BEAT: A VISUALIZING DEBUGGER FOR CONCURRENT, SHARED MEMORY JAVA PROGRAMS.

A thesis presented in fulfillment of the
requirements for the degree of
Master of Science
in
Computer Science

At Massey University, Palmerston North, New Zealand.

PAUL THOMAS JOHNSON

February 2011

Abstract

This thesis presents our research into the creation of a new concurrency visualization called Beat. Our research began with the observation that software used to create and record music has been incredibly successful and that there were broad similarities between these pieces of software and existing concurrent visualizations. This led us to question if there were reasons why music software had been successful, why concurrent visualizations hadn't been as comparatively successful and finally if this could teach us anything about building better visualizations.

The existing literature was examined to learn more about concurrency, visualization and music software and notations. For concurrency we wanted to see what existing solutions to the problems of concurrency exist and in particular existing solutions to concurrent debugging. With existing visualizations we wanted to see if there was anything we could identify about them that made them ineffective for solving the problems of concurrency. Conversely we wanted to see if we could identify what made music software and notations so effective for solving problems for musicians.

The examination of the existing literature led to the design and implementation of the Beat software, based on the ideas we had discovered in our search of the literature. To test how effective the design and software was we conducted an evaluation with programmers, which validated many of the decisions we had made and gave us a number of future directions to take with our research.

The main contribution of this thesis is a new approach to designing concurrency visualizations that emphasizes using the low level details of program execution and integrates execution data into a single view to help with the error debugging process. This in contrast to existing concurrency visualizations that focus on the debugging of performance problems.

Acknowledgements

Without the tireless support, encouragement, supervision and regular prodding of Dr Stephen Marsland this thesis would never have been completed. Thanks Stephen.

Thanks to Dr Elizabeth Kemp for valuable advice regarding testing the Beat software.

Thanks to Professor Hans Guesgen for help planning testing of the software.

Approval for the evaluation was granted under the low risk guidelines provided by the Massey University Human Ethics Committee.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Thesis Overview . . . 2

2 Literature Review **4**

2.1 Introduction . . . 4

2.2 Introduction to Concurrency . . . 4

2.2.1 Instruction Level Parallelism . . . 5

2.2.2 Data Parallelism . . . 7

2.2.3 Task Concurrency . . . 7

2.2.4 Models of Process Communication . . . 8

2.2.5 Concurrency and the Operating System . . . 12

2.2.6 Debugging Concurrent Programs . . . 15

2.3 Introduction to Visualization . . . 20

2.3.1 Elements of Visualization . . . 21

2.3.2 Visualization Data . . . 24

2.4 Software Visualization . . . 27

2.4.1 Visual VM . . . 28

2.4.2 PARADE . . . 33

2.4.3 Understanding Complex Multithreaded Software Systems by Using Trace Visualization . . . 37

2.4.4 Zinsight . . . 38

2.4.5 TIE: Thread Interleaving Explorer . . . 42

2.4.6 UML Sequence Diagrams . . . 43

2.4.7 Visual Programming Languages . . . 48

2.4.8 Common Features Summary . . . 51

2.5 Temporal Notations from outside Software Visualization . . . 52

2.5.1 Music Notation 52
2.5.2 Non-Linear Audio Editing 54
2.5.3 Comparison to Software Visualization 58

3 Design **60**
3.1 Design of Beat 60
3.2 Visualization Data 60
3.3 Space 61
3.4 Text 61
3.5 Lines, Colors and Icons 63
3.6 Loops, Branches, Errors and Exceptions 66
3.7 Context Switches 68
3.8 State 68
3.9 Concurrency Errors 70
3.10 Interaction 72

4 Implementation **74**
4.1 Implementation Choices 74
4.1.1 Language 74
4.1.2 Standalone program versus IDE 75
4.1.3 Gathering Trace Data 76
4.1.4 Visualization display choices 82
4.2 Plugin Infrastructure 82
4.2.1 Launcher and the JDT 83
4.3 Data Gathering 84
4.3.1 Trace data annotations 84
4.3.2 Source Code instrumentation 85
4.3.3 Runtime method probes 87
4.3.4 Data loading and processing 87
4.3.5 Context Switches and DTrace 90
4.3.6 Program State and Concurrency errors 91
4.4 Visualization Generation 91
4.4.1 Ruby and Templating 91
4.4.2 Text Positioning 92
4.4.3 Canvas Object and Threads 96
4.4.4 JQuery and CSS 97

5 Evaluation **98**
5.1 Evaluation Introduction 98
5.2 Evaluation Method 98
5.2.1 Tasks and Timing 99
5.2.2 Previous Experience 99
5.2.3 Timing and Experience Questions 100
5.2.4 UI Questions and Comments 100
5.3 Timing and Quantitative Questions Results 101
5.3.1 Was using Beat significantly faster? 101
5.3.2 Was Beat helpful in one not the other? 102
5.3.3 Was using Beat significantly easier? 102
5.3.4 What effect did experience have? 102
5.3.5 Did the order that Beat was used in have any effect? 103
5.3.6 Was one of the tasks harder than the other? 103
5.4 Visualization Design Questionnaire Results 104
5.5 Comments From Questionnaries 104
5.5.1 Like Comments 105
5.5.2 Minor UI problems 105
5.5.3 Major Dislikes/ Needed Features 106
5.6 Observations of Participants 106
5.7 Evaluation Limitations 107

6 Conclusion **108**
6.1 Conclusion 108
6.2 Future Work 110
6.2.1 Testing 110
6.2.2 Core Improvements 110
6.2.3 UI improvements 111
6.2.4 Debugger and Static Analysis 112
6.2.5 Education 112
6.2.6 Visual Language 112

A Questionnaire **113**

B Evaluation Source Code **114**
B.1 Source Code for Deadlock Task 114
B.2 Source Code for Race Condition Task 114

C Result Tables **124**

C.1 Timing and Experience Results 124

C.2 UI Questions 125

C.3 Comments 125

List of Figures

2.1 Pipelined instruction execution 6

2.2 Race condition diagram 8

2.3 Race condition fixed with locks 9

2.4 Two processes in a deadlock. The processes have each acquired a lock but are both waiting on the lock held by the other. Since the processes are unable to acquire the locks they need to continue they are unable to progress. 10

2.5 Processes messaging shared account process 11

2.6 Screenshot of Intellitrace [14] showing a list of the events in an executing program the system has traced, including exceptions and user interface events. 18

2.7 Treemap example [9] that shows the amount of downloads for software projects on sourceforge. This is an example of using space to convey amount and the relationship of the amount of one thing to another. . . . 22

2.8 Examples for continuous data from Tory and Mller 26

2.9 Examples for discrete data from Tory and Mller 26

2.10 Process overview for VisualVM [24] showing some basic bits of information about a process including the execution arguments and the process id. . . 29

2.11 Monitor view of VisualVM [24] that shows various graphs showing different aspects of a programs performance. The top left shows the CPU and Garbage collection activity. The top right shows the memory usage and the type of memory being used. The bottom left shows information about classes and the bottom right shows information about threads. . . . 30

2.12 Thread timeline view of VisualVM [24]. The timeline shows the thread state over time. 31

2.13 Sampler view of VisualVM [24]. 32

2.14 Profiler view of VisualVM [24] showing how long methods are taking to execute. 33
2.15 Threads view of Parade [54, 66] showing a list of threads as boxes. Colour is used to show state. 34
2.16 Functions view of Parade [54, 66] shows a trace of what functions have executed for a single thread. 35
2.17 History view of Parade [54, 66] showing the timeline of thread execution, somewhat like the Visual VM example. 36
2.18 Mutex view of Parade [54, 66]. The small coloured circles are threads that move inside the larger clear circle that represent the Mutex when they acquire it. 36
2.19 Barrier view of PARADE [54, 66] showing what threads have entered and exited a barrier synchronization point. 37
2.20 Trace Visualization showing two threads [62]. Each thread "lane" has an overview (top) and zoom in panel on the bottom that show what functions have been executing. 38
2.21 Event view of Zinsight [33]. Note that time flows vertically rather than horizontally like the previous examples. 39
2.22 Statistics view of Zinsight [33]. Some what like the profiling view of Visual VM this provides statistical information about the running program. . . . 40
2.23 Sequence Context view of Zinsight [33]. This is showing a summarization of the various execution paths the program has gone through. 41
2.24 Thread Interleaving Explorer software [42]. Note the numerous different panels used to provide enable a user to zoom in from the more general information at the top to more specific information in the bottom columns. 42
2.25 UML collaboration diagram [23]. The numbers show the order of method calls between the object instances which are represented by the boxes. . . 43
2.26 Basic UML sequence diagram [12]. Time flows vertically with the lines representing the execution history of the object instances and the narrow blocks on the lines showing when a method was called on an object. . . . 44
2.27 UML Sequence diagram with combined fragments [12]. The boxes are used to represent various aspects that are not covered by the lines and execution boxes. 45
2.28 Diagram of saUML [65]. Colour is used to represent thread state and the rounded corner boxes show monitor state information. 46

2.29 Context switches in Artho's UML extension [27]. Note the hexagon threads and the dashed lines that show when a thread starts executing in the object instances. 47
2.30 Condition variable wait and notification in Artho's UML extension. Note the dotted lines which demonstrate notification of a lock to the waiting thread. 47
2.31 Overview of JIVE [36] showing the various panels. The top left shows the object instances and there values the bottom left shows a sequence diagram of the execution. 48
2.32 Multiple threads in JIVE [36]. Colour is used to differentiate the threads. 49
2.33 LabVIEW – an example of a "boxes and wires" visual language [20][6]. "boxes" are operations on a stream of data that are carried between the operations by the "wires". Some "boxes" will be used for input and output. 50
2.34 Basic music notation 53
2.35 Different note lengths 53
2.36 Text and lines indicating volume. 53
2.37 Speed notation for music notation 54
2.38 Glyphs used in music notation 54
2.39 Track view of Logic Audio 55
2.40 Piano roll view from Logic Studio 56
2.41 Sound edit view from Logic Studio 57

3.1 Use of space in Beat 62
3.2 Method calls to object instances represented as blocks in the object tracks. 62
3.3 Design with source code included in method call boxes. 63
3.4 Adding lines showing path of execution between method blocks. 64
3.5 Thread state being shown by icons and dashed lines. 65
3.6 Thread looping shown by repeated loop body. 66
3.7 Branches shown by only showing the content of the branch taken. 67
3.8 Exceptions in a program thread shown by T, C and E icons. 68
3.9 Context switch in Beat 69
3.10 State changes shown by additional lines 70
3.11 State only mode for method blocks. Source code is hidden and only changes to program variables are shown. 71
3.12 Possible method of highlighting a program deadlock. 72

4.1 Class diagram of Beat model relationships 89
4.2 Space compression in method blocks 94
4.3 Parallel thread execution 95

List of Tables

5.1 Participants and tasks performed 99
5.2 Comparison of time taken on both tasks 101
5.3 Correlations between Eclipse and concurrency experience 103
5.4 Comparison of Beat order performance 103
5.5 Task difficulty comparison 104
5.6 Participants rating of what was task was easiest. 104

List of listings

1 Example for Instruction Parallelism 5

2 Simple example of a future in Java [5], the line containing future.get() is where the other thread is started and the main thread blocks waiting for the value to be generated. 13

3 Probing an open() system call with DTrace [22]. The lines containing “syscall” are the events in the kernel to trace and the line beneath is a predicate which checks whether the action contained in the body of the trace will execute. The trace body is written in a programming language allowing complicated tracing actions to be performed. 17

4 This example shows how to launch a program using the JDI library [15]. . 77

5 Registering to receive method entry events from remote program [15]. . . 77

6 Method to receive registered events from the debugger [15]. 78

7 Probing methods using AspectJ pointcuts. 80

8 Example of part of an ANTLR grammar used for processing a block of Java code to add tracing probes. 81

9 Configuration of a launcher extension for Eclipse in the XML plugin configuration. 83

10 Diagram showing basic tree structure of Java source code. A classes is made up of methods which contain blocks of statements. 86

11 Code to insert method probes around a method call. The lines containing ExpressionStatement are the code for the methods to insert. The lines like x.add(beforeExp) add the method probes. 88

12 Diagram showing the result of inserting probe methods before and after method call. 89

13 Probing scheduling in the kernel 91

14 ERB templating code 93

15 Javascript line drawing code 96

16 Contains main method of deadlock test. 115
17 Printer object for deadlock test. 116
18 First printer thread for deadlock test part 1. 117
19 First printer thread for deadlock test part 2. 118
20 Second printer thread for deadlock test part 1. 119
21 Second printer thread for deadlock test part 2. 120
22 Contains main method of race condition test. 121
23 Printer thread for race condition test. 122
24 Shared object for race condition test. 123

Chapter 1

Introduction

1.1 Motivation

The microprocessor industry has reached the point where the performance of an individual CPU core is not seeing significant improvement. This is largely because of the practical limit of keeping a CPU core cool. Additionally techniques such as pipelining and out of order execution have reached the limit of how much parallelism can be drawn from a single thread of execution [57].

While Moore's Law still holds for now, as transistor feature size has continued to shrink, the additional transistors have been used to add more CPUs to a single chip rather than to increase the performance of single CPU. To the programmer this means that there is a limit to how fast a single thread of execution can be expected to perform, so to achieve continuing performance gains requires the use of multiple threads. As we will see, this can be very difficult to get right.

Considerable effort (and money) has been spent to make this task easier for programmers. One of the core problems is communication between threads. A number of different methods of conducting communication between threads such as the use of locks and message passing have been developed with some success. Formal methods have also been developed to address the communication issue, but there are often difficulties in getting these tools into wider use. Erlikh [35] notes the importance of the debugging process with a great deal of time spent "programming" actually spent debugging. A variety of visual and non-visual tools have been created to assist this process, but they seem to be primarily focused on the area of performance analysis.

Our motivation is the observation that in areas outside of programming and software visualization, such as music notation and music recording, visual methods have been a tremendous success, in the sense of being widely adopted by the market. Early on we observed similarities between these fields and several of the visualization tools we investigated in our initial search of the literature. The first was a similarity between the kinds of visualization techniques used – both used a timeline-based approach where events are ordered by when they occur in time along a horizontal or vertical line or track. This lead to the observation that both concurrent programs and music notation and music recording are both strongly time-based, and are both, at least in a superficial sense, time-based "media".

This led us to our thesis questions:

> *Is there something about existing visualizations that make them unsuitable for the task of programming and debugging concurrent programs?*
>
> *What is it about music notation and music recording software designs which make them effective?*
>
> *Can we draw any conclusions from existing visualizations, music notation and music editing software and apply the lessons learned from them to the design of a visualization-based concurrent debugger?*
>
> *Will the resulting design be effective for the task of debugging concurrent programs?*

Our research and experimentation is primarily focused on tools for experienced developers, this leads to some additional practical considerations in that the software must have adequate performance, be relatively easy to use and integrate well with existing tools. It is possible that our research could be broaden to include questions about the creation of tools aimed at education, however this is not something we are pursuing at this time.

1.2 Thesis Overview

In this thesis we will address how we have answered our thesis questions and discuss the software which we have produced, called Beat.

In the next chapter we will review the existing literature. To help situate our research we will provide an overview of concurrency with some discussion of existing concurrent debugging techniques. Then we will provide a brief overview of the field of visualization before examining existing software visualizations for concurrency. Finally, we will discuss music software and notations and compare them to the software visualizations that we researched.

In the third chapter we will discuss our design and how it draws from and builds on existing debuggers, visualizations and music software and notations to create a novel design for debugging concurrent programs.

In the fourth chapter we will give an overview of how we implemented our design and the choices we made and challenges we faced while implementing it.

How we evaluated the software and the results of our evaluation will be discussed in the fifth chapter.

Finally, in the sixth chapter we will summarize and provide some concluding remarks about this thesis and some possible future directions we could take the ideas developed in it.

Chapter 2

Literature Review

2.1 Introduction

In the first part of this review we will investigate concurrency and some of the unique challenges it poses, followed by some of the existing approaches to addressing those challenges. Next we will provide a brief overview of the field of visualization before focusing on the software visualization area of trace visualization. Finally, we will discuss the design of music notation and software before concluding with a comparison of music notation and software and the trace visualization systems we have studied.

2.2 Introduction to Concurrency

In this section we will introduce the three categories of parallelism concurrency, instruction parallel, data parallel and task parallel. Next we will address some of the challenges of communication in task parallelism and some of the solutions proposed to address these challenges. We will also address the role of the operating system in concurrency and some techniques for how concurrent programs have traditionally been debugged.

We will use the definition of concurrency and parallelism found in Sun/Oracles multithreaded programming guide [56]. Parallelism is performing two or more tasks at the same time. Concurrency is performing two or more tasks in a period of time, potentially (but not necessarily) in parallel. Concurrency includes time-slicing “virtual”-parallelism that operating systems perform, which we will discuss further in the section 2.2.5.

While performing tasks at the same time isn't difficult, it can bring up significant issues related to the data that is operated on. Most of this section will be related to discussing how this affects programming concurrently.

2.2.1 Instruction Level Parallelism

Instruction level parallelism is not a single technique, but rather a variety of techniques that can extract parallelism from a stream of machine code instructions.

For example, the code in Figure 1 shows two simple integer additions followed by a multiplication. Since there is no dependency between the calculation of a and the calculation of d if a CPU has two integer addition units it could perform both calculations at the same time. This kind of instruction parallelism is called super-scalar execution. Modern CPU architectures (such as Intel's Nehalem architecture) can perform 3 computational operations per cycle such as integer addition and floating point multiplication [19].

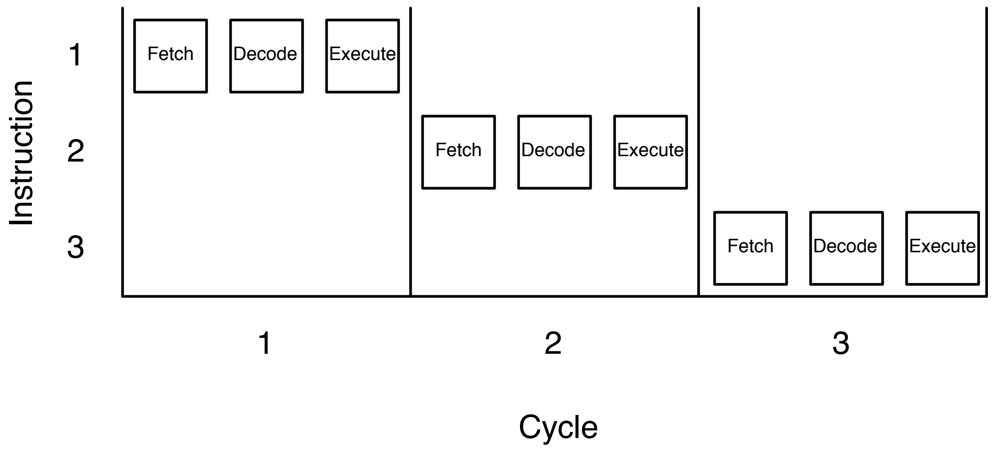
```
void test(int b, int c, int e, int f){
  int a = b + c
  int d = e + f
  int g = a * d
}
```

Listing 1: Simple example showing code with instructions that can be easily executed in parallel.

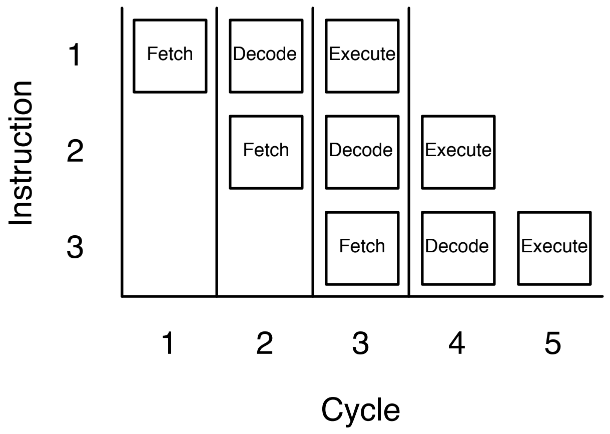
Operations such as addition are broken down into multiple processor steps such as fetching the instruction, decoding it and executing it (or potentially many more). Rather than executing all these steps in a single cycle of the CPU, modern CPUs execute each of these stages once per cycle so that at every clock cycle a fetch, decode and execute is performed. This technique is called pipelining. Figure 2.1 shows instruction execution without pipelining and with pipelining.

A common problem with pipelining is the issue of dependencies, if a later calculation depends on an earlier one then the pipeline either has to stall to wait for previous operations to complete or special hardware has to be used to send values from earlier instructions back to later instructions. Branches in a program also cause a problem, since a branch can cause the entire contents of the pipeline to be emptied and restarted at another point in the program.

Out-of-order execution is a technique that executes instructions as the data for them becomes available, rather than in the order they are specified in the program. This means that more recent instructions can complete before older instructions that are waiting on data from main memory. For example in the code in Listing 1 if b or c aren't available in registers and have to be brought from main memory but e and f are then e + f may execute before b + c. Problems can arise for concurrent programmers creating



(a) Without Pipelining



(b) With Pipelining

Figure 2.1: Instruction execution without and with pipelining. Parallelism can be seen by different phases of instruction execution occurring in the same cycle unlike the non-pipelined example where each execution takes a whole cycle that is longer. [46].

lock implementations, since out-of-order execution may reorder instructions that are supposed to be inside a lock to be outside it. To address this, computers that execute out-of-order have special instructions to ensure that all instructions are executed before a point called a memory barrier, preventing instructions from being executed after it.

Branch prediction is a technique that tries to predict the destination of a branch instruction. This is used with speculative execution, which pre-executes code that may or may not be executed depending on whether a branch is taken or not.

Most of the techniques discussed here aren't visible to programmers since they are usually hidden within CPUs, so we won't say more about them in this thesis. For more information, see e.g. Tanenbaum [59].

2.2.2 Data Parallelism

Data parallelism is essentially concurrency in its simplest form: each task is the same and has its own pieces of data to work on and no data is shared between tasks. This is commonly known as Single-Instruction-Multiple-Data (SIMD) since each task will execute the same sequence of instructions, but each will have their own data.

One of the most common examples of this are modern graphics cards, which are essentially data parallel computers processing vector data such as polygons and textures to produce images.

This does not mean that the only kind of data parallelism is performed by specialized data parallel hardware, more that data parallel is a pattern of work consisting of the same code working on multiple pieces of data and communicating in a fairly fixed and synchronous pattern. Many types of processing performed on large clustered multi-computers would fit this definition.

2.2.3 Task Concurrency

Task parallelism is multiple different tasks working on multiple different pieces of (potentially shared) data, commonly known as Multiple-Instruction-Multiple-Data (MIMD). In certain cases the tasks will operate on entirely separate pieces of data, this kind of problem is commonly called an "embarrassingly parallel" problem. A common example of this kind of concurrency is serving web pages, where requests are handled simultaneously with no data shared between requests.

Of course, this will not always be the case and data sharing can lead to many problems. This is the challenge of concurrency: making multiple threads communicate effectively to solve problems. In the next section we will outline these problems and how people have addressed them.

Throughout the rest of this thesis we will be referring to this type of concurrency since it is in many ways the most difficult to solve and most relevant to programmers.

2.2.4 Models of Process Communication

Shared Memory and Locks

This model is a direct extension of the existing sequential programs. Processes each have their own stack, but share regions of memory, which can be accessed by any of the processes allowing communication between them.

A major problem with this approach is called a race condition or data race. In Figure 2.2 we see a common example of a data race. Two bank processes accessing a shared memory location, the account. Process 1 reads the memory location into local memory, performs an addition operation, but before the thread can write the value back to memory another thread runs. Process 2 reads, modifies and writes back to memory. Process 1 runs again and writes back to memory, leading to an incorrect value.

