By clicking on these color bars, it is possible to go to a particular thread trace in the main panel, but, it is not optimal when looking at complex traces. It would be often useful to have a similar yet different way to show that two or three events are related, which would be very helpful for visualizing lock/unlock events. The current version of the VA4JVM tool cannot do this, although it may be possible in the future.

7.2.2 Happens-before relation in the VA4JVM

It can be assured that as the action of the particular thread is visible to the other thread by giving partial ordering to events in the program, as proposed by Lamport happens-before relationship. The ChoiceGenerator actions of the thread such as, START, JOIN, LOCK, UNLOCK, WAIT and RELEASE help to establish the happens-before relation in the JVM execution
trace. However, this information is admittedly slightly hidden in VA4JVM.

The visualization of the Fibonacci program in the VA4JVM tool ensures that each event in a single thread precedes every other event in that thread, comes later in the program order. The establishment of happens-before relation for the volatile variable is visualized using Racer program in the VA4JVM tool. It ensures that write to a volatile variable is visible to all its subsequent reads by any thread. It may help to maintain data consistency across threads. However, with a lot of thread switches due to true concurrency in the VA4JVM, it is not possible to guarantee the visibility of this relation. The visualization of the start() and run() method of the producer-consumer problem in VA4JVM ensures that the Thread.start() method happens before all the events in the started thread. It shows that a combination of the thread-state diagram and events in the main panel help to visualize the happens-before relation for the thread start rule.

The visualization of the join() and end of the run() method; of Racer problem ensures that all the events in the started thread, finish before all subsequent actions in a thread where the thread.join() method is called. However, the start and end of the run() method are represented by same location string, this makes it difficult to identify the thread on which join() is called. In addition, it may confuse the viewer for visualization of the happens-before relation for the thread join rule in complex program execution. Visualization of the Readers-writers program shows the establishment of happens-before relation for the lock monitor rule. It ensures, the unlock on monitor has happened before every subsequent acquiring of lock on the same object. The representation of lock/unlock events in the thread state diagram help to identify these captured actions in the main panel. In addition, visualization of the Readers-writers program in the VA4JVM tool ensure the establishment of the happens-before relation for transitivity rule. However, a lot of thread switches in the trace cannot be reduced without an extensive program analysis, makes it difficult to visualize the happens-relation for large programs.

7.2.3 Overhead

VA4JVM preliminarily calculates execution time of the program using System.nanoTime(). It ensures that instrumentation adds less overhead whereas, initialization of all the GUI widgets occurs ahead of time. However, this calculated execution time just gives preliminary information about the performance of the VA4JVM. In general, the tool shows its usefulness for analyzing smaller execution trace of the smaller program. For the larger program,
VA4JVM shows its usefulness by allowing user to concentrate on the particular part of events but with some overhead.

7.3 Discussion

In general, visualization of the concurrent runtime events in the VA4JVM tool has shown that the AspectJ instrumentation tool is able to capture most of the concurrent events. However, the added instrumentation code can cause extra thread switches resulting in incorrect interleavings. Apart from that, when determined by the happens-before relation, the order of events may be slightly different from the actions that occurred in the program. However, this issue occurs when debugging code with logging and tracing in general and is not limited to the VA4JVM tool. It is very difficult to understand the data that is traced without visualization but VA4JVM tool solves this problem. It allows developers to selectively mine and visualize the trace, and gain insights, to an extent that is very difficult to achieve by textual log analysis alone.

For more accurate tracing, JVMTI is likely better, but this approach also has drawbacks such as being not portable across different operating systems and potentially not being available in all JVM implementations. The only way to resolve this issue would be to modify the JVM, in which case the modifications likely have other impacts on the program's behavior. This may allow seeing the actual behavior on the modified JVM, but that may not correspond to behaviors on standard JVMs.
Chapter 8

Conclusions

Dynamic analysis can generate a huge amount of execution trace data for analysis. Hence, designing a visual analytics tool that is able to cope with huge amounts of execution trace data and does not confuse the viewer remains a challenge. This thesis proposed a generic solution to tackle this issue by the implementation of the visual analytics tool VA4JVM, which is an extension of the jpf-visual for JVM execution trace. This approach visualizes the concurrent runtime events along with thread activity of the regular JVM execution trace. The filtering of runtime events along with a thread-based view helps to understand the concurrent program and analyze the lengthy trace efficiently. Adapter classes and code refactoring make the JVM trace compatible to JPF. This allows the successful integration of JVM trace into the jpf-visual tool.