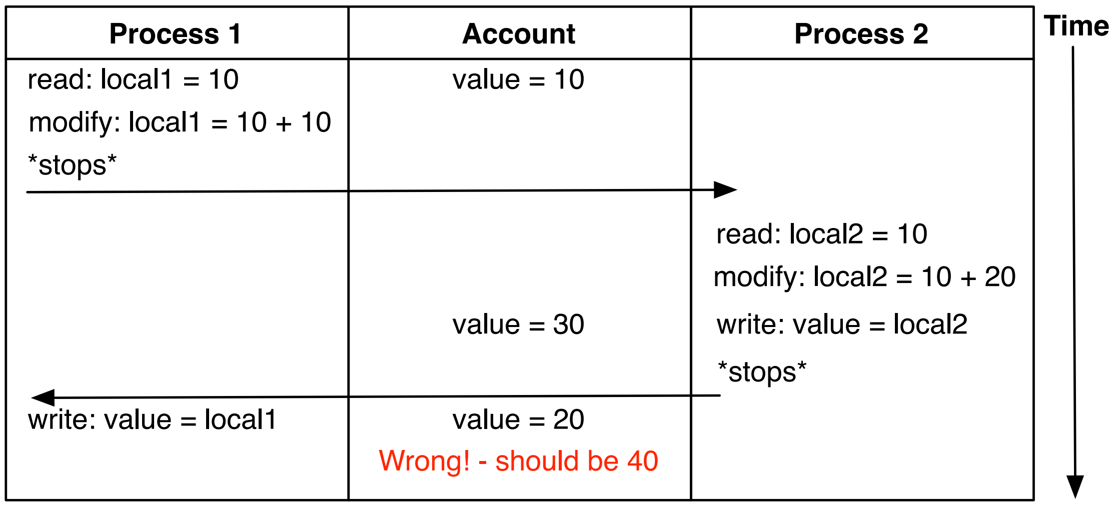


Figure 2.2: Two processes update a shared memory location causing a race condition. The process columns show the instructions executing in time and the center column shows the value of the shared value as it is set by the processes. Arrows show where the execution of the instruction sequence changes from the first to the second process.

To prevent race conditions a number of techniques for creating regions of code where only a single thread can execute have been devised. These techniques are commonly called locks. In Figure 2.3 we can see the above race condition with locks added, making the read, modify, write operation an “atomic” or uninterruptible operation. Many languages have locks available as libraries or built in as Java has with its synchronized keyword.

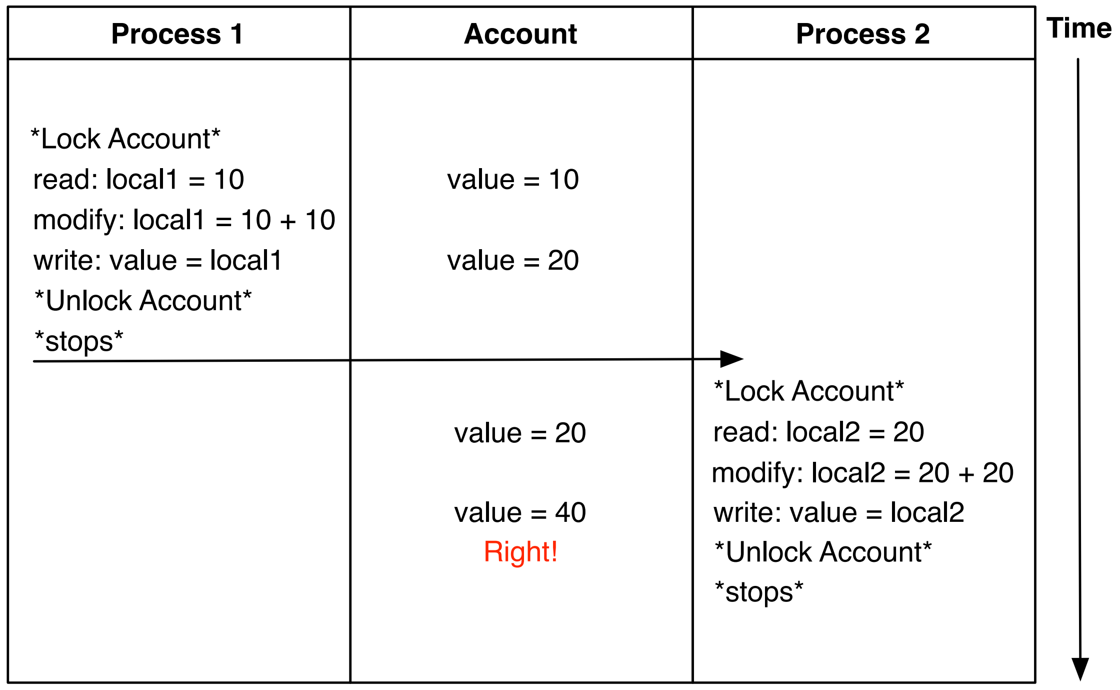


Figure 2.3: Fixing the race condition by adding locks. The locks prevent another process modifying the shared memory value while the code to modify the value is executing.

Adding locks fixes the problem of race conditions, but can create problems of their own, such as deadlocks which arise when two or more locks are locked in such a way that a process is waiting for a lock held by another process, but that process is waiting for a lock held by the first meaning that neither can do anything. Figure 2.4 shows a graphical representation of this.

Detecting deadlocks at runtime is difficult and proving a program is deadlock free is also difficult.

Starvation and Livelock are related problems, starvation is when a high priority process is waiting on results from a lower priority process. Little work gets done because the high priority thread prevents the low priority thread from running. There are a couple of ways this can be addressed, one is to fix the scheduler so it allows lower priority processes more opportunity to perform work.

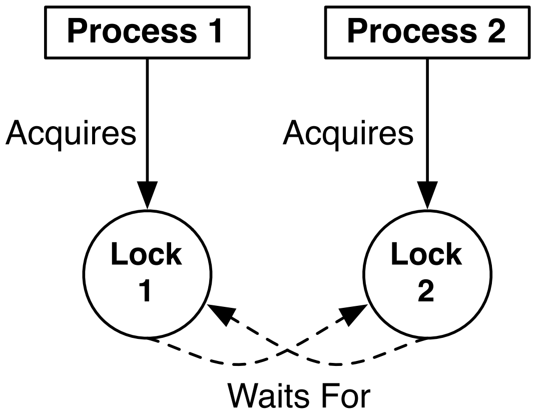


Figure 2.4: Two processes in a deadlock. The processes have each acquired a lock but are both waiting on the lock held by the other. Since the processes are unable to acquire the locks they need to continue they are unable to progress.

Another is to rewrite the program using condition variables. This is a mechanism that is often used with locks that allows a process to wait (block) for another process to signal that it should continue. Because other processes might need the lock that the process is holding when the process waits the lock will be released after the process decides to wait.

Livelock is a more serious version of starvation, it can be described as a process that is active (running), but busy waiting on a condition that will never occur. This means that a process is alive, but not doing any useful work.

Finally, locks suffer from a human problem: in practice they are difficult for programmers to understand and it is difficult to implement programs using them correctly.

More information about the problems of locks can be found in an article called 'The Problem With Threads' [41].

Actor Model

Due to the limitations of the shared memory communication model, a number of other models have been tried to address its problems.

The actor model [37] is roughly analogous to packet-based networking systems such as IP. Processes (actors) without shared memory send messages to each other through some communication mechanism, often a queue in memory. This can be synchronous, where a process waits for a response after sending a message, or asynchronous, where a process sends a message then carries on running.

The actor model doesn't suffer from race conditions since no memory is shared, and all access to remote processes are automatically serialized by the messaging mechanism. Figure 2.5 shows the race condition example rewritten using actor model. The bank is modeled as an independent process receiving add messages from the teller processes.

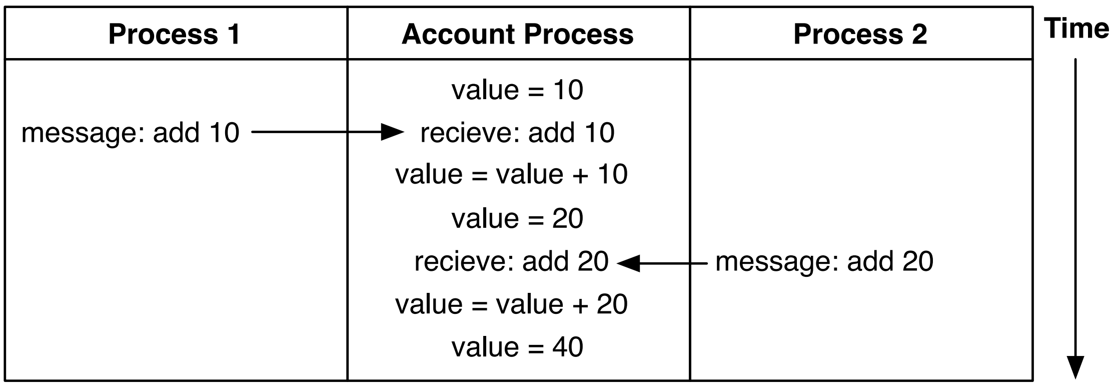


Figure 2.5: Two processes sending messages to an account process to update the value contained in the account. This avoids the problems of locking by serializing access to the account value by using a queue.

Deadlocks are still possible in the actor model: if two threads wait for a message from each other with no timeout then no progress will be made.

Probably the main advantage of this model for programmers is that it is familiar, easy to understand and reason about, making writing complex concurrent programs easier because inadvertent deadlocks and race conditions are easier to avoid.

Unlike the shared memory model the actor model has been mathematically formalized, allowing things such as proving the lack of race conditions and deadlocks. Hoare's CSP [38] is one of the original formalizations of the actor/messaging model.

Given the serious problems with the shared memory model it is perhaps surprising that the actor model is not more widely used. This may be changing as the language Erlang [13], which uses the actor model for concurrency is achieving some commercial use.

Transactional Memory

Transactional memory [50] is a relatively new approach that is similar to how SQL database transactions work. Access and operations on shared variables are wrapped in a transaction so that if a concurrent access occurs the changes can be rolled back and retried. This is similar to how the BEGIN, COMMIT and ROLLBACK statements work in SQL databases.

Because transactional memory doesn't block on locks, deadlocks are impossible to cause and race conditions can't occur because transactional memory will detect that another thread has made a change to a variable.

A major, and arguably insoluble, problem with transactional memory is IO, IO actions are irreversible, i.e you can't unsend a packet on a network or unprint something to the screen. Work has been done to address these problems using methods like switching to locks when IO occurs. Despite being conceptually simple these problems may make transactional memory useful only in limited circumstances or require careful design to ensure that IO happens outside a transaction.

Since transactional memory is a relatively new approach it has yet to achieve widespread use.

Futures and Reactive Programming

Futures (also called promises) are a mechanism for performing long running tasks in the background while a main process advances. A programmer creates a future that is placed into a background process that carries out the operation while the main process advances. When the value is needed the main thread blocks waiting for the future to complete. Once complete the future returns its value to the main thread, which then continues. Listing 2 shows an example of a future in use in the Java language.

The Mozart language and programming system integrates this mechanism directly into the language. A program's variables can cause the process to block waiting on their value to be set by another process. This technique is called data flow programming and is related to the concept of reactive programming [48, Ch. 4].

Although many of the concepts behind futures are relatively old they have only recently been added to mainstream programming languages like Java, and like actors and transactional memory they have only recently gotten more attention.

2.2.5 Concurrency and the Operating System

In most cases (such as desktops and servers) the number of processes exceeds the number of CPUs available, so some system for dividing up the CPU between them is needed. This job is handled by a piece of software called the scheduler, the scheduler chooses what process or thread to run next after a thread has used the CPU enough or when a process waits for something to happen, such as during IO.

```
class CallableImpl implements Callable<Integer> {
  private int myName;
  CallableImpl(int i){
    myName = i;
  }

  public Integer call() {
    for(int i = 0; i < 10; i++) {
        System.out.println("Thread : " + getMyName() + " I is : " + i);
    }
    return new Integer(getMyName());

  }

  public int getMyName() {
    return myName;
  }

  public void setMyName(int myName) {
    this.myName = myName;
  }

}

public class CallableTester {
  public static void main(String[] args) {
    Callable<Integer> callable = new CallableImpl(2);

    ExecutorService executor = new ScheduledThreadPoolExecutor(5);
    Future<Integer> future = executor.submit(callable);

    try {
        System.out.println("Future value: " + future.get());
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

Listing 2: Simple example of a future in Java [5], the line containing future.get() is where the other thread is started and the main thread blocks waiting for the value to be generated.

There are two basic approaches to scheduling: cooperative scheduling and pre-emptive scheduling. Cooperative scheduling means that processes must voluntarily relinquish the CPU, either by calling an explicit yield function or implicitly when they make an IO call. An obvious problem with this design is that if a process never relinquishes the CPU then other threads will never run, potentially leading to a process hanging.

Pre-emptive scheduling is when the operating system can "pre-empt" a running process, forcing it into a waiting state even if it hasn't explicitly yielded or performed IO. This is triggered by a timer periodically interrupting the CPU causing a piece of code to run in the operating system that checks whether the process currently running on the CPU has used up its available time called the "quantum". If the operating system decides that the process has had enough time on the CPU another process will be chosen to run on the CPU. This process is called a context switch.

Pre-emptive scheduling is used on most (if not all) desktop servers, cooperative scheduling is still used in some situations, particularly mobile phone operating systems that don't want the overhead in terms of power consumption that constantly interrupting the CPU causes.

The implementation of the scheduler can have an enormous effect on how a system performance, for instance it can effect the responsiveness of a system or the throughput; for a more extensive discussion of scheduling (and operating systems in general) see Tanenbaum [58, p. 145].

As mentioned above, threads are similar to processes in that they each have their own stack and registers and program counter. There are two major approaches to implementing threads, kernel threads and user (or green) threads. Kernel threads are threads that are part of the kernel. Generally, the thread scheduler is the same as the process scheduler in this case, and often processes and threads are no different at the kernel level.

User threads are threads that are implemented as part of a user space library so the scheduler and threads correspond to a single operating system process. Some languages (such as Ruby) use user threads to implement threading. One of the main limitations of user threads is that they can't easily use multiple CPUs, since an operating system process is scheduled on a single CPU.

Combinations of these two approaches are possible: a common strategy is to have one kernel thread with multiple "fibers" which are sort of mini-threads that are cooperatively scheduled running on top of the kernel thread.

At the user level there is usually some type of API for using threads regardless of whether they are kernel or user threads, for example the POSIX thread library for UNIX systems or Windows threads for Windows systems.

2.2.6 Debugging Concurrent Programs

Debugging concurrent programs poses some unique challenges that don't appear with debugging sequential programs. Being able to successfully debug concurrent programs is important since it is necessary to ensure software has no errors (to a reasonable standard). Debugging can also take a significant amount of time (Erlikh [35] notes that 85-90% of time is taken up with debugging) in the software development process, so rapid debugging can save time and money.

Debugging can take a number of forms, such as finding the causes of specific exceptions or errors in a program under development or finding performance problems in a system.

A great deal of research and a large number of tools have been developed to address these challenges. In this section we will discuss some of the existing approaches to addressing this challenge and some of the problems with these approaches. We will focus on debuggers and tracing frameworks that include a visual component in Section 2.4 after we have introduced visualization.

For more information about debugging see 'Debugging Parallel Systems: A State of the Art Report' [39].

Stress Testing

Stress testing, also called exhaustive testing, is an extension of existing testing practices such as unit testing. Instead of testing something once, a concurrent program is run many times to try to detect hard-to-debug concurrency errors like race conditions and deadlocks. Tools such as ConTest [55] are designed to assist this process by randomizing the scheduling of threads by inserting sleeps and pauses at random points within the code.

Unfortunately, this approach isn't an absolute guarantee that concurrency bugs will be found, since it can't guarantee that every single possible execution of the entire space of executions will be tried, which is generally quite large in multithreaded programs.

Problems with traditional cyclic debuggers

The traditional debugging process for sequential programs is often called cyclic debugging because of the way that users run a program over and over "zooming" in on the bug with each run. This is generally not feasible for debugging concurrent programs, since many concurrency bugs are non-deterministic, since their appearance depends on timing factors such as the operating system scheduler, which may not be consistent between runs.

Even worse, the presence of breakpoints in code can change the timing of the code, potentially leading to bugs disappearing when run under a debugger and reappearing when breakpoints are removed.

Trace/Event Frameworks and Replay Frameworks

For this reason most debuggers for concurrency take an approach based on tracing which is, in a sense, a more structured version of the kinds of "printf" or logging debugging that people do – outputting the state of variables and the position in code where a thread has reached such as method entries or branches. This also addresses the problems associated with stopping a thread to examine it by simply time stamping events instead. As long as there is a consistent source of time this will work well, but may be more complex in distributed computing scenarios.

Tracing (also known as monitoring) can be performed by computers equipped with special hardware such as hardware to snoop bus activity without altering the timing (avoiding something called the probe effect discussed below.) There are a number of limitations to this approach, first of all machines equipped with this hardware are uncommon as it increases the cost. Secondly, the data collected is quite low level so some processing must be done to relate it to higher level program structures. Finally, the development of multicore processors and system-on-a-chip architectures mean there often aren't buses or other hardware structures to probe, so monitoring must be integrated with the chip architecture. Having said that, there are a number of systems with these capabilities built in such as IBM Z series mainframe computers.

More commonly, tracing can be done by inserting calls to tracing methods into machine code, source code or in some cases integrated into a runtime such as Java. The inserted methods are called probes and the process of inserting probes is called instrumentation. Two examples of this kind of tracing framework are DTrace [28] developed by Sun/Oracle and IntelliTrace [11] by Microsoft.

One aspect that makes DTrace especially interesting is that it can be used to trace parts of the kernel along with user space programs, allowing deep examination of the actions and performance of a program. DTrace is also in some senses "programmable" since tracing is enabled by writing code in a scripting-like fashion, as shown in the example in Listing 3, which shows tracing of an open() system call in a UNIX system.

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}

syscall::open:return
/self->path != NULL && arg1 == -1/
{
    printf("open for %s failed", self->path);
    ustack();
}
```

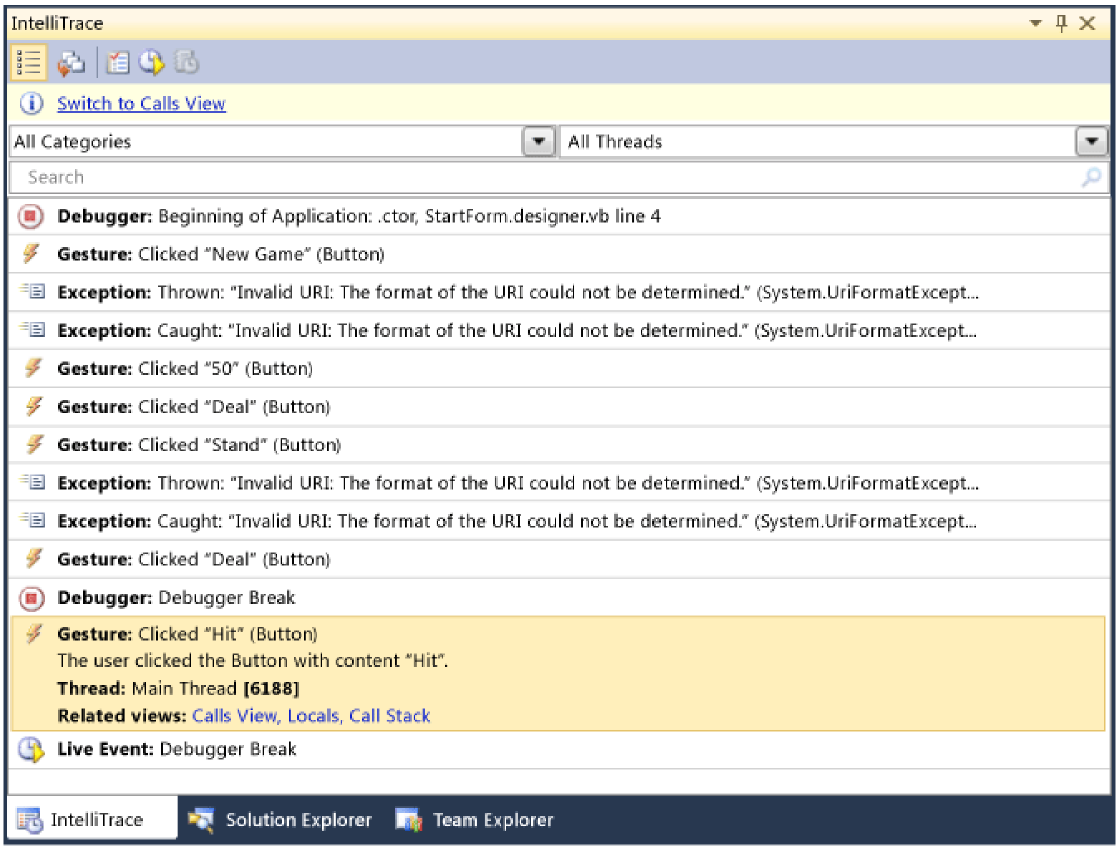
Listing 3: Probing an open() system call with DTrace [22]. The lines containing "syscall" are the events in the kernel to trace and the line beneath is a predicate which checks whether the action contained in the body of the trace will execute. The trace body is written in a programming language allowing complicated tracing actions to be performed.

This allows the user to single out parts of their program, library or runtime for examination. Programs and libraries have to be prepared to allow DTrace probes to be added.

IntelliTrace from Microsoft provides similar features except that it only operates on user space programs. The types of data recorded can be chosen from a list and recorded data can be saved in a file that allows a program trace to be shared with other developers. IntelliTrace integrates with the Visual Studio IDE and can be used with the Visual Studio Debugger.

Tracing systems for concurrent programs face a number of unique challenges. The Probe Effect is somewhat analogous to Heisenbergs' uncertainty principle, which roughly states that it is impossible to measure the momentum and direction of a particle simultaneously to a high degree of accuracy. In the case of a concurrent system this means that attempts to monitor and record events occurring in the system will take up a certain amount of time, potentially altering the timing of the program. This has the possibility of removing an error that we are looking for such as a race condition by altering the timing of events, or add an error that wouldn't otherwise be present. This is a less extreme version of the effect that breakpoints have on a program. One way of addressing this is to simply never remove the probes and make them a fundamental part of the system, though this might have serious consequences for performance. Note that this isn't a problem with hardware monitors, since they do not alter the timing of events.

In distributed systems there is another problem called the observability problem.



Debugger: Beginning of Application: .ctor, StartForm.designer.vb line 4
Gesture: Clicked "New Game" (Button)
Exception: Thrown: "Invalid URI: The format of the URI could not be determined." (System.UriFormatExcept...
Exception: Caught: "Invalid URI: The format of the URI could not be determined." (System.UriFormatExcept...
Gesture: Clicked "50" (Button)
Gesture: Clicked "Deal" (Button)
Gesture: Clicked "Stand" (Button)
Exception: Thrown: "Invalid URI: The format of the URI could not be determined." (System.UriFormatExcept...
Exception: Caught: "Invalid URI: The format of the URI could not be determined." (System.UriFormatExcept...
Gesture: Clicked "Deal" (Button)
Debugger: Debugger Break
Gesture: Clicked "Hit" (Button)
The user clicked the Button with content "Hit".
Thread: Main Thread **[6188]**
Related views: Calls View, Locals, Call Stack
Live Event: Debugger Break

Figure 2.6: Screenshot of Intellitrace [14] showing a list of the events in an executing program the system has traced, including exceptions and user interface events.

This problem is where timing delays in a system without a shared timebase may mean a node sees an incorrect ordering of events or where two different nodes see different event orders. This problem doesn't apply to multiprocessor systems.

Tracing generally produces a tremendous amount of data, so some method of summarizing of data is often used. The most common example of this is called profiling, which aims to gather statistics about the timing of events within a program. This allows a user to debug performance problems in areas such as CPU use, memory use or IO time. This kind of data is often visualized as well; we will explore this further in Section 2.4.

Another technique commonly used as part of tracing is replaying, which allows the execution of a program to be replayed after it is complete. This can be used to allow a user to debug a parallel program in the same manner as cyclic debugging because the replay framework can ensure that threads don't advance past a breakpoint in another thread. A technique called checkpointing can be used to wind the execution of a program back to an earlier point and to start again. This is also present in cyclic debuggers such as the debugger that is part of Java. More info about replay systems can be found in 'A Taxonomy of Execution Replay Systems' [32].

Tracing frameworks are often used as part model checkers that we will discuss at the end of the next section.

Formal Methods

The goal of formal methods is to verify that a program is provably correct, not simply tested a great deal, as we have discussed above. Formal techniques can be used at different stages of software development (such as at the design stage) to help reveal flaws in the system before it is constructed. In later stages the constructed system can be compared to the specification, potentially automatically.

The first step is to create an abstract model of the system to be constructed. There are numerous languages and tools for doing this starting with Hoare's Communicating Sequential Processes [38] and its descendants.

Once a model has been created there are two techniques that can be used to verify that a system is correct. The first, called theorem proving, generates mathematical proofs from axioms of the system. Often, this process needs some guidance from the user and may require some expertise to use properly. One advantage of theorem proving is that it can deal with systems that have an infinite number of states, which model checking, the next approach, has difficulty with in some situations.

Model checking basically amounts to checking every possible execution of the model.

In a system with a small number of states this is relatively easy, but with more states this can become infeasible. In particular, the interleaved nature of concurrent programs can lead to an explosion in the number of states to be checked. Several solutions to this problem have been developed, one of the first approaches was an improved representation that greatly speeded up the performance of automatic checking. Another approach is based on the observation that, although there are many sequences of states, only a few of them may be significantly different. This can greatly speed up the process of checking.

Often, the model that is checked is simply the abstract model that a programming language forms, meaning that programs written in these languages can be checked for well known concurrency bugs such as deadlocks and race conditions. To check, the system data is traced from a running program and often some mechanism of controlling the scheduling of threads is used. Two examples of this kind of system are Java Path Finder [64] from NASA and CHESS [45] from Microsoft.

Though they can prove a system correct there are many limitations to formal methods. We have mentioned a couple already – the expertise required and the state explosion problem. Other problems are that models are an abstraction of the system under construction and are limited by the level of detail that the model provides: they can only prove what is modeled. Some practical problems with model checkers are that they require modified operating systems or runtimes (like Java Path Finder) or separate languages to implement.

This is only a brief overview of formal methods, see Merz [44] for a more in depth (and mathematical) overview.

2.3 Introduction to Visualization

Every year more and more data is accumulated, but unfortunately the increase in data isn't being matched by an increased ability of humans to process that data. Visualization aims to address this problem by using the human vision system to communicate a large amount of data, often multidimensional, through the medium of the 2 dimensional space of the page or screen. Edward Tufte [63], a visualization pioneer, states that:

> *"The world is complex, dynamic, multidimensional; the paper is static, flat. How are we to represent the rich visual world of experience and measurement on mere flatland?"*

In this section we will look we will look at the different elements that visualizations are composed of and the different kinds of data that can be visualized and how they form the basis of taxonomies of visualization. In the next section we will discuss the relationship of software visualization to visualization

2.3.1 Elements of Visualization

A number of different elements can be used to visualize data, in this section we will provide a brief overview of the various elements that are commonly used. Throughout the rest of this thesis we will refer to these elements and how different areas of visualization use them and how we have used these elements in our design.

Space

As noted above, the use of space depends on the type of data that is to be visualized. For multi-dimensional data, temporal data or data from a continuous function this may be obvious and require a simple calculation to place a data point within the available space, or potentially the use of a logarithmic function to transform the data. Space includes the use of things such as points, lines, areas and volumes.

In information visualization (discussed in Section2.3.2) the use of space is determined by the designer either by their explicit choice or by using an algorithm to position the elements in space. Algorithms such as force-directed layout are used to separate nodes in a diagram of network data by viewing the connections between nodes as springs and computing the layout by minimizing the “energy” of the system.

For tree data a common diagram type is the tree map that lays out children by containing them inside their parents, as shown in Figure 2.7.

Colour

Much of what applies to space applies to colour since continuous data is relatively simple to map to a colour scale, while in information visualization it is up to the designer to determine how to map values to colours. Also consideration of things such as colour blindness, which is quite common, must be given to ensure that differences in data can be easily determined.

There are a number of different scales from where to draw colours from such as using a “cold-hot” scale that has blue at one end and red at the other or a “traffic light” scale that goes from green through yellow to red. Certain scales called linear optimal colour scales are used to ensure that the perceived distance between values is the same as the actual distance between values.

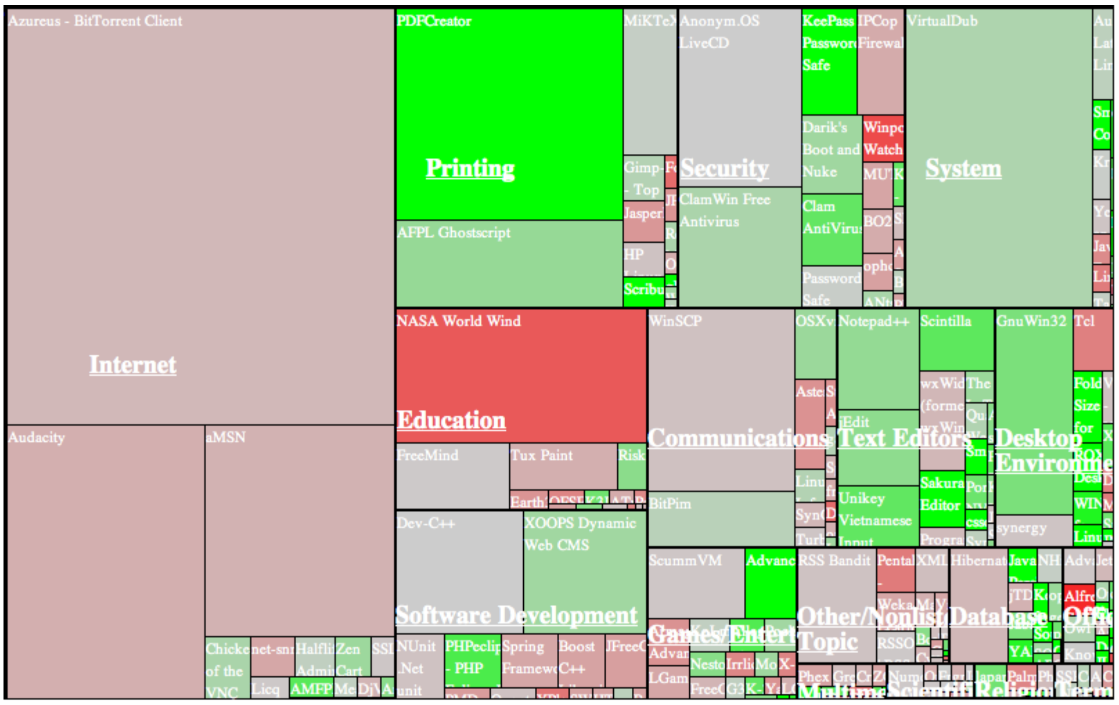


Figure 2.7: Treemap example [9] that shows the amount of downloads for software projects on sourceforge. This is an example of using space to convey amount and the relationship of the amount of one thing to another.

Text

Text doesn't play a prominent part in scientific visualizations beyond providing labels, however in many parts of information visualization text provides a valuable role. For example, tag or word cloud visualizations use text size to indicate word frequency and often use colour for additional purposes.

Another common area is in software visualization since software is primarily composed of text in the form of source code. We will explore this further in Section 2.4.

Glyphs / Icons

In information visualization a familiar use of glyphs is as markers of locations on maps such as crosses to represent church or other important locations. In a sense, glyphs are used as a simple ideographic language to conserve space by using familiar images in the place of text.

A common example from scientific visualization is the use of different icons to show data from different sets of data when using a scatterplot. This allows easy differentiation of the two (or more data sets) and helps makes things like clusters in the data clear.

Lines

Lines are probably most familiar in line graphs where they are used to show trends in data where it might be difficult to perceive an overall pattern in the raw data. This use occurs throughout scientific and information visualization, where lines are used to highlight trends, flows, and order in time. Another use, common to information visualization, is to demonstrate connection between two items such as a parent-child relationship in a tree graph or to represent the flow between events, such as in UML sequence diagrams discussed in Section 2.4.6.

Interaction

Most useful visualizations aren't simply static pictures, but change and respond to interaction from the user. Schneiderman [51] offers a useful mantra called the Information seeking mantra:

- **Overview** Gain an overview of the entire collection.
- **Zoom** Zoom in on items of interest
- **Filter** Filter out uninteresting items.

- **Details-on-demand** Select an item or group and get details when needed.
- **Relate** View relationships among items.
- **History** Keep a history of actions to support undo, replay, and progressive refinement.
- **Extract** Allow extraction of sub-collections and of the query parameters.

A similar principle is called focus+context [29] Diehl [34] explains:

> *"A detailed visualization of some part of the information the focus - is embedded within a visualization of the context, i.e. more coarse-grained information about parts related to the focus. Thus focus+context techniques provide both an overview and detail at the same time."*

Tufte [63] also brings up a similar principle in his book 'Envisioning Information' with his discussion of a visualization having micro/macro views where broad patterns can be observed while still maintaing the visibility of individual data items. In a sense, this is focus+context in a single view, since Tufte was focussing on static views.

In practice, this means that a visualization system may not simply be a single view, but may have multiple subviews of a main view, or even several different visualizations to represent different levels of the data. In addition, there may be controls to carry out the task of filtering or querying data.

2.3.2 Visualization Data

Although it might seem that categorizing visualizations on the basis of what they look like is the obvious choice, most taxonomies of visualization have been based on the types of data visualized, since the type of data generally determines the range of options for visualizing the data.

One of the first attempts to produce a taxonomy of visualization was by Shneiderman in the paper 'The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations' [51]. Schneiderman divides visualization up based on the following categories of data:

- **1-D Linear** - Data such as text documents, source code and lists of text that have a sequential order, but are non-numeric.

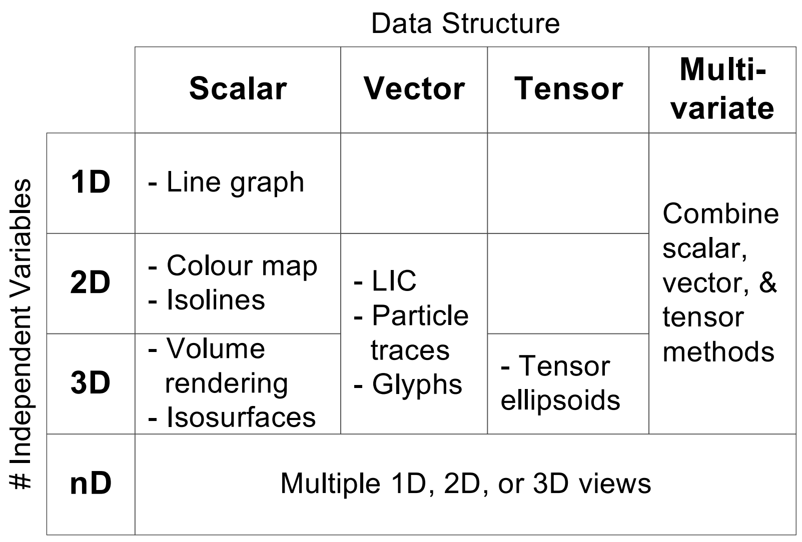
- **2-D Map** - Data is things like map data, including geographic maps for GIS systems, or 2-D diagrams and layouts that correspond to space in the real world.
- **3-D World** - Data from real-world objects such as molecules, the human body, and buildings have items with volume and some potentially complex relationship with other items. Some examples are CAD, Medical, Architecture, Molecule visualization.
- **Multi-Dimensional** - Data that is tabular in a sense, including statistical data and data from relational databases, and diagrams such as time-series and scatter plots.
- **Temporal** - Data from things such as timelines or Gantt charts. Shneiderman separates this from the above categories by saying "The distinction in temporal data is that items have a start and finish time and that items may overlap".
- **Tree** - Data from things like file systems where items have a hierarchical relationship to each other, with each item having a single parent and potentially many child elements.
- **Network** - Data that corresponds to the mathematical concept of graphs, and can be seen in things such as social network data or the links in web pages.

Tory and Mller [61] provide a slightly different taxonomy based on whether the underlying data is continuous , (drawn from a continuous mathematical function), or discrete (such as series of names or lines of source code). The following diagrams (Figures 2.8 and 2.9) show the major divisions and some subtypes of data in Tory and Mller's taxonomy.

In both of these examples of taxonomies we can see the broad lines of the two branches of visualization: scientific visualization and information visualization. Chen [31] offers the following definition to differentiate them:

> *"A key point to differentiate information visualization from data visualization and scientific visualization is down to the presence or absence of data in quantitative forms and how easy one can transform them to quantitative forms. This is why researchers emphasize the ability to represent non-visual data in information visualization."*

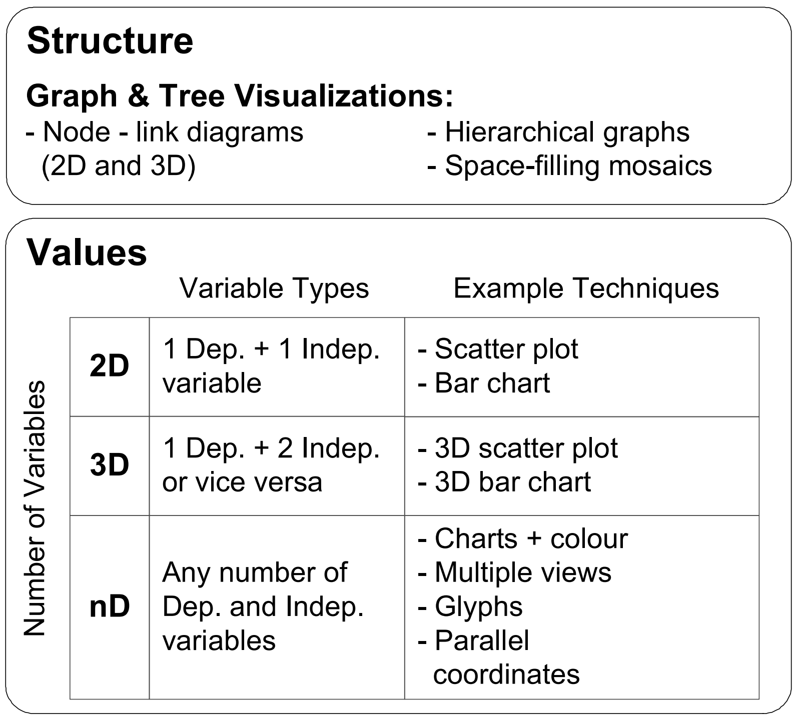
The most well-known example of scientific visualization is probably the time series, a simple plot of some independent variable against time. Many other types of diagram



Data Structure

# Independent Variables	Scalar	Vector	Tensor	Multi-variate
1D	- Line graph			Combine scalar, vector, & tensor methods
2D	- Colour map - Isolines	- LIC - Particle traces - Glyphs		
3D	- Volume rendering - Isosurfaces		- Tensor ellipsoids	
nD	Multiple 1D, 2D, or 3D views			

Figure 2.8: Examples of data and visualization types for continuous data from Tory and Mller's taxonomy [61].



Structure

Graph & Tree Visualizations:

- Node - link diagrams (2D and 3D)
- Hierarchical graphs
- Space-filling mosaics

Values

Number of Variables	Variable Types	Example Techniques
2D	1 Dep. + 1 Indep. variable	- Scatter plot - Bar chart
3D	1 Dep. + 2 Indep. or vice versa	- 3D scatter plot - 3D bar chart
nD	Any number of Dep. and Indep. variables	- Charts + colour - Multiple views - Glyphs - Parallel coordinates

Figure 2.9: Examples of data and visualization types for discrete data from Tory and Mller's taxonomy [61].

exist, however almost any quantitative data can be translated directly into a spatial representation. This kind of data would roughly correspond to the multidimensional data in Schneiderman's taxonomy and to the continuous data in Tory and Mller's taxonomy.

In contrast to scientific visualization the data in information visualization is non-quantitative and doesn't have a natural mapping to space or things such as colour. To design a mapping to an arbitrary point in space is the task of information visualization. Some examples of data of this type could be data in the form of a tree, a graph or even a corpus of text. This kind of data corresponds to 1D, 2D, 3D, Temporal, Tree and Network data in Schneiderman's taxonomy and discrete

Visual Analytics [60] is a new member of the visualization world and is as concerned with cognition as it is with the visual aspects:

> *"The panel defined visual analytics as the science of analytical reasoning facilitated by interactive visual inter-faces. People use visual analytics tools and techniques to synthesize information and derive insight from massive, dynamic, ambiguous, and often conflicting data; detect the expected and discover the unexpected; provide time-ly, defensible, and understandable assessments; and communicate assessment effectively for action."*

We won't say anymore about this since it doesn't seem to have an impact on the field of software visualization (yet).

Providing a comprehensive overview of such a vast field is difficult, for a more detailed account see the books 'Visualization Handbook' [40] for scientific visualization, and 'The Craft of Information Visualization: Readings and Reflections' [52] for information visualization and Chen's paper 'Information Visualization' [31], and for visual analytics see 'An Agenda for Visual Analytics' [60]. For more about the difference between scientific visualization and information visualization see the Tory and Mller taxonomy paper [61].

2.4 Software Visualization

Diehl [34] defines software visualization as "the visualization of artifacts related to software and its development process" - this can include things such as source code commits, bug reports, executions of the software and any data the process of software development generates. This is a diverse range of data including quantitative data such as the timing of events within a trace of a running program, non-quantitive data such as the names of people committing software patches to a project or even a network or tree of the relationships of objects in an object-oriented program. As such, software visualization is neither a branch of scientific visualization or information visualization, but draws on both of them, often in the same visualization or software system.

In his book 'Software Visualization' [34] Diehl identifies three broad areas of research into software visualization: Structure is the static parts of a system, which can be analyzed without running the program, such as the class inheritance and composition, while behaviour is data generated from the execution of a system (often generated by tracing), and evolution is how a system changes over time through things such as commits to a source control system.

As mentioned in our introduction, our goal was to investigate how software tools could be used to help software developers write concurrent software. The structure of a program can be an important part of this, however we felt this had already been addressed reasonably successfully by techniques such as UML class diagrams. The evolution of software may be important for understanding how certain bugs have occurred, but it is not as useful for the day-to-day debugging of software. Program behaviour visualization has a close relationship to the existing debugging technique of tracing, because to analyze a program it is often necessary to gather trace data from it's execution.

Also, these were the types of software that seemed similar to music notation and software when we began searching the literature. There are many visualizations in this area, so in the rest of this section we will investigate a representative sample of the work and provide details about how they work and their goals. Finally, we will make some summary remarks about the systems we have investigated.

2.4.1 Visual VM

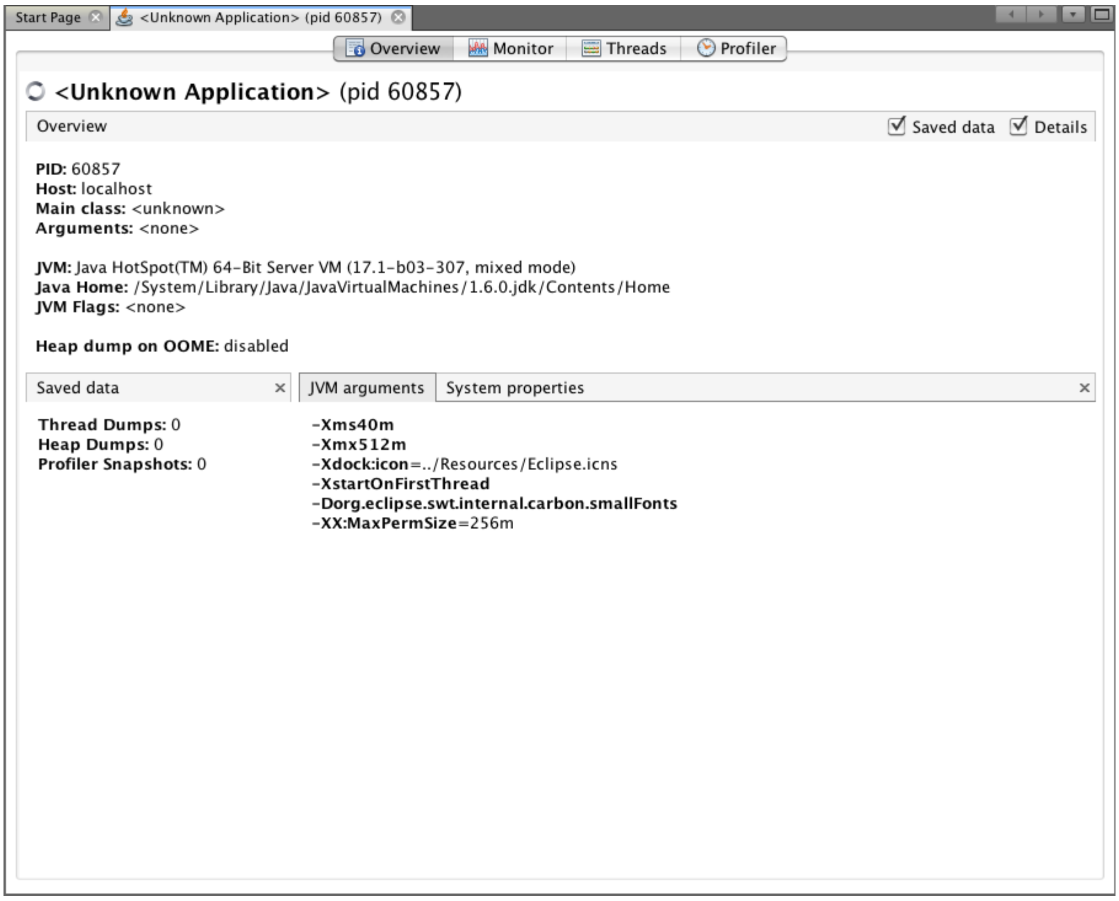
VisualVM [24] is a tool by Sun/Oracle to analyze the performance of Java programs running on a JVM. Trace data is gathered using the Java Virtual Machine Tool Interface [17] (JVMTI) which is built into the JVM. VisualVM has multiple views, only some of which are visualizations. The first view is the overall view, which provides basic information about a process as shown in Figure 2.10.

The monitor view provides various graphs for different parts of the program memory and classes, shown in Figure 2.11.

The thread timeline view provides an overview of the a thread state in time, colour is used to indicate the state of a thread, shown in Figure 2.12.

The sampler view allows quick profiling of a running application as is shown in Figure 2.13.