An approach of code refactoring, use of adapters and artificially created ChoiceGenerator and instructions objects have shown that it is possible to produce JPF compatible trace for successful integration with jpf-visual. It found that the open source AspectJ code instrumentation, in particular, worked well to capture most of the runtime events. VA4JVM shows that AspectJ instrumentation is sufficiently expressive to analyze the problems and reasonably efficient in most cases.

The implementation preliminarily shown in this work reveals that it is possible to analyze the concurrent events using regular JVM trace. The experiments and results in Chapter 6 show the program visualization using VA4JVM. It shows that it is possible to visualize the existing runtime features except line-by-line display of execution trace of the tool with regular JVM trace. In addition, it is possible to visualize the happens-before relation to some extent. However, there are some limitations to visualize happens-before rela-
There is a need to have a better way to visualize lock/unlock events and grouping of such related events for better analysis. This visualization helps to understand the concurrent program and analyze concurrent issues. Furthermore, AspectJ restricted join point model and limited weaving capability reduce code coverage.

The current version of the VA4JVM shows that deadlock analysis is possible with code rewriting with ReentrantLock. This rewriting step may clutter the visualization. Hence, there is a need to find an efficient way to analyze the deadlock with the original code. In addition, lock/unlock pointcuts for synchronized blocks capture the same source location, which results in highlighting of the same string location for lock and unlock events. Such visualization for complex trace confuses the viewer, hence use of a way to show that two or three events are related will resolve this issue. This way of visualizing lock/unlock events in the VA4JVM helps to get a better understanding of these events.

VA4JVM shows the excessive number of thread switch due to multiple concurrent threads executing at once. Hence, there is increase in the overhead to process larger program trace which restricts use of the current version of the VA4JVM tool to smaller program. The current VA4JVM can only show one observed interleaving, and the visualization may not relate interleavings. This may cause a problem if one does not actually observe in an execution that uses such a thread schedule.

The VA4JVM achieved a trade-off between collecting many concurrent events and processing such events. It results in getting a transition of thread along with its events before a thread switch. However, some events may be independent of each other, so grouping related events of same thread to one transition in the future may help to get a better visualization. The tool needs further improvement such as implementation of happens-before relation or combining related events for a better visualization of lock/unlock events that can allow the VA4JVM tool to be used for a larger program analysis.
Chapter 9

Future work

Many of the ideas mentioned in this section still need to be implemented and refined further in the VA4JVM, for an efficient implementation of visual analytics tool for JVM. Although AspectJ is expressive, there are several limitations to the current AspectJ implementation, such as, restricted join point model and limited weaving capability, which reduces code coverage. To cover all runtime monitoring, AspectJ needs to perform statement level weaving, instrument basic block, loops and capture local variable access. However, in the current VA4JVM, this information is not captured by AspectJ hence, it is necessary to use its extension AspectBench Compiler to add several new pointcuts [50]. In addition, the DiSL [51] instrumentation framework for runtime verification also overcome these limitations by using a DiSL compiler to unmodified AspectJ-based runtime verification tool. It results in a significant increase in code coverage and allows the developers to find a violation in the Java class library [51].

Capturing thread actions for ChoiceGenerator helps to visualize happens-before relation in VA4JVM to some extent. However, the current visual analytics platform of VA4JVM is not exactly designed to handle the happens-before relation and has some limitations. A better way of visualizing lock/unlock events in VA4JVM can help to track lock monitor events efficiently. An approach of storing events execution trace in the database until termination of the program and accessing these events from the database while processing can allow tracking of the pending locks on particular thread. This approach may help to track the pending lock to detect and check the deadlock condition with original code. A future implementation of this approach in the future may allow grouping related events, reduction of the thread interleaving through extensive trace analysis, detection of the deadlock using original code, and to
visualize the happens-before relation in VA4JVM. Such a strategy in the future will help VA4JVM tool to be made applicable in many contexts and visualize any type of data race or deadlock analysis.

In addition, storing runtime captured events into database make the processing and visualization of events by an external process which allows to use shutdown hook in jpf-visual. In this external procedure, captured events will be processed from the database and converted into data structures. This may allow the use of shutdown hook for graceful termination of all running thread and gives a visualization after the program really terminated. In addition, this approach will also help to start the visualization of the application which is terminated using ctr-c in case of deadlock.

Different types of diagrams in the VA4JVM are useful for certain problems, so it helps to have a choice. A sequence diagram shows how data from different classes are accessed and processed by methods in different classes. Such a sequence diagram will be helpful to debug data dependencies. UML supports concurrency and makes it possible to represent the concept in different kinds of diagrams, such as activity diagram and state machine diagram. UML activity diagrams can be useful for understanding system functional behavior \(^{[52]}\) and visualizing a live stream of events from a running program. In order to create the jpf-visual tool specialized for the events from the JVM, in this case, events are processed directly without creating artificial instruction objects. This will reduce the overhead of creating a data structure. Analysis of various concurrency issues will be efficient by use of different diagrams.
Bibliography


Appendix A

Event processing pseudocode

The pseudocode for event processing to create an error path from the right type of instruction

// Generation of the right type instruction based on captured event
Instruction insn = new Instruction();
addPreviousStep()
{
    // Update the ChoiceGenerator bases on the action of the thread
    cg = new ChoiceGenerator();
    currentThreadId = Thread.currentThread().getId();
    // Change in thread
    if ((tr == null) || (currentThreadId != threadId))
    {
        // If there is thread change, then adding the transition to stack not for the first event
        if ((tr != null))
        {
            addPreviousTr(instance.tr);
        }
        thread = new ThreadInfo();
        tr = new Transition(cg, thread);
    }
    /* When important thread action of ChoiceGenerator is captured */
*/
else
    if((cg.getId()=="ROOT"||cg.getId()=="START"||cg.getId()=="JOIN"||
    == "TERMINATE") )
    {
        addPreviousTr(tr);
        //Create threadInfo
        thread = updateThreadInfo();
        //create new transition
        tr = new
        se.kth.tracedata.jvm.Transition(cg,thread);
    }
assert(insn != null);
step = new Step(insn);
tr.addStep(step);
}
void addPreviousTr(Transition tr) {
    assert (tr != null);
    stack.add(tr);
    tr = new Transition(null,null); // reset transition
    record
}
//Error path is created bases on added transition to the stack and application name
Path p= new Path(app,stack);
Listing A.1: Event processing pseudocode
Appendix B

Deadlock analysis

B.1 Modified dining philosopher

The below modification in the code helps to detect the deadlock in dining philosophers problem and start the program visualization in VA4JVM.

```java
public ReentrantLock rightFork;
public ReentrantLock leftFork;
static Object globalLock = new Object();
public void run() {
    while (true) {
        synchronized (globalLock) {
            rightFork.lock();
        }
        synchronized (globalLock) {
            while (leftFork.isLocked()) {
                detectDeadLock();
                try {globalLock.wait();}
                catch (InterruptedException e) {e.printStackTrace();}
                leftFork.lock();
                detectDeadLock();
            }
        }
        synchronized (globalLock) {
            leftFork.unlock();
            globalLock.notifyAll();
        }
    }
}
```
 synchronzied (globalLock)
{
    rightFork.unlock();
    globalLock.notifyAll();
}
}

private void detectDeadLock() {
    for (Thread t:global.getThread()) {
        if ((Thread.currentThread().getState().toString()=="RUNNABLE") && (t.getState().toString()=="RUNNABLE")
            countval++;
    if (countval >= 1) {
        System.out.println("Deadlock...");
        //Start GUI visualization
        global.displayErrorTrace();
    }
}

Listing B.1: Modified dining philosophers problem for deadlock analysis