Finally, the Monitor tool allows more complete profiling of the CPU or memory of



<Unknown Application> (pid 60857)

Overview ☑ Saved data ☑ Details

PID: 60857
Host: localhost
Main class: <unknown>
Arguments: <none>

JVM: Java HotSpot(TM) 64-Bit Server VM (17.1-b03-307, mixed mode)
Java Home: /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
JVM Flags: <none>

Heap dump on OOME: disabled

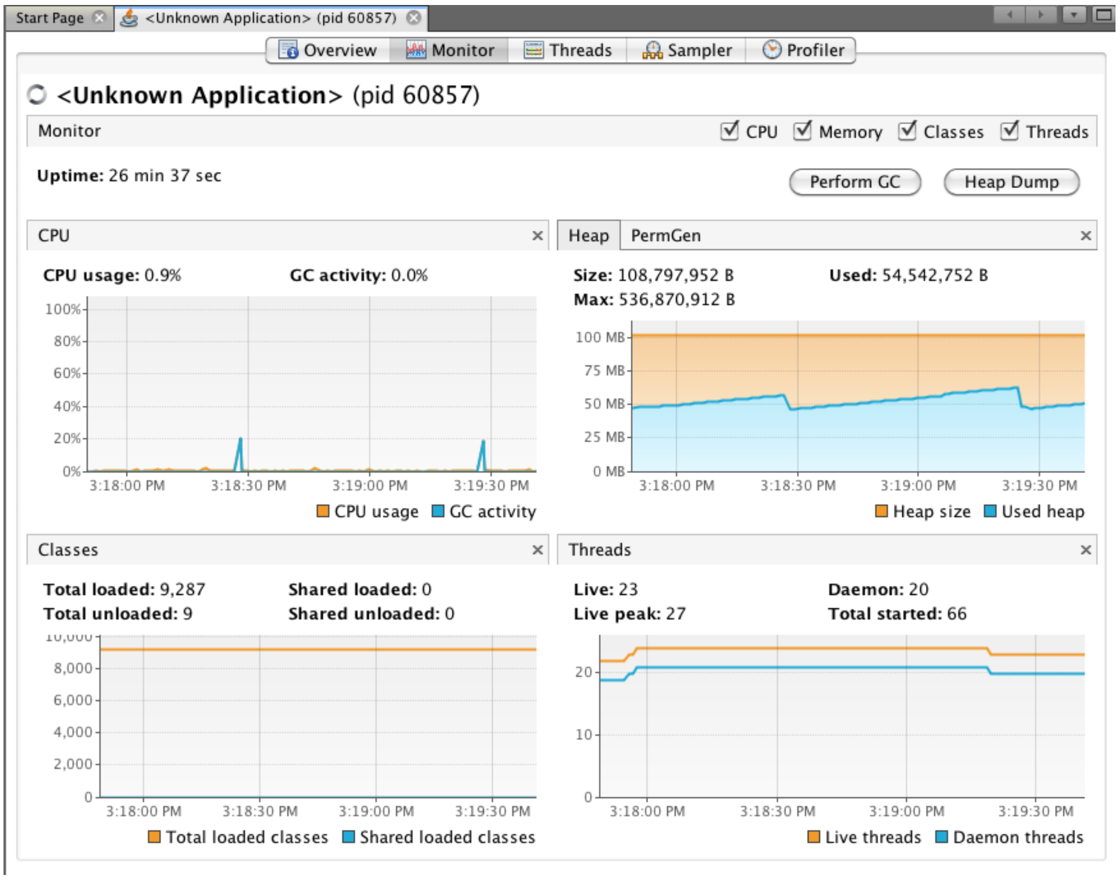
Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments | System properties

```
-Xms40m
-Xmx512m
-Xdock:icon=../Resources/Eclipse.icns
-XstartOnFirstThread
-Dorg.eclipse.swt.internal.carbon.smallFonts
-XX:MaxPermSize=256m
```

Figure 2.10: Process overview for VisualVM [24] showing some basic bits of information about a process including the execution arguments and the process id.



<Unknown Application> (pid 60857)

Uptime: 26 min 37 sec

CPU usage: 0.9%
GC activity: 0.0%

Size: 108,797,952 B
Used: 54,542,752 B
Max: 536,870,912 B

Total loaded: 9,287
Shared loaded: 0
Total unloaded: 9
Shared unloaded: 0

Live: 23
Daemon: 20
Live peak: 27
Total started: 66

Figure 2.11: Monitor view of VisualVM [24] that shows various graphs showing different aspects of a programs performance. The top left shows the CPU and Garbage collection activity. The top right shows the memory usage and the type of memory being used. The bottom left shows information about classes and the bottom right shows information about threads.

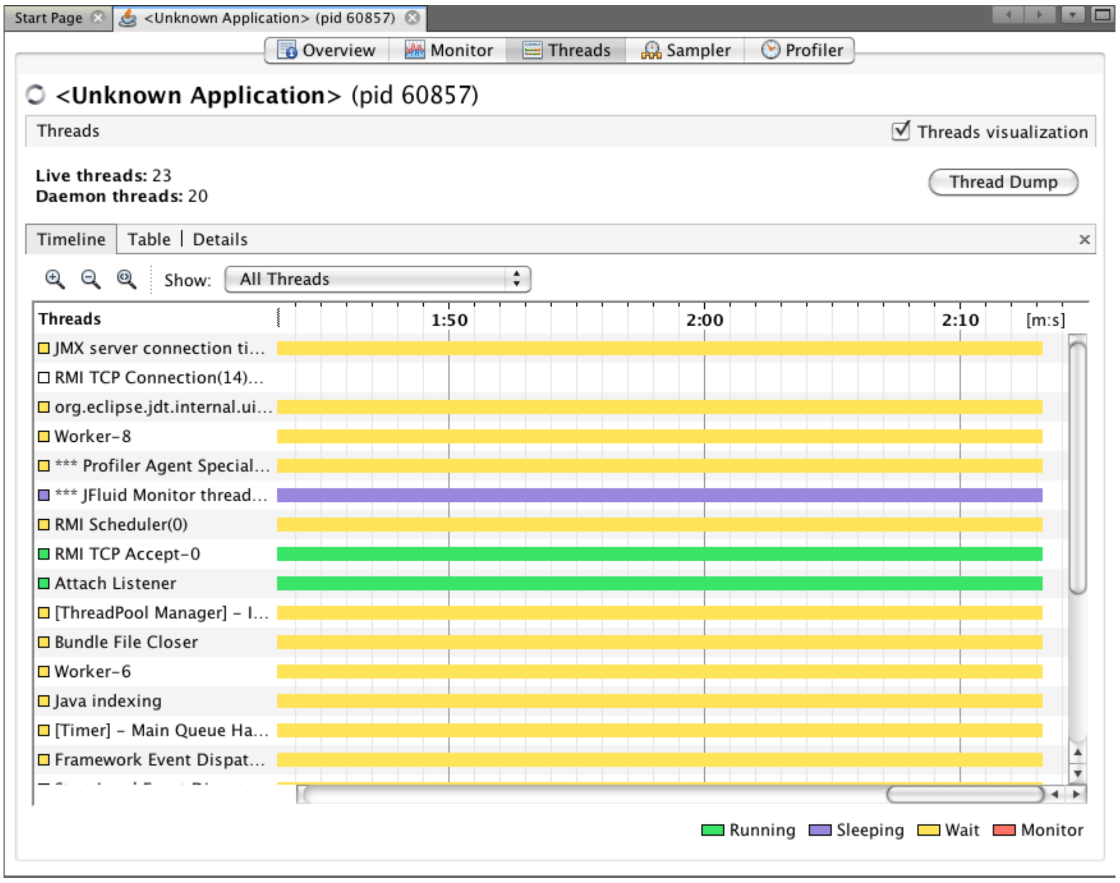
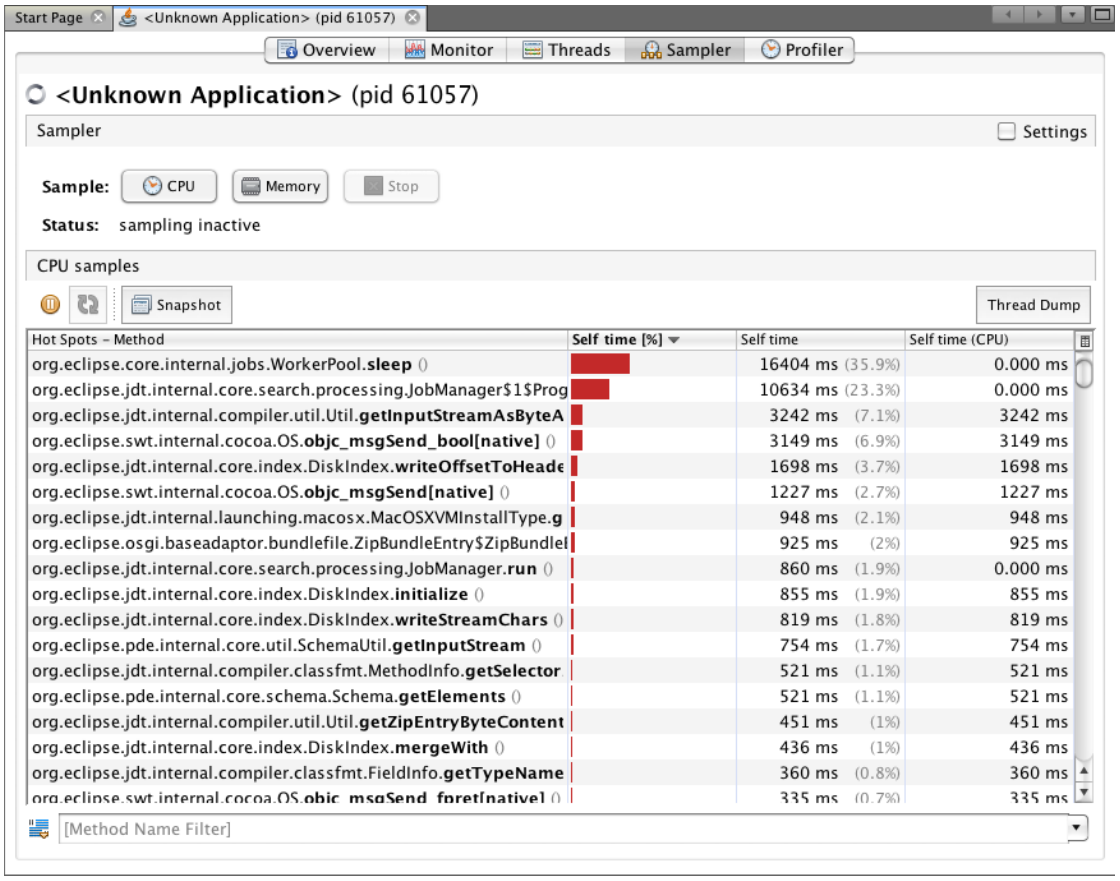


Figure 2.12: Thread timeline view of VisualVM [24]. The timeline shows the thread state over time.



<Unknown Application> (pid 61057)

Sampler

Sample:

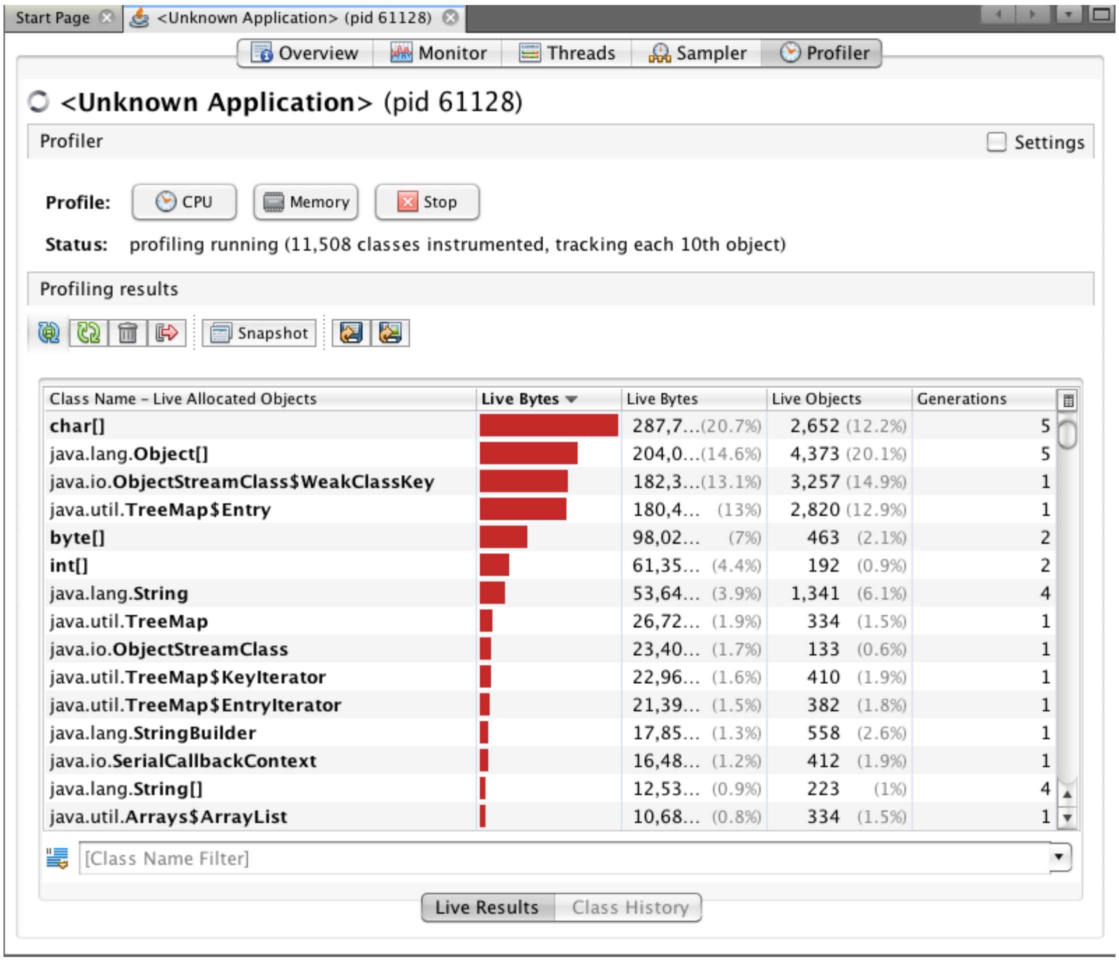
Status: sampling inactive

CPU samples

Hot Spots – Method	Self time [%]	Self time	Self time (CPU)
org.eclipse.core.internal.jobs.WorkerPool.**sleep** ()	🟥	16404 ms (35.9%)	0.000 ms
org.eclipse.jdt.internal.core.search.processing.JobManager$1$Prog	🟥	10634 ms (23.3%)	0.000 ms
org.eclipse.jdt.internal.compiler.util.Util.**getInputStreamAsByteA**	🟥	3242 ms (7.1%)	3242 ms
org.eclipse.swt.internal.cocoa.OS.**objc_msgSend_bool[native]** ()	🟥	3149 ms (6.9%)	3149 ms
org.eclipse.jdt.internal.core.index.DiskIndex.**writeOffsetToHeade**	🟥	1698 ms (3.7%)	1698 ms
org.eclipse.swt.internal.cocoa.OS.**objc_msgSend[native]** ()	🟥	1227 ms (2.7%)	1227 ms
org.eclipse.jdt.internal.launching.macosx.MacOSXVMInstallType.**g**	🟥	948 ms (2.1%)	948 ms
org.eclipse.osgi.baseadaptor.bundlefile.ZipBundleEntry$ZipBundleE	🟥	925 ms (2%)	925 ms
org.eclipse.jdt.internal.core.search.processing.JobManager.**run** ()	🟥	860 ms (1.9%)	0.000 ms
org.eclipse.jdt.internal.core.index.DiskIndex.**initialize** ()	🟥	855 ms (1.9%)	855 ms
org.eclipse.jdt.internal.core.index.DiskIndex.**writeStreamChars** ()	🟥	819 ms (1.8%)	819 ms
org.eclipse.pde.internal.core.util.SchemaUtil.**getInputStream** ()	🟥	754 ms (1.7%)	754 ms
org.eclipse.jdt.internal.compiler.classfmt.MethodInfo.**getSelector**	🟥	521 ms (1.1%)	521 ms
org.eclipse.pde.internal.core.schema.Schema.**getElements** ()	🟥	521 ms (1.1%)	521 ms
org.eclipse.jdt.internal.compiler.util.Util.**getZipEntryByteContent**	🟥	451 ms (1%)	451 ms
org.eclipse.jdt.internal.core.index.DiskIndex.**mergeWith** ()	🟥	436 ms (1%)	436 ms
org.eclipse.jdt.internal.compiler.classfmt.FieldInfo.**getTypeName**	🟥	360 ms (0.8%)	360 ms
org.eclipse.swt.internal.cocoa.OS.**objc_msgSend_fpret[native]** ()	🟥	335 ms (0.7%)	335 ms

Figure 2.13: Sampler view of VisualVM [24].

the target application as is shown in Figure 2.14.



<Unknown Application> (pid 61128)

Profiler

Profile:

Status: profiling running (11,508 classes instrumented, tracking each 10th object)

Profiling results

Class Name – Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Generations
char[]		287,7...(20.7%)	2,652 (12.2%)	5
java.lang.Object[]		204,0...(14.6%)	4,373 (20.1%)	5
java.io.ObjectStreamClass$WeakClassKey		182,3...(13.1%)	3,257 (14.9%)	1
java.util.TreeMap$Entry		180,4... (13%)	2,820 (12.9%)	1
byte[]		98,02... (7%)	463 (2.1%)	2
int[]		61,35... (4.4%)	192 (0.9%)	2
java.lang.String		53,64... (3.9%)	1,341 (6.1%)	4
java.util.TreeMap		26,72... (1.9%)	334 (1.5%)	1
java.io.ObjectStreamClass		23,40... (1.7%)	133 (0.6%)	1
java.util.TreeMap$KeyIterator		22,96... (1.6%)	410 (1.9%)	1
java.util.TreeMap$EntryIterator		21,39... (1.5%)	382 (1.8%)	1
java.lang.StringBuilder		17,85... (1.3%)	558 (2.6%)	1
java.io.SerialCallbackContext		16,48... (1.2%)	412 (1.9%)	1
java.lang.String[]		12,53... (0.9%)	223 (1%)	4
java.util.Arrays$ArrayList		10,68... (0.8%)	334 (1.5%)	1

Figure 2.14: Profiler view of VisualVM [24] showing how long methods are taking to execute.

VisualVM is interesting to us because it is not an academic product, but a tool aimed at everyday programmers, like many of the systems we looked at it is designed for profiling and performance debugging.

2.4.2 PARADE

PARADE [54, 66] is both a visualization system for concurrency and an environment for creating visualizations of concurrent programs. The software is aimed at both programming comprehension and debugging performance and program errors. Data is gathered by macros that implement POSIX threads (pthreads) library calls by tracing the method calls before calling the underlying pthread method.

The visualization system is composed of a number of separate views. First is the threads overview (Figure 2.15), which provides a list of all the threads in the system at the current time within an execution. A unique colour is given to each thread to help differentiate them, also the amount of colouring in the thread box so that when the box is completely filled with colour the thread is running and when half-filled the thread is waiting or blocked.

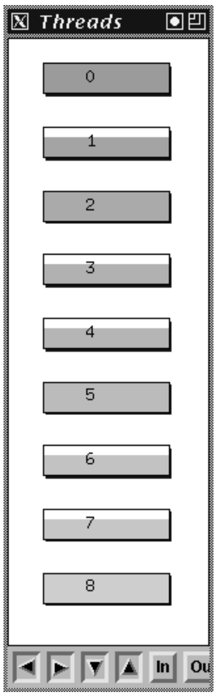


Figure 2.15: Threads view of Parade [54, 66] showing a list of threads as boxes. Colour is used to show state.

The function view (Figure 2.16) displays the static call structure of a program at a point in time using a graph that displays each function as a rectangle and the call structure as lines running between the boxes. Function boxes are given a unique colour to identify them. The position of a thread within the call structure is shown as a small circle (coloured the same as the thread overview) that moves between the different functions of the call graph.

The history view (Figure 2.17) is a timeline of each thread and the functions it has called. Each bar represents a single thread, with the drop shadow of the bar being the same as the colour in the thread overview. The bar itself is divided into sections

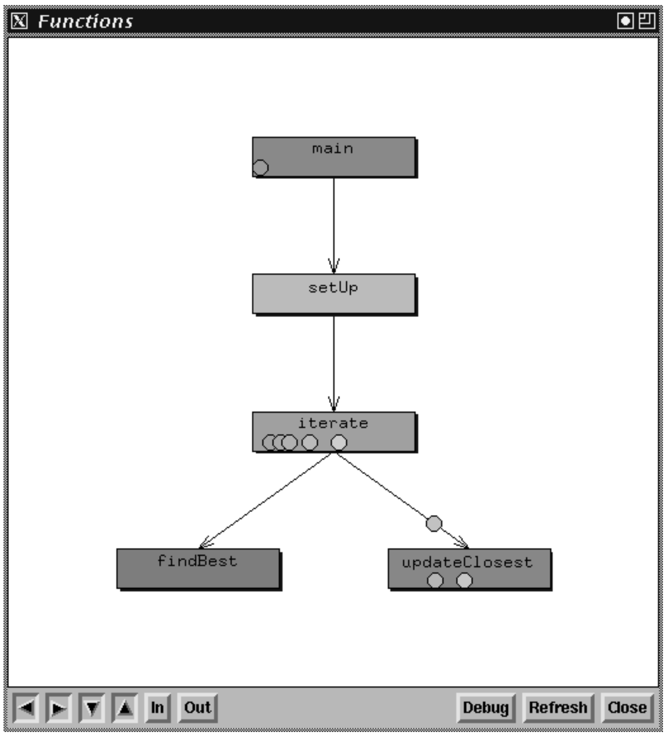


Figure 2.16: Functions view of Parade [54, 66] shows a trace of what functions have executed for a single thread.

indicating the functions called by the colour, which is the same as the colour in the function view. Small triangles within the bars represent calls and returns.

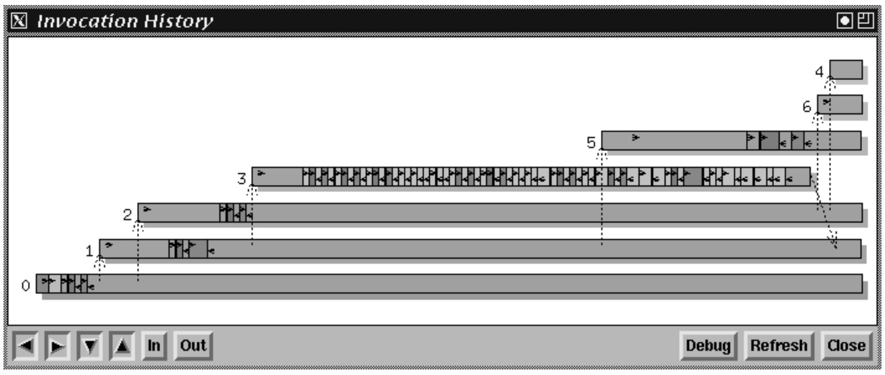


Figure 2.17: History view of Parade [54, 66] showing the timeline of thread execution, somewhat like the Visual VM example.

The mutex view (Figure 2.18) shows which threads are trying to lock a mutex as a circle surrounded by smaller filled circles that represent the threads that are trying to lock. When a thread succeeds in locking a mutex the smaller circle representing the thread moves inside the larger circle indicating that it has been locked.

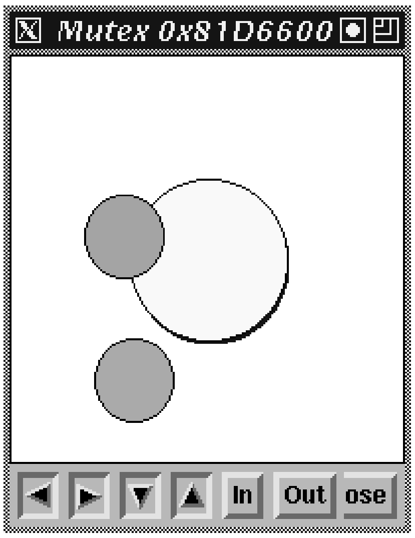


Figure 2.18: Mutex view of Parade [54, 66]. The small coloured circles are threads that move inside the larger clear circle that represent the Mutex when they acquire it.

The barrier view (Figure 2.19) is used to show which threads have entered and exited a barrier synchronization point. A new row of boxes are added as each barrier synchronization is complete.

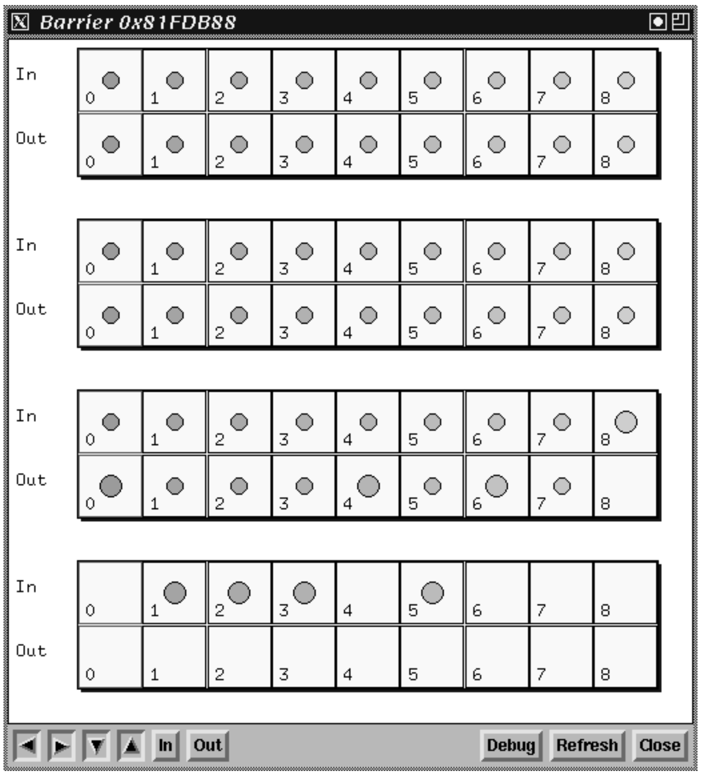


Figure 2.19: Barrier view of PARADE [54, 66] showing what threads have entered and exited a barrier synchronization point.

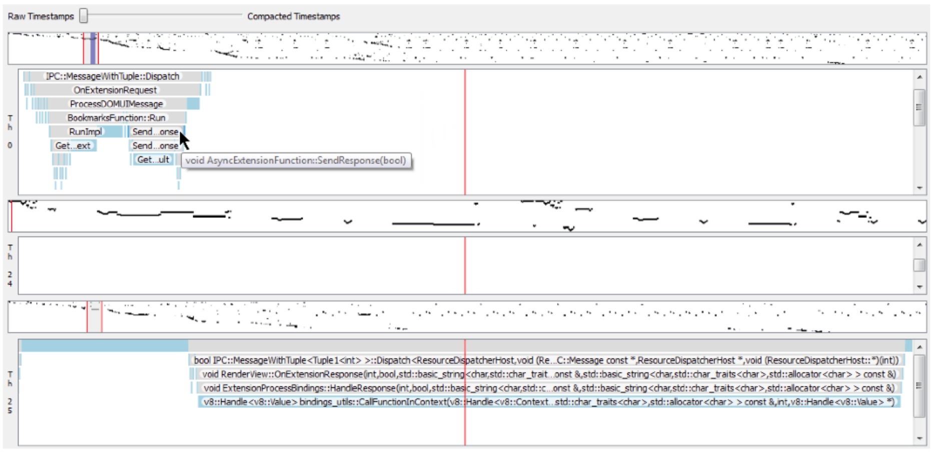
PARADE is of particular interest because it seems to not be as orientated towards performance debugging as software such as VisualVM is.

2.4.3 Understanding Complex Multithreaded Software Systems by Using Trace Visualization

In their paper 'Understanding Complex Multithreaded Software Systems by Using Trace Visualization' Trumper et al. [62] describe a system for visualizing the threads of a program. Like PARADE their software is made up of a series of separate views, however it is slightly more integrated and cohesive.

The system uses a two part tracing mechanism, where an overview trace is used to trace the entire system before the user selects what details they want traced.

Like PARADE, the system has a thread list that provides an overview of threads in the system and some statistics about the threads. The main view is a series of timeline views for each thread in the system, similar to the timeline view in VisualVM or the history view in PARADE. Unlike these two systems, a zoomed-out overview of the threads entire execution is provided above a view of a section of a threads execution, as shown in Figure 2.20. The zoomed-in view shows the call stack as a series of bars representing method or function calls placed below the main thread function call.



```
bool IPC::MessageWithTuple<Tuple1<int> >::Dispatch<ResourceDispatcherHost,void (Re...C::Message const *,ResourceDispatcherHost *,void (ResourceDispatcherHost::*)(int))
void RenderView::OnExtensionResponse(int,bool,std::basic_string<char,std::char_trait...onst &,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &)
void ExtensionProcessBindings::HandleResponse(int,bool,std::basic_string<char,std::c...onst &,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &)
v8::Handle<v8::Value> bindings_utils::CallFunctionInContext(v8::Handle<v8::Context...std::char_traits<char>,std::allocator<char> > const &,int,v8::Handle<v8::Value> *)
```

Figure 2.20: Trace Visualization showing two threads [62]. Each thread ”lane” has an overview (top) and zoom in panel on the bottom that show what functions have been executing.

Because threads can have a massive variation in method call time, because of things like thread sleeping or waiting, visualizations might contain large amounts of space where nothing is happening. To address this the developers use logarithmic time scaling, which means in practice large times are shrunk and small periods of time remain the same. Panning, zooming or changing the time compaction of a thread is applied to all other threads to keep the separate thread views synchronized to avoid the problems of having multiple thread views where events are no longer in their execution order.

This software provides a number of useful additions, including the use of time compaction to make visualizing threads much easier, the thread overview panel (which provides the focus+context that makes it easy to zoom in on a section of a thread) and an interesting method of showing the current call stack. The system seems to be orientated to analyzing the performance of programs, as the evaluation examples given show.

2.4.4 Zinsight

Zinsight [33] is a trace visualization tool for the IBM system Z mainframe series of computers. Data gathering is performed using special hardware specific to the Z series mainframe.

Like the systems mentioned above, Zinsight uses multiple views to visualize the generated data. The first view is called the event trace (Figure 2.21) and is a timeline view similar to the ones mentioned above, except that it flows vertically rather than horizontally. Processes are spaced horizontally across the view and within the process events are represented as blocks and are also spaced horizontally. There are various colouring schemes available for events, a common one is to colour event blocks based on the module they are from, such as from a database module. Interaction is provided by zooming and panning.

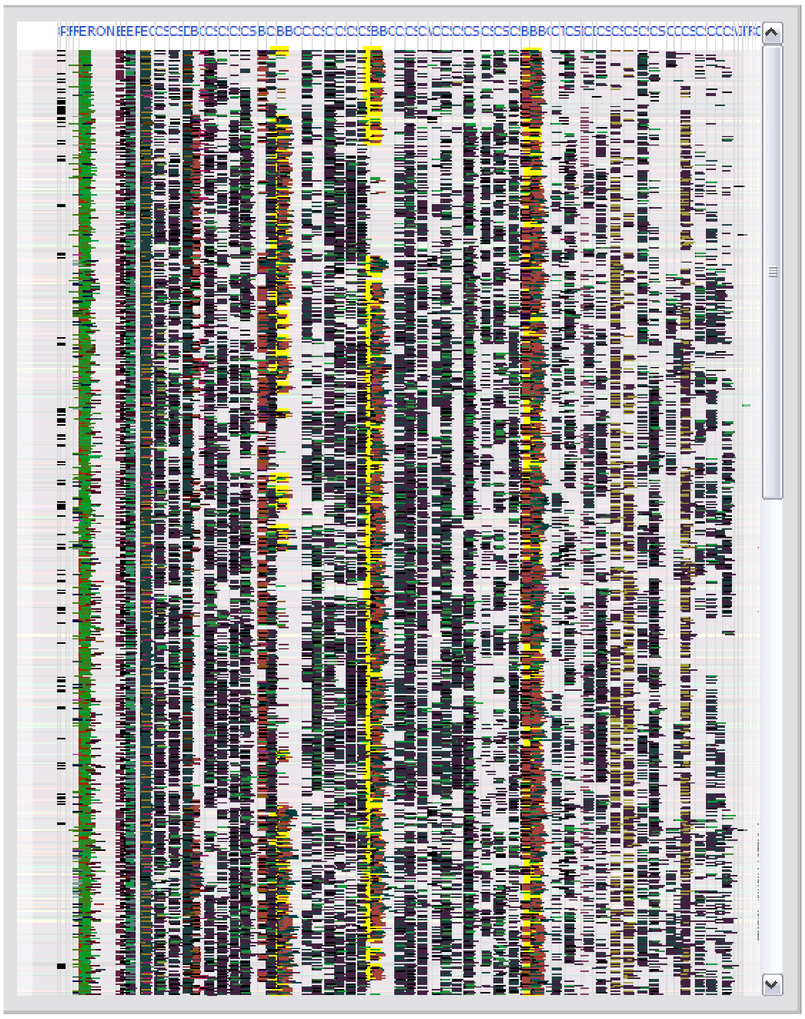
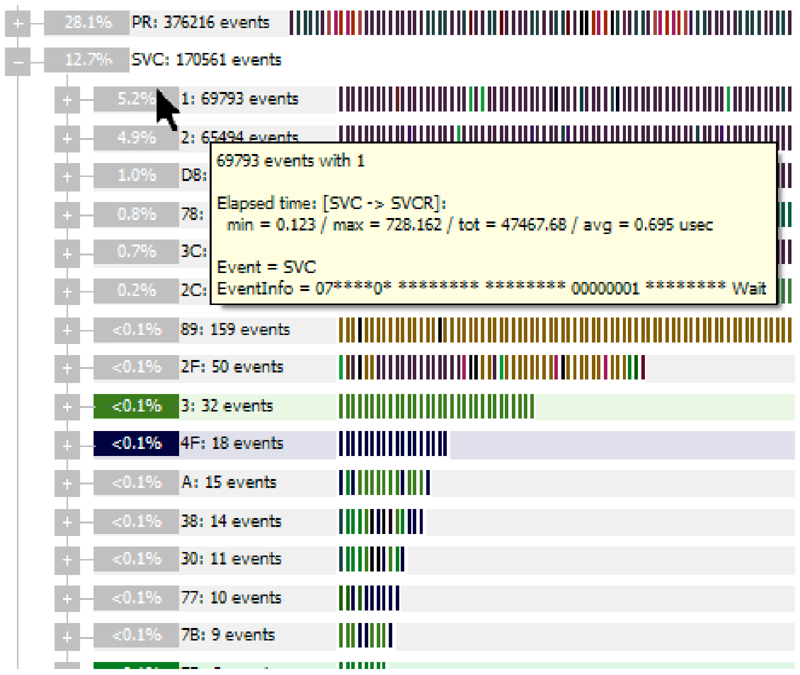


Figure 2.21: Event view of Zinsight [33]. Note that time flows vertically rather than horizontally like the previous examples.

The next view is called the event statistics view (Figure 2.22) which provides some statistical information subviews similar to those that VisualVM provides. The first subview is the event type view, which shows events by their module and the amount of time they take up. The pathology view is a semi-automatic view designed to draw attention to potentially interesting events that might indicate performance problems. Finally, the location view divides up events by process and thread to provide more specific information about the events that processes perform.



28.1% PR: 376216 events
12.7% SVC: 170561 events
5.2% 1: 69793 events
4.9% 2: 65494 events
1.0% D8:
0.8% 78:
0.7% 3C:
0.2% 2C:
<0.1% 89: 159 events
<0.1% 2F: 50 events
<0.1% 3: 32 events
<0.1% 4F: 18 events
<0.1% A: 15 events
<0.1% 38: 14 events
<0.1% 30: 11 events
<0.1% 77: 10 events
<0.1% 7B: 9 events

```
69793 events with 1

Elapsed time: [SVC -> SVCR]:
  min = 0.123 / max = 728.162 / tot = 47467.68 / avg = 0.695 usec

Event = SVC
EventInfo = 07*****0* ******** ******** 00000001 ******** Wait
```

Figure 2.22: Statistics view of Zinsight [33]. Some what like the profiling view of Visual VM this provides statistical information about the running program.

The final view is called the sequence context view (Figure 2.23) and looks somewhat similar to the function view in PARADE, showing a call stack of the program. However, Zinsight doesn't display a single point in time, but instead summarizes similar event sequences leading up to some event. This is designed to answer questions like 'how do we get to a certain event in a program such as a lock call?' and 'what are the common traces leading up to a certain event?'. A count on the lines leading from one event to another indicates the number of times that that execution path was taken.

Like the systems mentioned above, Zinsight seems to be orientated towards performance debugging and it was evaluated as such. The use of summarizing multiple calls to events into a single view is very interesting.

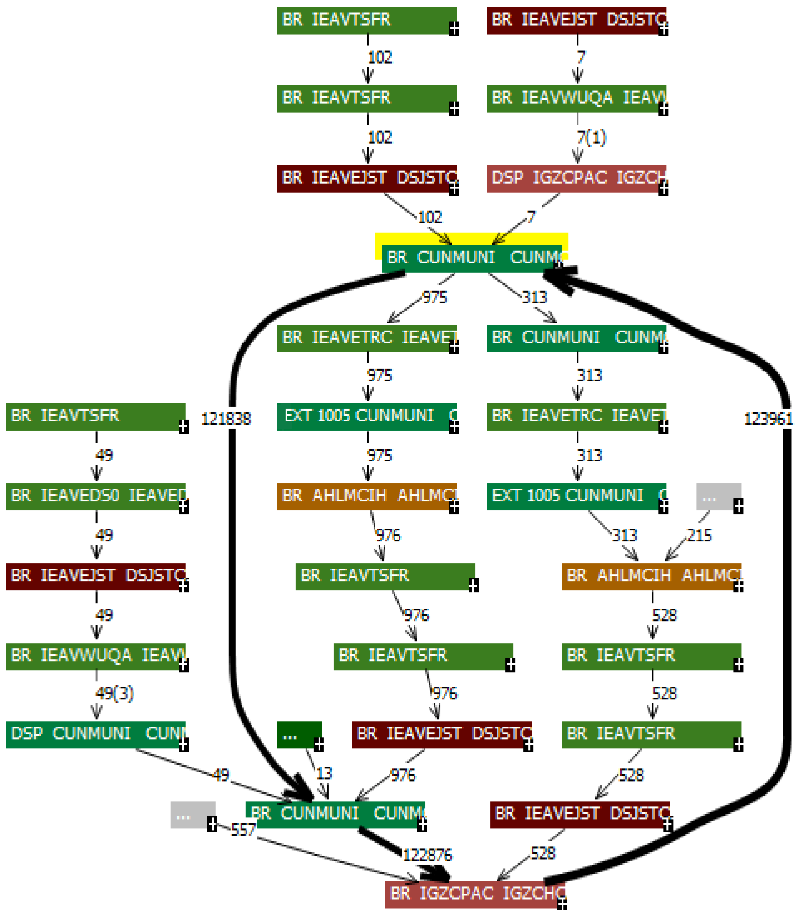
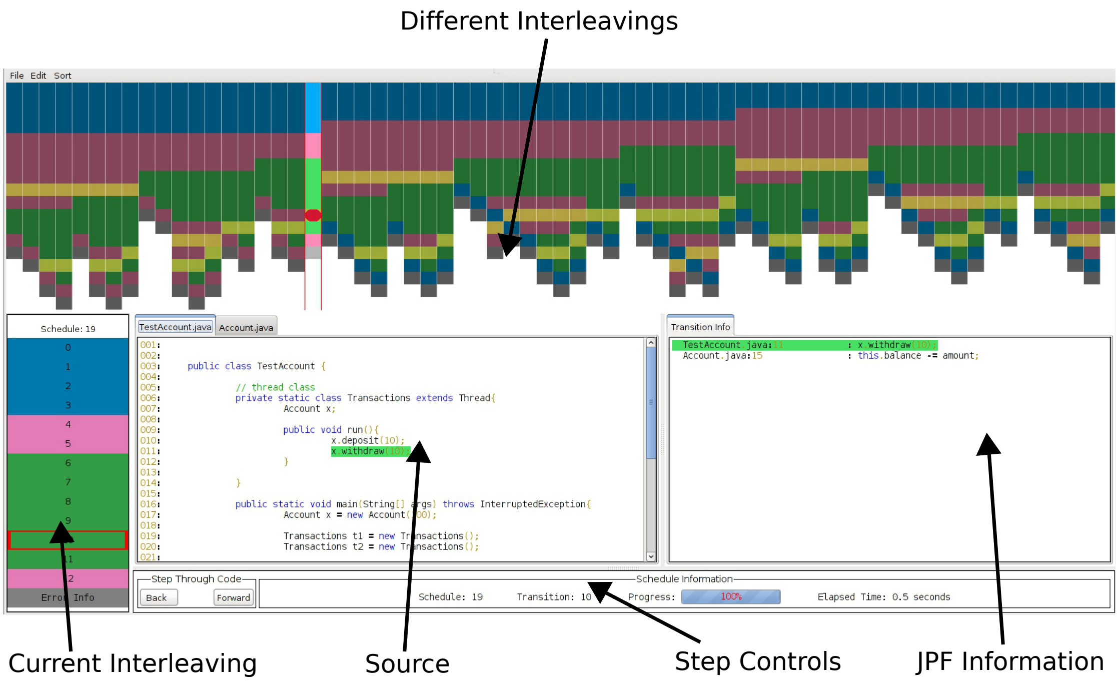


Figure 2.23: Sequence Context view of Zinsight [33]. This is showing a summarization of the various execution paths the program has gone through.

2.4.5 TIE: Thread Interleaving Explorer

Unlike the systems above Thread Interleaving Explorer [42] doesn't use data from a tracing framework as input, instead it uses data from the Java Pathfinder model checking tool mentioned above.

Like the software above, TIE is composed of multiple views. At the top is a panel showing inter-leavings of the software with errors, which allows selection of a specific interleaving for closer examination. A larger view of a particular interleaving is provided, allowing the user to click on a specific transition in an execution schedule to examine the source code and raw data from JPF at that point. A panel on the bottom provides the means to step through the erroneous interleaving.



```
    public class TestAccount {

            // thread class
            private static class Transactions extends Thread{
                    Account x;

                    public void run(){
                            x.deposit(10);
                            x.withdraw(10);
                    }

            }

            public static void main(String[] args) throws InterruptedException{
                    Account x = new Account(100);

                    Transactions t1 = new Transactions();
                    Transactions t2 = new Transactions();
```

```
TestAccount.java:11          : x.withdraw(10);
Account.java:15              : this.balance -= amount;
```

```
Schedule: 19     Transition: 10          Elapsed Time: 0.5 seconds
```

Figure 2.24: Thread Interleaving Explorer software [42]. Note the numerous different panels used to provide enable a user to zoom in from the more general information at the top to more specific information in the bottom columns.

TIE is interesting because it shows another possible use for concurrent visualization – not as a means of displaying a single execution, but as a means of showing multiple incorrect potential executions.

2.4.6 UML Sequence Diagrams

UML [49, 26] is described by its creators as "a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system". UML has many different diagrams for different parts of software and different stages of system development. In this thesis we are most interested in debugging the behaviour of programs, so we focused on the UML diagrams specifically for analyzing the behaviour of a program, which included state machine diagrams, activity diagrams and communication and sequence diagrams.

State machine diagrams are similar to state machine diagrams from engineering and indicate the state of single object and transitions to and from its various states. This was too low level for analyzing the behaviour of a large system with many interacting objects. An activity diagram contains nodes representing activities, with lines running between the various activities, choices are represented by diamonds with several lines of control leaving the diamond. Concurrency is represented by thick horizontal lines with several lines of control leading off them. Generally, activity diagrams were too high level for our purposes, although they did include support for concurrency. Communication diagrams show the flow of communication among instances of objects by numbering lines leading between objects represented by boxes, as shown in Figure 2.25. Communication diagrams are as much about the static structure of communicating objects as they are about showing object behavior and did not provide clear enough indication of control flow for the task of debugging.

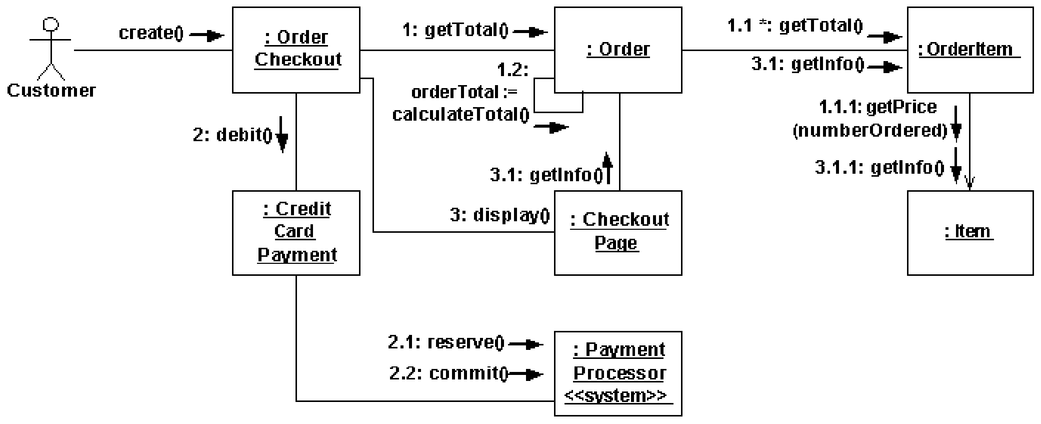


Figure 2.25: UML collaboration diagram [23]. The numbers show the order of method calls between the object instances which are represented by the boxes.

Sequence diagrams show flow control as a set of object instances with the execution of methods (called execution specifications) shown along “life-line” that run down below a box containing the object class. Method calls are shown as arrows between execution specification blocks with call arrows having solid lines and return arrows being dashed. Nested calls to functions are shown by adding execution bars on top of the outer function. Figure 2.26 shows an example of a UML sequence diagram.

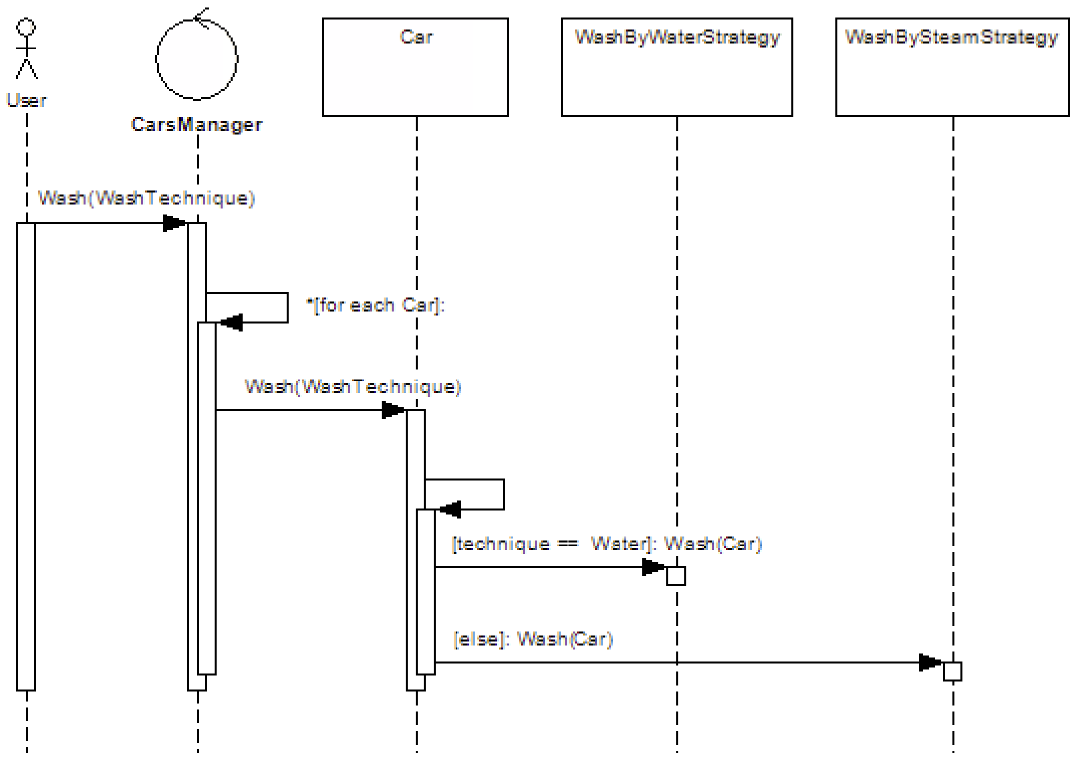


Figure 2.26: Basic UML sequence diagram [12]. Time flows vertically with the lines representing the execution history of the object instances and the narrow blocks on the lines showing when a method was called on an object.

Sequence diagrams show things such as loops, parallelism and locks using labelled rectangles (called combined fragments) surrounding a group of execution specifications, as shown in Figure 2.27.

Although there is some support for concurrency in UML, a number of researchers have found the support inadequate and have proposed extensions.

Xie et al [65] proposed a set of extensions to sequence diagrams, called saUML (synchronization adorned UML) to make them useful for teaching students about concurrency. They use different colours for the method blocks to indicate thread state, such as green for running, red for suspended and yellow for ready. When two or more threads

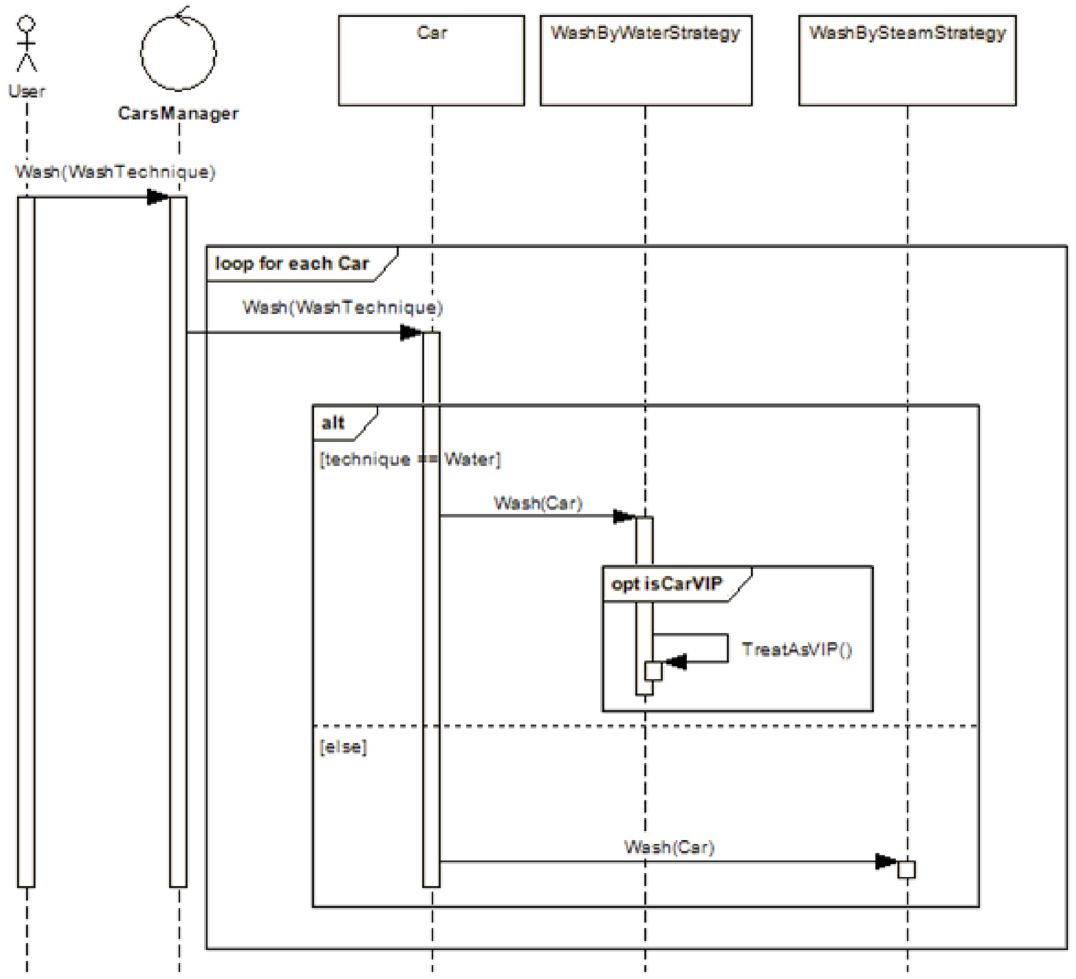


Figure 2.27: UML Sequence diagram with combined fragments [12]. The boxes are used to represent various aspects that are not covered by the lines and execution boxes.

are inside an object the colour block shows the deepest nested one. Small boxes with rounded corners overlaid on the method blocks indicate the state of the monitor of that object instance. Figure 2.28 shows an example of a saUML.

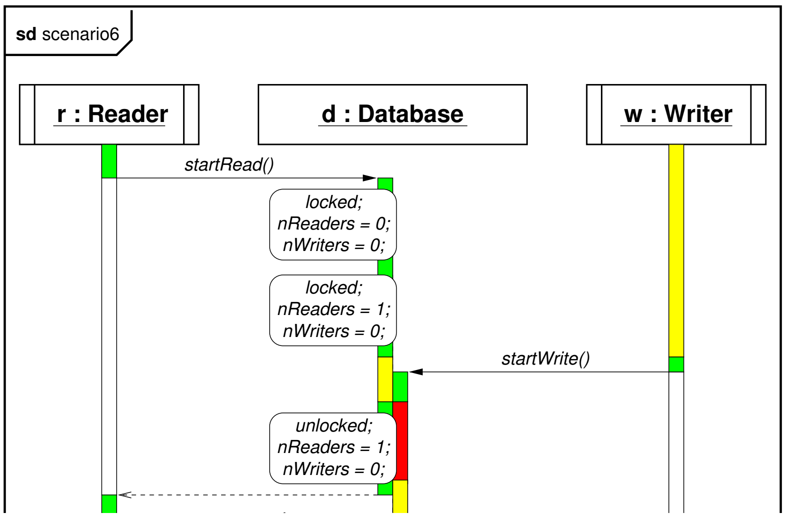


Figure 2.28: Diagram of saUML [65]. Colour is used to represent thread state and the rounded corner boxes show monitor state information.

Like TIE in section 2.4.5 Artho et al [27] describe a system that is used to visualize the results of model checking. To do this they extend UML in a number of ways to better support concurrency, including the addition of context switches and threads sleeping and waiting.

Unlike standard sequence diagrams, threads are represented as hexagons on the right-hand side of the diagram, rather than as active objects, which are represented as objects with thick borders. Execution is shown by dashed arrows leading from the thread hexagon to the object instances. Context switches are shown by placing a different thread hexagon below the previously executing thread, as seen in Figure 2.29.

When threads sleep, wait or call join a dashed arrow is shown returning to a thread hexagon. Actions which have a “happens before” relationship to events in other threads are shown as dotted lines running from the action to the thread hexagon. Examples of this kind of relationship are threads starting, notification and threads joining other threads, as shown in Figure 2.30.

JIVE [36] is a software system for visualizing and debugging the execution of Java programs using UML. Object diagrams are used to show the relationships of the object instances in a system to each other and Sequence diagrams are used to show the execution

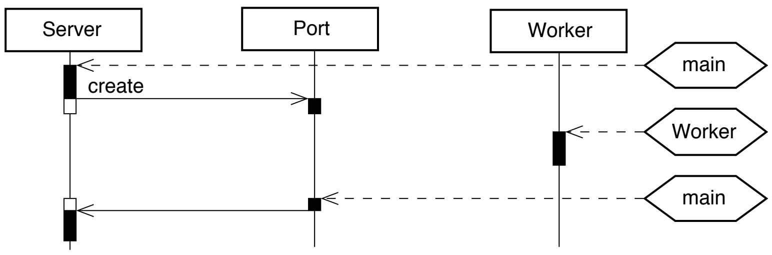


Figure 2.29: Context switches in Artho's UML extension [27]. Note the hexagon threads and the dashed lines that show when a thread starts executing in the object instances.

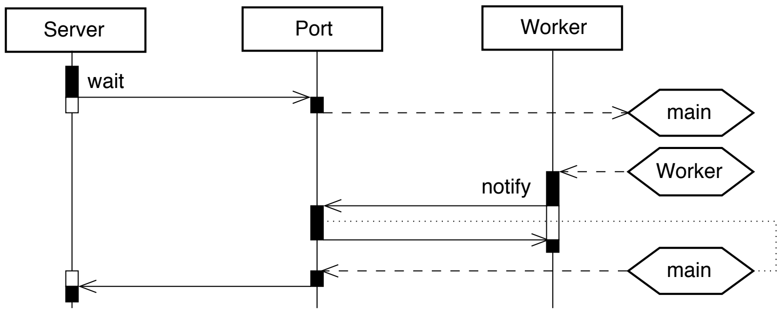
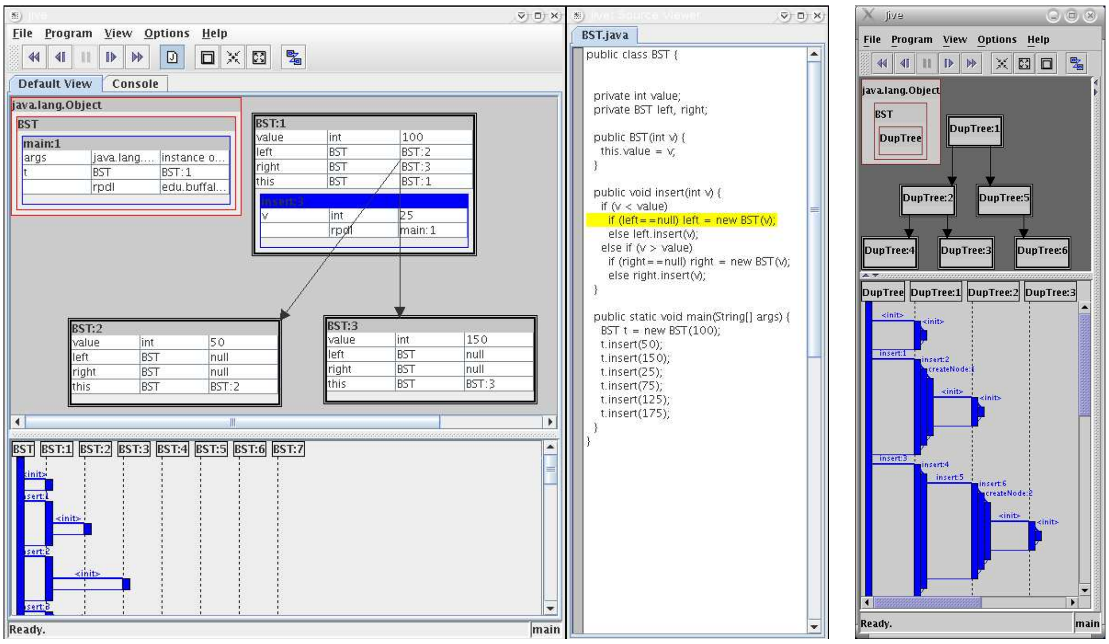


Figure 2.30: Condition variable wait and notification in Artho's UML extension. Note the dotted lines which demonstrate notification of a lock to the waiting thread.

of a thread amongst object instances. Various levels of zoom are available to provide more or less detail about an object's state, as can be seen in Figure 2.31.



```
public class BST {


  private int value;
  private BST left, right;

  public BST(int v) {
    this.value = v;
  }

  public void insert(int v) {
    if (v < value)
      if (left==null) left = new BST(v);
      else left.insert(v);
    else if (v > value)
      if (right==null) right = new BST(v);
      else right.insert(v);
  }

  public static void main(String[] args) {
    BST t = new BST(100);
    t.insert(50);
    t.insert(150);
    t.insert(25);
    t.insert(75);
    t.insert(125);
    t.insert(175);
  }
}
```

Figure 2.31: Overview of JIVE [36] showing the various panels. The top left shows the object instances and there values the bottom left shows a sequence diagram of the execution.

Different threads are indicated by different colours as can be seen in Figure 2.32.

2.4.7 Visual Programming Languages

Visual programming languages are programming languages that have a visual component, Shu [53] defines them as:

> *"a VPL can be informally defined to be a language which uses some visual representation (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language"*

In addition Chang et al [30] state that visual languages have three main goals:

> *"(1) to aim to make programming more accessible to some particular audience,"*

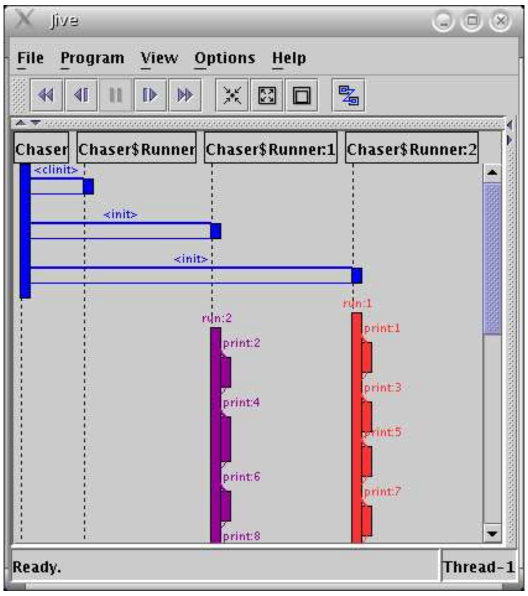


Figure 2.32: Multiple threads in JIVE [36]. Colour is used to differentiate the threads.

> *"(2) to aim to improve the correctness with which people perform programming tasks, and/or "*
>
> *"(3) to aim to improve the speed with which people perform programming tasks "*

A common type of visual language is the "boxes and wires" language where a programs elements such as branches, loops and mathematical expressions are represented somewhat like an engineering flow chart, as shown in Figure 2.33.

By Chang et al's [30] definition many of the systems we have investigated have elements of visual languages. As Diehl [34] notes visual languages essentially have a two directional information flow. A user performs actions sending information to the program; the program sends changes back by updating the user interface. The systems we have studied don't operate like this. Information flows from an executing program to the visualization system, not back from the visualization to the executing program.

Another difference is that often visual languages are showing the static structure of a program and not the dynamic behaviour of a system, which is important for debugging.

Finally, there are some large differences in overall goals. The systems we have studied have a variety of purposes, such as debugging, monitoring and in some cases documentation like UML. Generally they aren't designed to be complete programming languages environments.

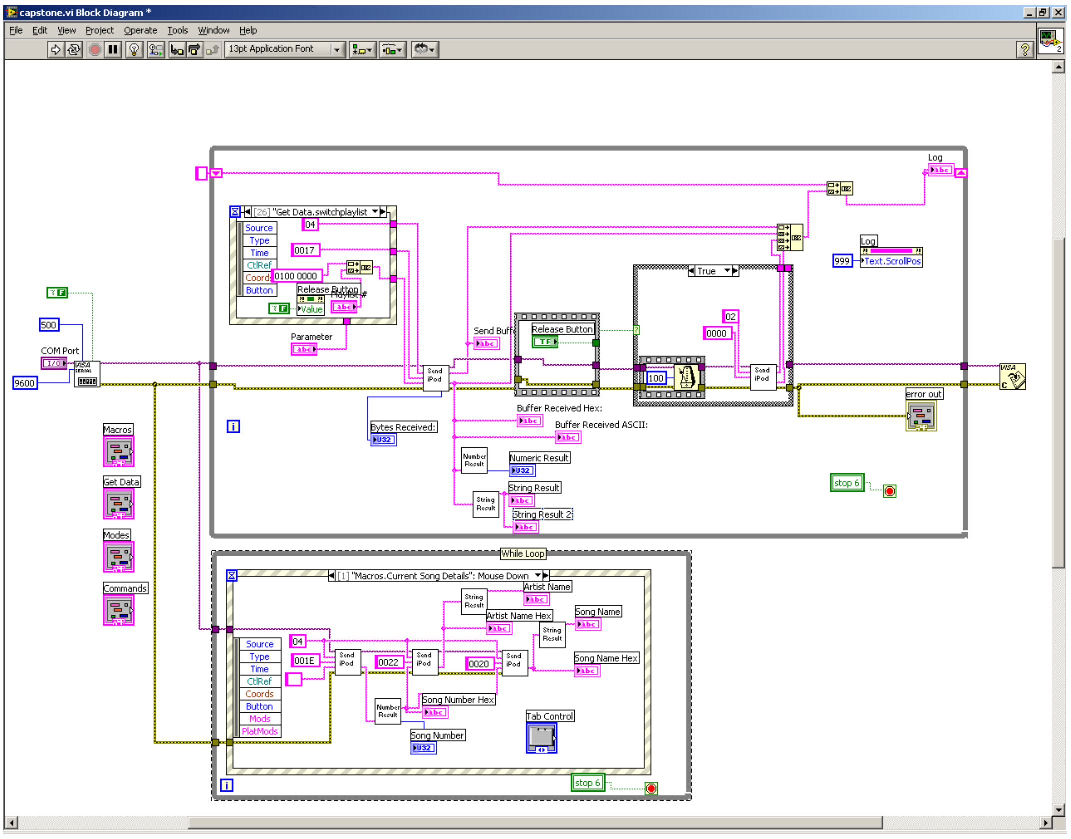


Figure 2.33: LabVIEW – an example of a "boxes and wires" visual language [20][6]. "boxes" are operations on a stream of data that are carried between the operations by the "wires". Some "boxes" will be used for input and output.

We decided that ultimately visual languages weren't directly relevant to our goals of improving concurrent debugging. In Section 6.2.6 of Chapter 6 we give a brief discussion of how our system could be extended to become a visual language.

2.4.8 Common Features Summary

Only a small sample of such a vast field has been given here, however we feel that we can draw some conclusions about some of the common elements, goals and of the field.

A common element across all the visualizations is the use of a timeline or space to represent the flow of time. This seems like the easiest approach as can be seen when compared to UML communication diagrams, which represent event order using a sequence of numbers.

Although systems like the Thread Interleaving Explorer were orientated towards finding errors (specifically concurrency errors) the majority of systems we looked at were oriented towards performance debugging. Debugging performance is obviously important, but we contend that it isn't used as frequently as error debugging is.

For data gathering, these systems generally gathered all, or at least a large amount, of data from the running program. This volume of data may be relevant to performance debugging, but it is somewhat hard to see its relevance to the performance of error checking debugging. In these cases programmers may have a fairly good idea of the location of the problem and may only need a very small set of trace data to achieve their goals.

Most systems, except for UML sequence diagrams, were oriented around threads rather than the objects.

Another problem that we noticed was the lack of debugging for program state, in the case of PARADE and JIVE it seems that how program state is changing over time isn't visualized effectively. The state of a program is arguably as important to error debugging as the location within an execution and something that we felt hasn't been properly addressed.

Finally, most of the work seems to be focused on providing a high level overview of an execution, even UML sequence diagrams often provide only an abstraction of an execution and not low level detail about state and specific program actions such as branches and loops that programmers need to conduct error debugging.

2.5 Temporal Notations from outside Software Visualization

While investigating software visualization, we noticed similarities between the visualizations being created for visualizing execution traces and software from the fields of music production software and Western music notation. It was realized that these were only two of a rich array of "visualizations" dealing with time from outside the traditional scope of visualization and information visualization.

What interested us about these fields is that they deal with complex multidimensional data and have been an enormous success from an artistic and market perspective. We hoped that an analysis of these systems in comparison to visualization would provide insight into what makes them effective and whether ideas from them can be used to build a better system for concurrent trace visualization.

Though there are many different systems and notations in this area we will outline the two we are most familiar with - Audio editing and sequencing software (particularly the software Logic Studio) and Music Notation. After giving an outline of each and how they use the various elements of visualization, we will briefly compare them to concurrency and software visualization.

Unfortunately, there does not seem to be a wide variety of published literature analyzing these systems as visualizations.

2.5.1 Music Notation

Although composers will write and edit music using notation, the primary use for notation is for musicians to be able to perform music. In this section we will focus on the elements of music notation common to most instruments and its common uses, so we will make certain assumptions for example we will use a treble clef and a 4/4 meter for the sake of simplicity.

At the most basic level music could be viewed as a scatter plot relating pitch and time, with pitch being determined by the position of the mark vertically and the sequence in time being determined by the horizontal position. The plot points are called notes and have an ellipsoid shape, horizontal lines, called a stave, make it easier for performers to see the pitch of a note. This is shown in Figure 2.34.

This doesn't tell the whole story since music where the notes were all the same duration would be boring, so different relative lengths are shown using several different glyphs as shown in Figure 2.35. Bar lines are used to divide up major divisions of time

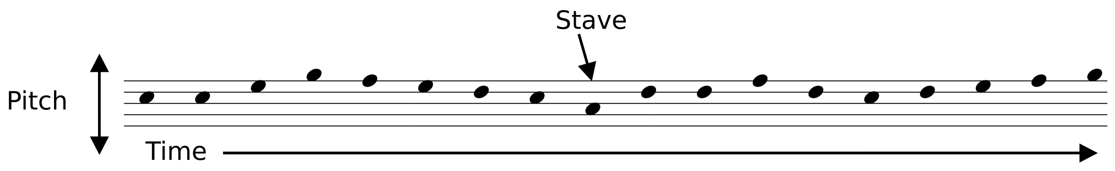


Figure 2.34: Basic music notation showing pitch, time and the stave.

in this case every bar has 4 subdivisions that are one quarter note long (a whole note takes up all four beats in a bar). In practice, notes of longer duration are engraved with more space in the bar, providing two indications to the performer of the note duration.

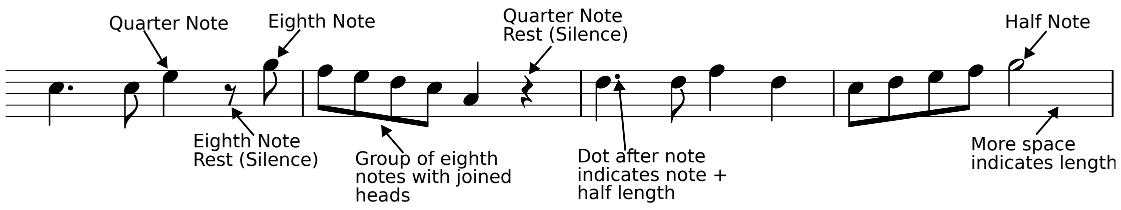

Quarter Note
Eighth Note
Quarter Note Rest (Silence)
Half Note
Eighth Note Rest (Silence)
Group of eighth notes with joined heads
Dot after note indicates note + half length
More space indicates length

Figure 2.35: Note lengths indicated by different glyphs.

Music is not simply a series of pitches of constant volume played at constant speed with the instrument producing the same tone every time, so various methods have been invented to indicate volume, speed and timbre.

Volume is indicated by text markings such as p for soft and f for loud. Changes in volume are indicated by crescendo lines, as shown in Figure 2.36.

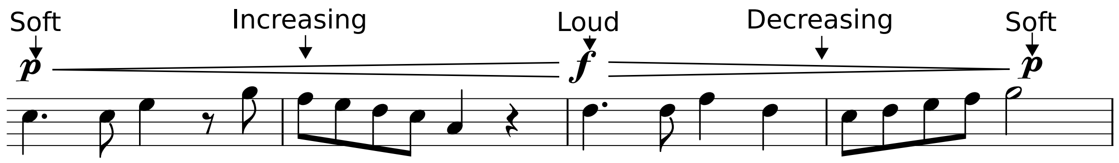


Figure 2.36: Text and lines indicating volume.

Text is used to indicate what speed a section should be played at, often in combination with a text metronome marking. Text can also be used to indicate speeding up or slowing down as is shown in Figure 2.37

Various glyphs are commonly used to mark changes to the timbre of an instrument such as staccato marks (a dot above a note) indicating a short and sharp attack then silence. Lines are also used to indicate that a phrase should be played smoothly Also glyphs and text are widely used for instrument-specific things such as fingering numbers for guitar music as is shown in figure 2.38.

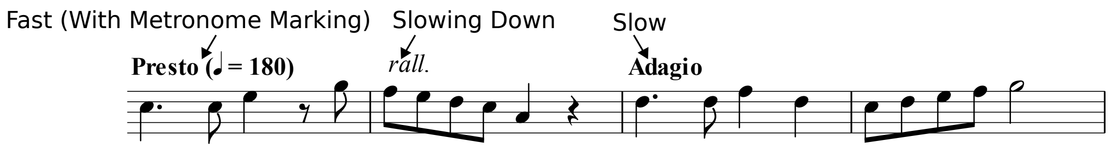


Figure 2.37: Different text used to indicate speed and changes to speed.

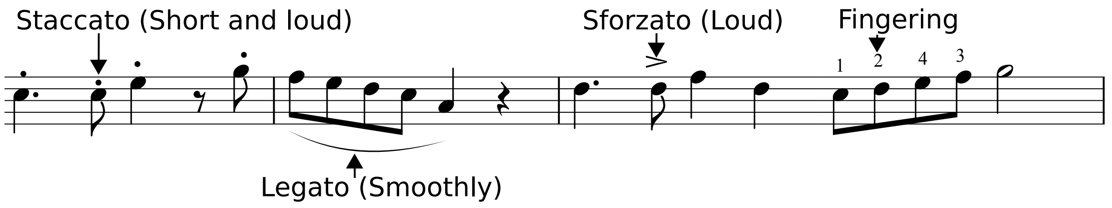


Figure 2.38: Different glyphs used to communicate a variety of different kinds of information.

By definition, music on a page is static so there is little interaction beyond what a pen or pencil can do. There is however a micro and macro view possible since the shape of a line of notes can provide a kind of macro view to a user. This is possibly more important for conductors of groups of musicians, who use a score that combines multiple parts. A conductor can use the overall shape to tell what is happening without having to recognize every single note of every single instrument.

More information about music notation can be found in 'Music Notation' by McGrain [43]. An overview of the history of music notation can be found in Rastall [47].

2.5.2 Non-Linear Audio Editing

Historically, the recording and editing of music was conducted entirely aurally using multitrack tape recorders. The creation of music recording software such as the Logic Studio [8] series allowed musicians to see a visual representation of the recording or composition they were creating.

At the heart of this software is the timeline view (Figure 2.39), which provides an overall view of the production. Vertical space is used to divide the different "tracks" or instruments, and horizontal space is used for the blocks of content of the tracks. Tracks can be composed of different information, either digital sound recorded from an external source or midi data used to create sounds on synthesizers. Often, a zoomed out view of this content is displayed in the content blocks as shown below, to give some idea of what a track is composed of.

The lines overlaid on the tracks are "console automation", which is named after

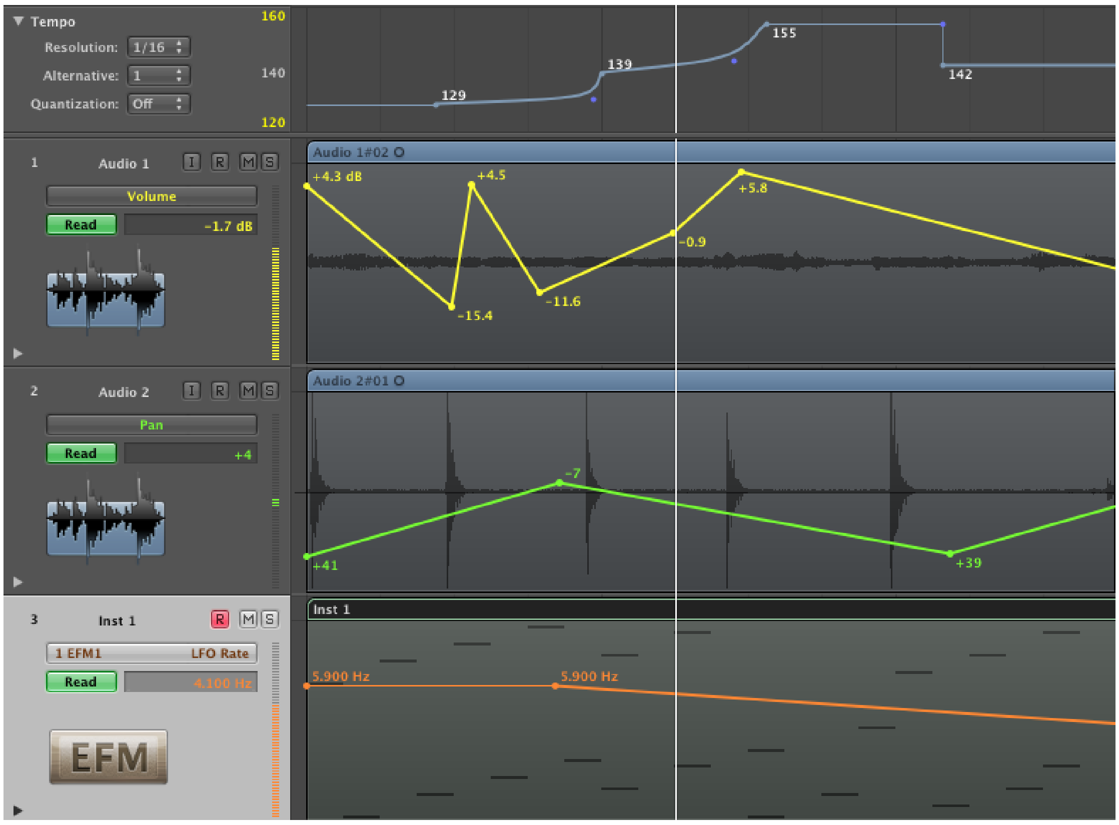


Figure 2.39: Track view of Logic Audio showing audio and midi tracks with console automation lines overlaid [8].

the motorized sliders sometimes seen on large mixing consoles. This allows control of parameters such as volume and panning over time.

Colour is used throughout to differentiate elements of the production.

Numerous subsidiary views exist providing things such as mixing control, transport control (play, rewind etc), control of synthesizers and more. Of most interest to us are the piano roll view and the sound editor view. These provide low level editing of the different kinds of content used in music production software.

The piano roll view (Figure 2.40) is based on old player pianos, all the notes and semitones are arranged vertically with notes and their length arranged horizontally as shown. Often, colour is used to indicate relative volume of individual notes, providing information similar to music notation. The screen can be scrolled in both directions to allow longer passages and higher or lower notes.

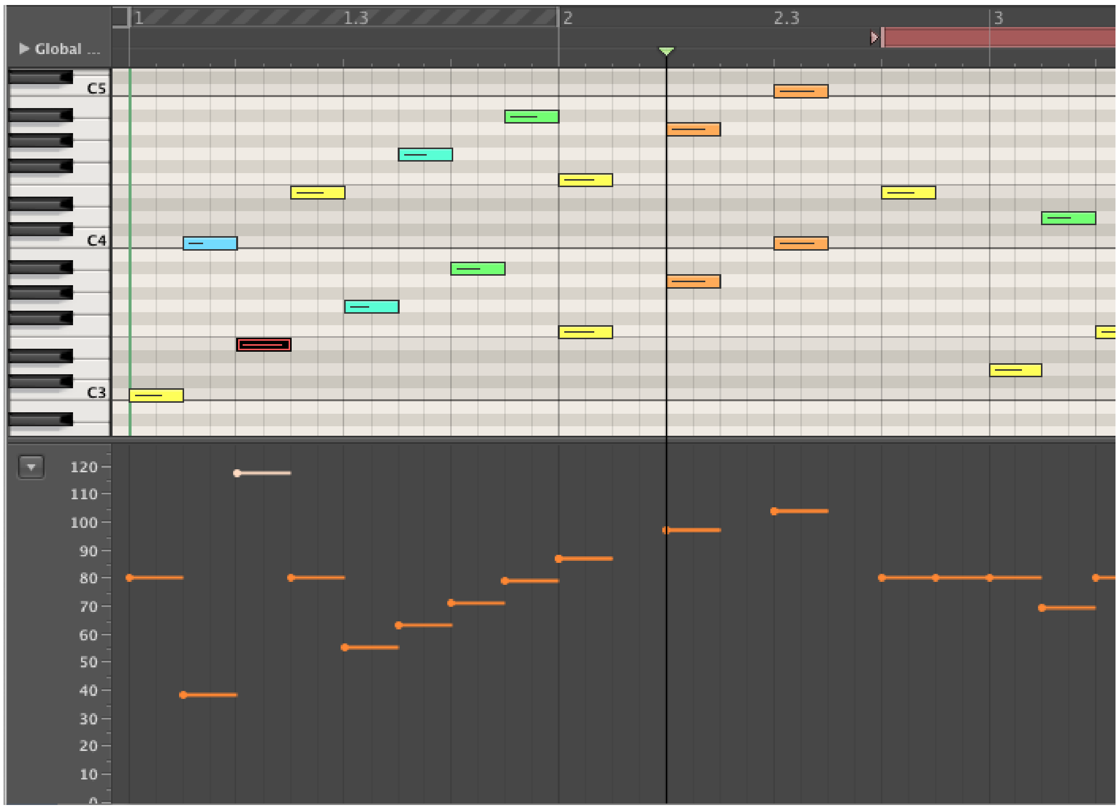


Figure 2.40: Piano roll view of midi data with note pitch at the top and note volume at the bottom [8].

The sound editor view (Figure 2.41) allows editing of the recorded sound down to the sample level.

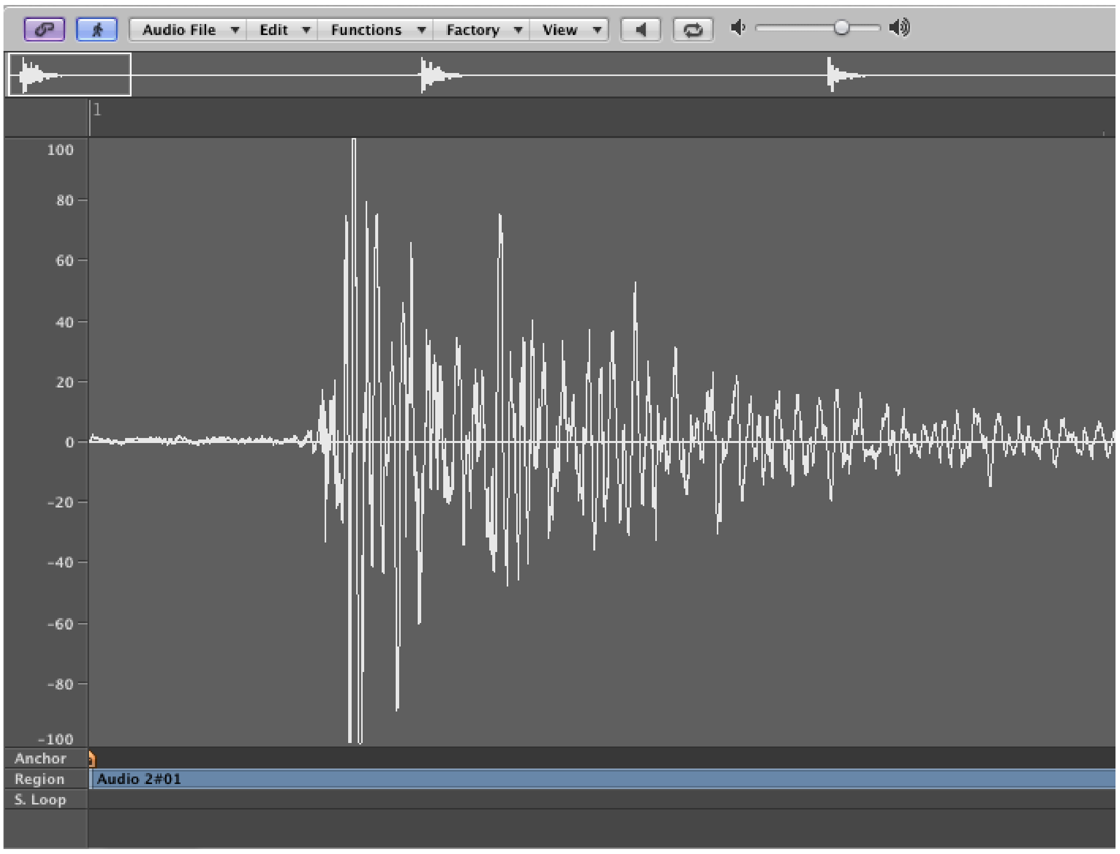


Figure 2.41: Sound edit view showing editable sound file [8].

These views provide the micro/macro view that is common to visualization, providing detailed information and control over subsets of the data.

2.5.3 Comparison to Software Visualization

There are several ways we can compare software visualization, but first it will be useful to look at some of the differences between music and programming and in particular concurrent programming.

Music is (generally) composed as a series of events fixed in time at the point of composition. In contrast, the position in time of events in a concurrent program is determined by their sequential order in the program and also by the operating systems scheduler at runtime. This means there is no single "performance" of a program, as there is with music.

Another important difference is that music is essentially "embarrassingly parallel" although there is communication in the form of rhythm this is completely synchronous, happening at regular and easily predictable intervals that are known in advance. This is completely different from many parallel applications, which have an undefined pattern of events. There is also no competing for shared resources, you will never see two violinists competing for a bow, which is a common feature of concurrency.

Despite this, we feel that the most important difference is the goals of these approaches and software visualization. Software visualization takes its goals and impetus from scientific visualization and information visualization, in that it attempts to organize and summarize complex data to provide insights that might not be obvious from the display of raw data. This leads to things like multiple views and call stack views present in Zinsight or PARADE which provide independent views of an underlying data set to gain insights into different areas of a programs execution.

In contrast recording, editing and composing have a functional goal of allowing the most efficient production of music. It is assumed that a user will spend an often considerable amount of time familiarizing themselves with the notation or software. Ultimately, this means that the goal isn't simplification, but displaying as much of the information as possible in as small as space as possible to make it easy for an experienced user to perform their task.

If music notation was designed in the same way as some of the software visualizations we have looked at, there would be separate "views" for the speed, dynamics, volume and time versus pitch, instead these are all integrated into a single view allowing rapid communication of information to the user. This "integrationist" approach is also seen to a lesser extent in software like Logic Studio, which has a main view timeline view with dense integrated data and numerous subsidiary views displaying additional data. We feel that these sub-views are categorically different from the views found in software, since they usually aren't meant to be viewed at the same time and often aren't absolutely crucial to understanding the production in its entirety. It is possible that these sub-views could be designed away with features such as a zoomable timeline, which smoothly integrates low level editing and high level editing.

The "integrationist" approach greatly influenced our thinking about how to design a system useful to programmers debugging concurrent programs.

Chapter 3

Design

3.1 Design of Beat

Our investigation of existing visualizations and temporal notations led to the design of a new trace visualization system called Beat. In this section we address the design of Beat and how it draws from and improves on existing visualizations and temporal notations to meet the goals we laid out in the introduction. In particular, we will discuss what data we chose to visualize, how we used the various elements of visualization we have previously identified and various other issues specific to the data we were visualizing.

3.2 Visualization Data

To meet our goals of being useful for debugging it was important that the right information be selected for visualization.

Existing trace visualizations informed our design in their use of tracing method entry and exit events and displaying the method calls as blocks. Also, designs such as PARADE mentioned in section 2.4.2 informed our decision to include information about locks and thread state.

From the temporal notations we had investigated we decided that we wanted to include lower level data such as the source code, and information about loops, branches and exceptions in some form. We hoped that the addition of this data would make debugging more effective, as the addition of lower level data in the temporal notations did.

Experience with tracing frameworks such as DTrace suggested that inclusion of state in our diagram would greatly improve the usefulness of our software for debugging.

Areas such as the static program structure didn't interest us as much because we didn't feel that they would help with debugging, being quite high level in contrast to our goals of representing lower level information effectively. Also, this information didn't seem to be used much by existing software visualizations, leading us to believe it wasn't much use for debugging.

3.3 Space

As with the other techniques we studied, the main component of our design is space. Music software and notation gave us the idea of using all the space for a single, highly integrated view for all the data, rather that using separate windows for different data.

Many of the trace visualizations we had investigated had time running along the horizontal direction, we decided against doing it this way because we were going to include source code in the diagram which "naturally" runs with time vertically rather than horizontally.

Having the source code in the diagram also added to our decision to use tracks like in audio software, rather than lines like UML sequence diagrams. As we mention in our discussion of existing visualizations there were two different approaches to using tracks, one was to be have a track represent a thread like PARADE. The other was to represent object instances like UML sequence diagrams. We decided on the second approach because we felt that an object instance view would be more familiar to our users and useful for the task of correctness debugging. Figure 3.1 shows the basic layout of tracks.

Method calls were represented by blocks similar to UML sequence diagrams or Zinsight, as shown in Figure 3.2.

3.4 Text

The decision to include source code text within the method blocks was inspired by how audio visual software includes elements such as wave forms and images within the blocks of content. We feel by including text we can gain similar advantages of making it easy to relate the visual elements to the underlying sounds or execution.

Object instances are identified by having their name in a fixed position header for the track. As an example for our design we will use Java code with a basic producer consumer program. Due to limits of horizontal space some parts of the source code will

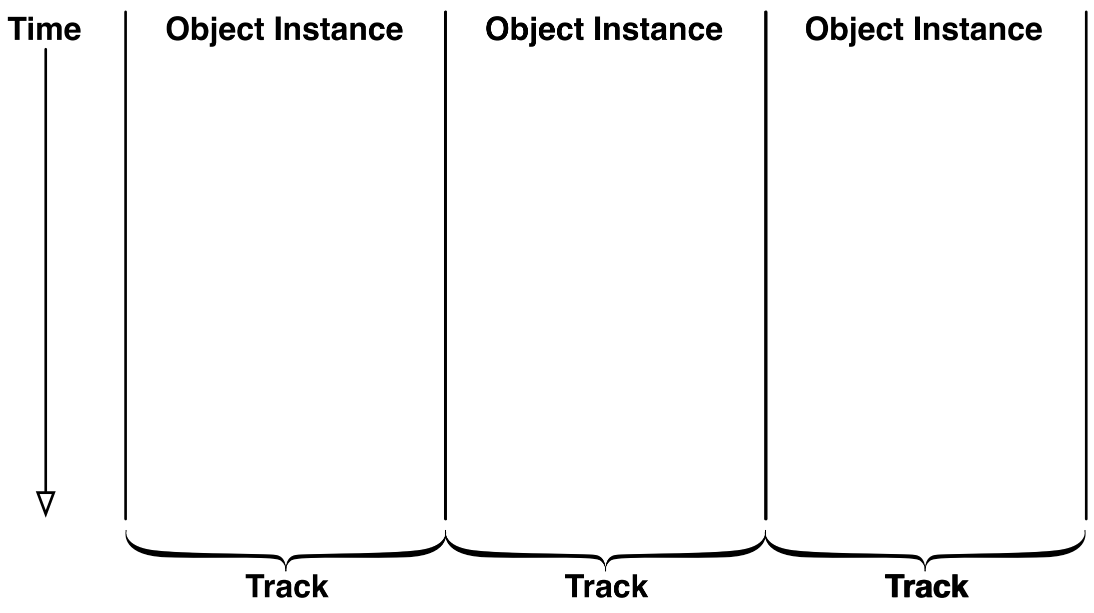


Figure 3.1: Basic use of space for object instances in Beat.

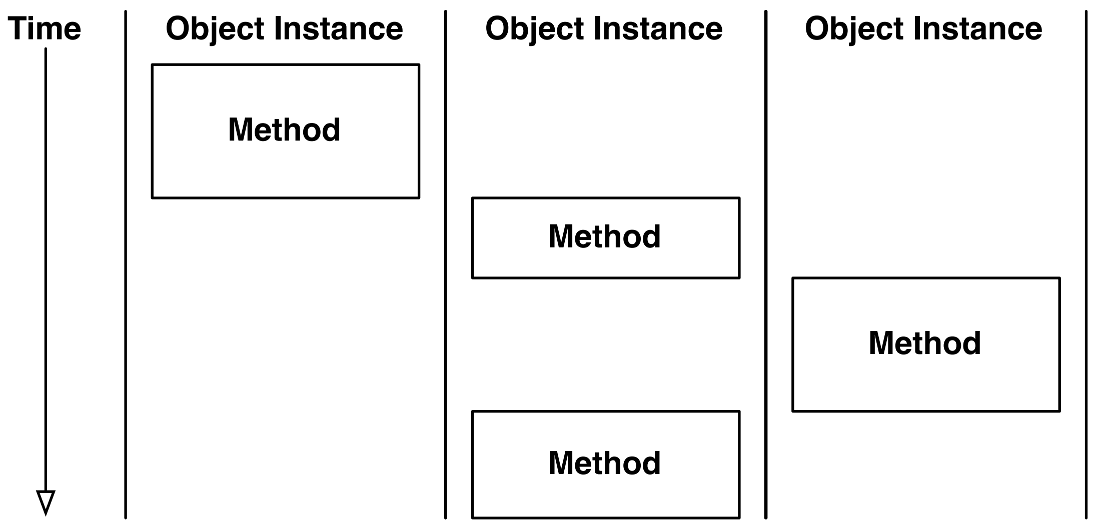
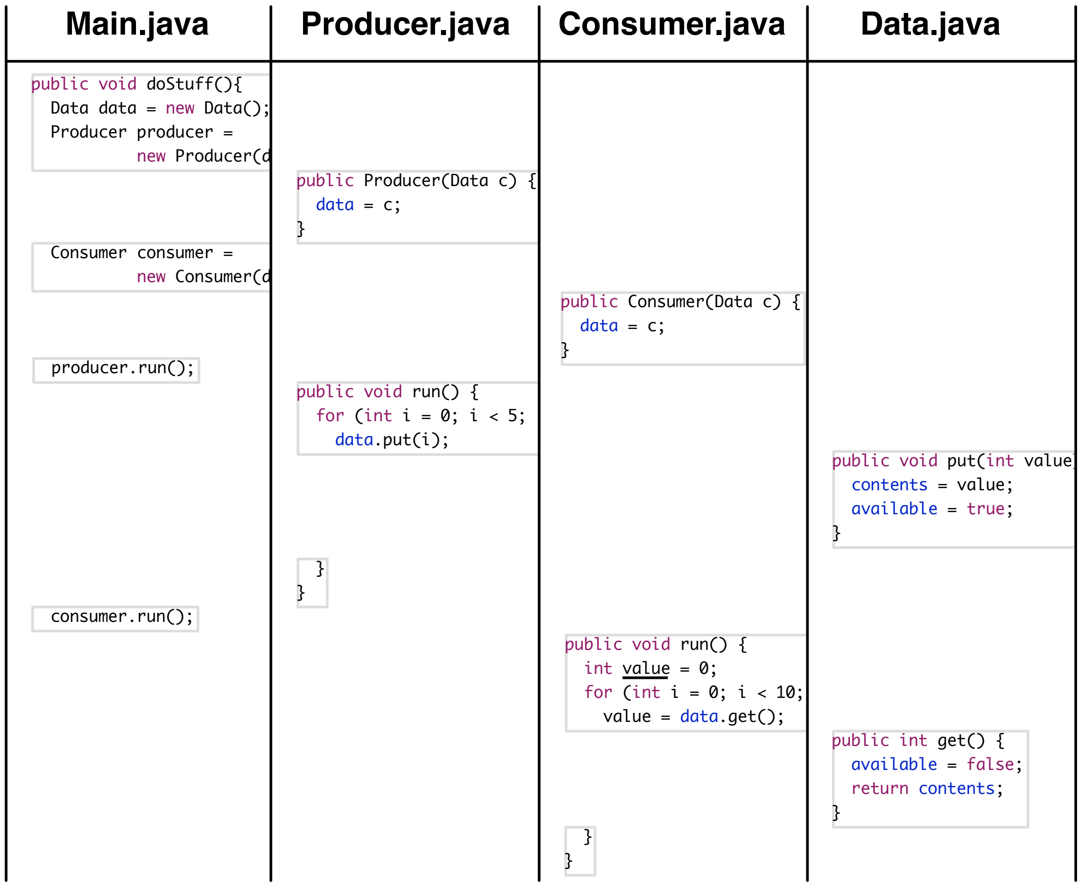


Figure 3.2: Method calls to object instances represented as blocks in the object tracks.

be cut off, the full source code for these examples will be in the appendices. This was noted as a problem and some methods of dealing with it are discussed in section 3.10.

Because we were concerned that if method blocks got particularly large the user might not be able to identify which method had been called when it returned we added the line of the method call at the top of the method block where the method call returns. The addition of source code can be seen in Figure 3.3 below.



```
public void doStuff(){
  Data data = new Data();
  Producer producer =
           new Producer(d
```

```
public Producer(Data c) {
  data = c;
}
```

```
  Consumer consumer =
           new Consumer(d
```

```
public Consumer(Data c) {
  data = c;
}
```

```
  producer.run();
```

```
public void run() {
  for (int i = 0; i < 5;
    data.put(i);
```

```
public void put(int value
  contents = value;
  available = true;
}
```

```
  }
}
```

```
  consumer.run();
```

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10;
    value = data.get();
```

```
public int get() {
  available = false;
  return contents;
}
```

```
  }
}
```

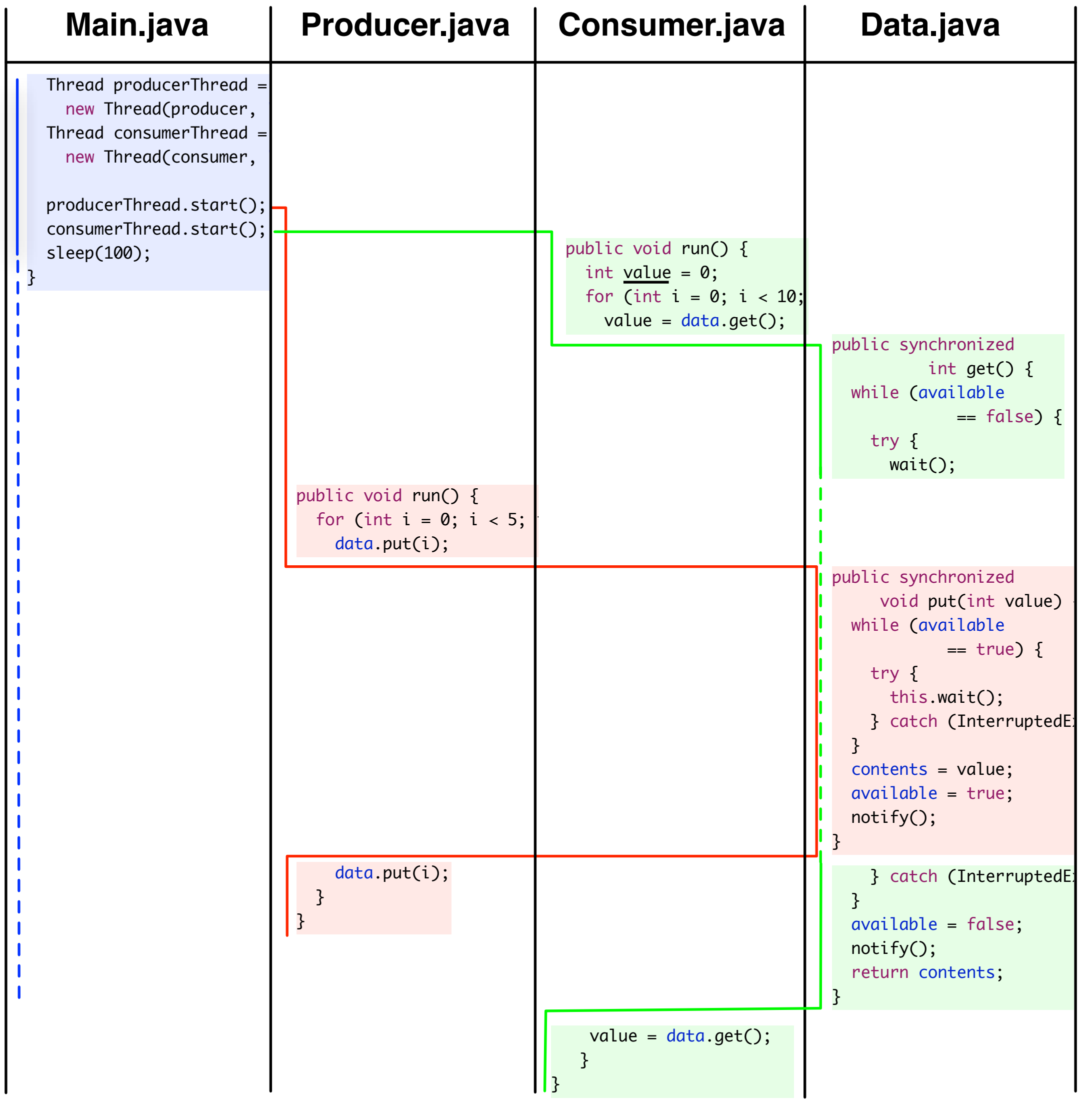
Figure 3.3: Design with source code included in method call boxes.

3.5 Lines, Colors and Icons

Lines are used to make the flow of execution between methods in object instances clear, similar to the way that they are used in UML sequence diagrams. Lines are placed as an overlay on top of the blocks and object tracks.

There were a number of different things that could have been represented using colour, we could have used it to represent thread state like saUML or to differentiate threads like JIVE. We felt a problem with using colour for thread state would make it hard for the user to differentiate threads when there were a large number executing at the same time. By using colors for threads it would be easy to distinguish threads.

Another area we applied colour was to the method blocks, where we gave a soft version of the thread colour as a background so that it was obvious what thread was executing in a method. Lines and colours are shown in Figure. 3.4.



```
Thread producerThread =
  new Thread(producer,
Thread consumerThread =
  new Thread(consumer,

producerThread.start();
consumerThread.start();
sleep(100);
}
```

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10;
    value = data.get();
```

```
public synchronized
          int get() {
  while (available
            == false) {
    try {
      wait();
```

```
public void run() {
  for (int i = 0; i < 5;
    data.put(i);
```

```
public synchronized
     void put(int value)
  while (available
           == true) {
    try {
      this.wait();
    } catch (InterruptedE
  }
  contents = value;
  available = true;
  notify();
}
```

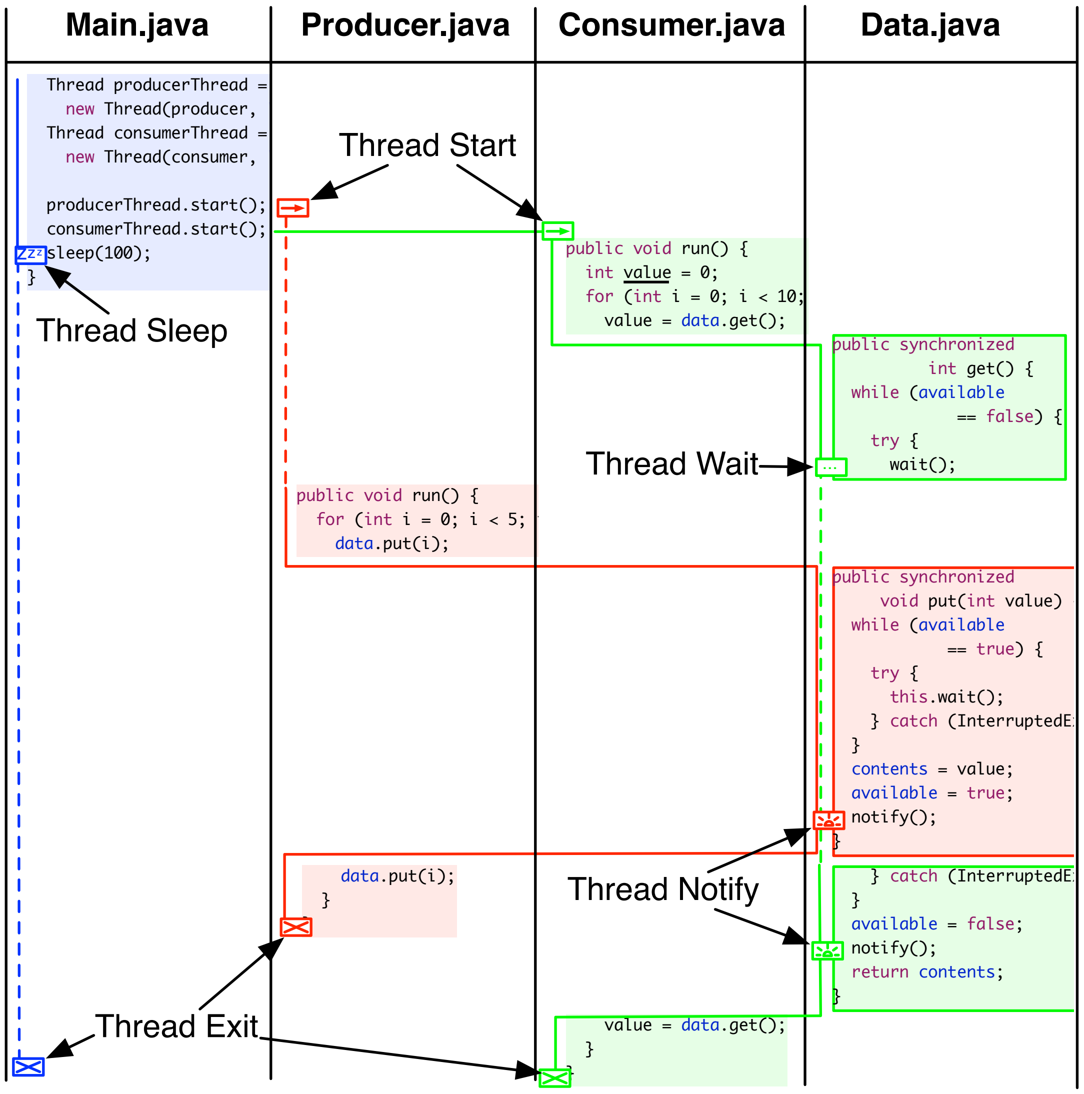
```
    data.put(i);
  }
}
```

```
    } catch (InterruptedE
  }
  available = false;
  notify();
  return contents;
}
```

```
    value = data.get();
  }
}
```

Figure 3.4: Adding lines showing path of execution between method blocks.

Different thread specific events (such as a thread starting, waiting or sleeping) were represented using icons. A dashed line was used to indicate the period of time that a thread was inactive i.e. sleeping or waiting. A bright border around synchronized code blocks and methods was used to indicate that an object was locked by a thread. This can be seen in Figure. 3.5.



```
Thread producerThread =
   new Thread(producer,
Thread consumerThread =
   new Thread(consumer,

producerThread.start();
consumerThread.start();
sleep(100);
}
```

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10;
    value = data.get();
```

```
public synchronized
          int get() {
  while (available
            == false) {
    try {
      wait();
```

```
public void run() {
  for (int i = 0; i < 5;
    data.put(i);
```

```
public synchronized
     void put(int value)
  while (available
          == true) {
    try {
      this.wait();
    } catch (InterruptedE
  }
  contents = value;
  available = true;
  notify();
}
```

```
    data.put(i);
  }
```

```
    } catch (InterruptedE
  }
  available = false;
  notify();
  return contents;
}
```

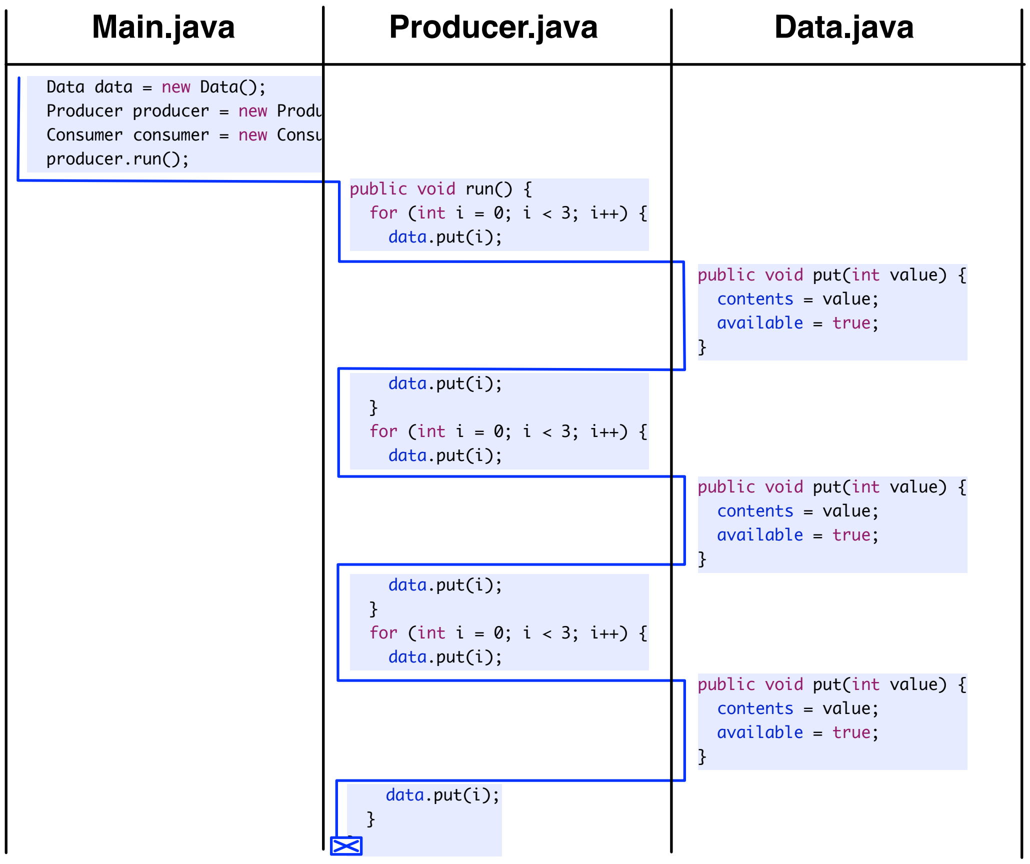
```
    value = data.get();
  }
}
```

Figure 3.5: Thread state being shown by icons and dashed lines.

3.6 Loops, Branches, Errors and Exceptions

These elements of program execution represent jumps to other points within the linear flow of the source code and so need some special handling.

For loops it was decided to simply repeat the loop the number of times it was executed, at shown in Figure. 3.6.



```
Data data = new Data();
Producer producer = new Produ
Consumer consumer = new Consu
producer.run();
```

```
public void run() {
  for (int i = 0; i < 3; i++) {
    data.put(i);
```

```
public void put(int value) {
  contents = value;
  available = true;
}
```

```
    data.put(i);
  }
  for (int i = 0; i < 3; i++) {
    data.put(i);
```

```
public void put(int value) {
  contents = value;
  available = true;
}
```

```
    data.put(i);
  }
  for (int i = 0; i < 3; i++) {
    data.put(i);
```

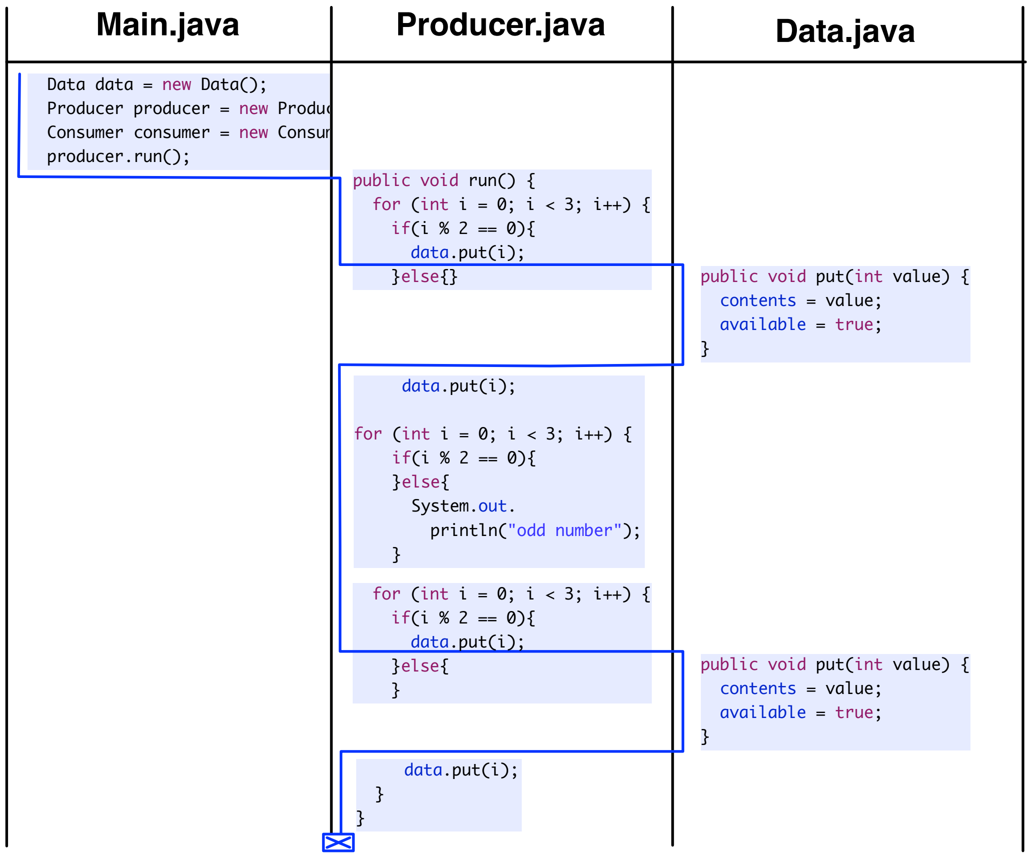
```
public void put(int value) {
  contents = value;
  available = true;
}
```

```
    data.put(i);
  }
```

Figure 3.6: Thread looping shown by repeated loop body.

For branches it was decided that we would only show the content of the block executed by the branch, this is shown in Figure 3.7.

For errors and exceptions the place where the exception is thrown is usually different from where an exception is caught. Throws use a T icon and catches use a C icon, errors terminate the thread using an E icon, as shown in Figure 3.8.



```
Data data = new Data();
Producer producer = new Produc
Consumer consumer = new Consum
producer.run();
```

```
public void run() {
  for (int i = 0; i < 3; i++) {
    if(i % 2 == 0){
      data.put(i);
    }else{}
```

```
public void put(int value) {
  contents = value;
  available = true;
}
```

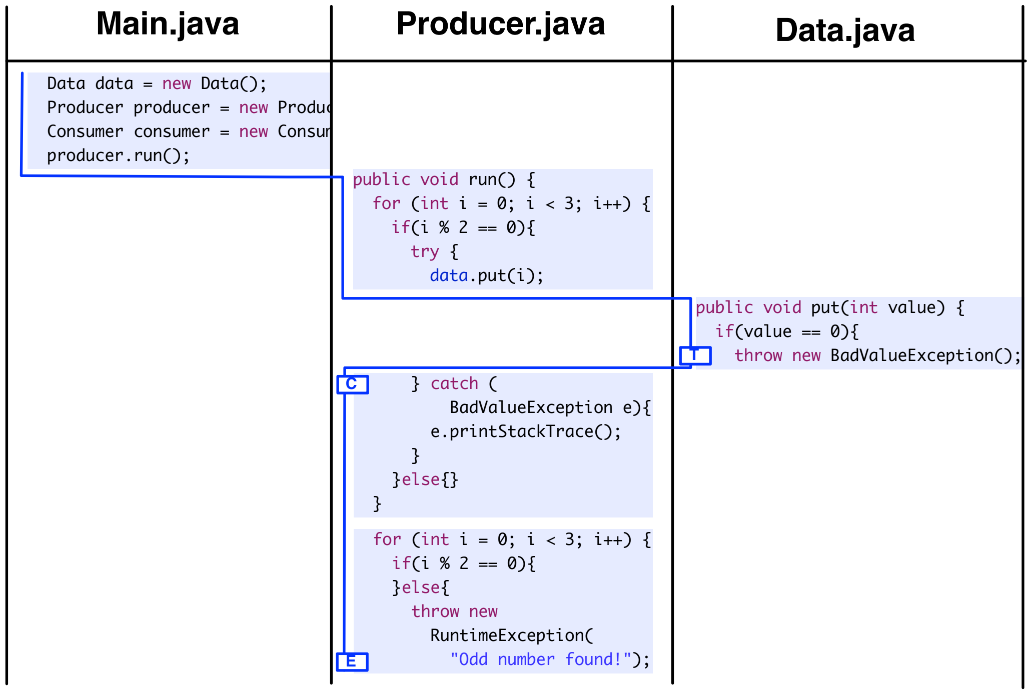
```
     data.put(i);

for (int i = 0; i < 3; i++) {
    if(i % 2 == 0){
    }else{
      System.out.
        println("odd number");
    }
```

```
  for (int i = 0; i < 3; i++) {
    if(i % 2 == 0){
      data.put(i);
    }else{
    }
```

```
public void put(int value) {
  contents = value;
  available = true;
}
```

```
     data.put(i);
  }
}
```

Figure 3.7: Branches shown by only showing the content of the branch taken.



```
Data data = new Data();
Producer producer = new Produc
Consumer consumer = new Consur
producer.run();
```

```
public void run() {
  for (int i = 0; i < 3; i++) {
    if(i % 2 == 0){
      try {
        data.put(i);
```

```
public void put(int value) {
  if(value == 0){
    throw new BadValueException();
```

```
      } catch (
          BadValueException e){
        e.printStackTrace();
      }
    }else{}
  }
```

```
  for (int i = 0; i < 3; i++) {
    if(i % 2 == 0){
    }else{
      throw new
        RuntimeException(
          "Odd number found!");
```

Figure 3.8: Exceptions in a program thread shown by T, C and E icons.

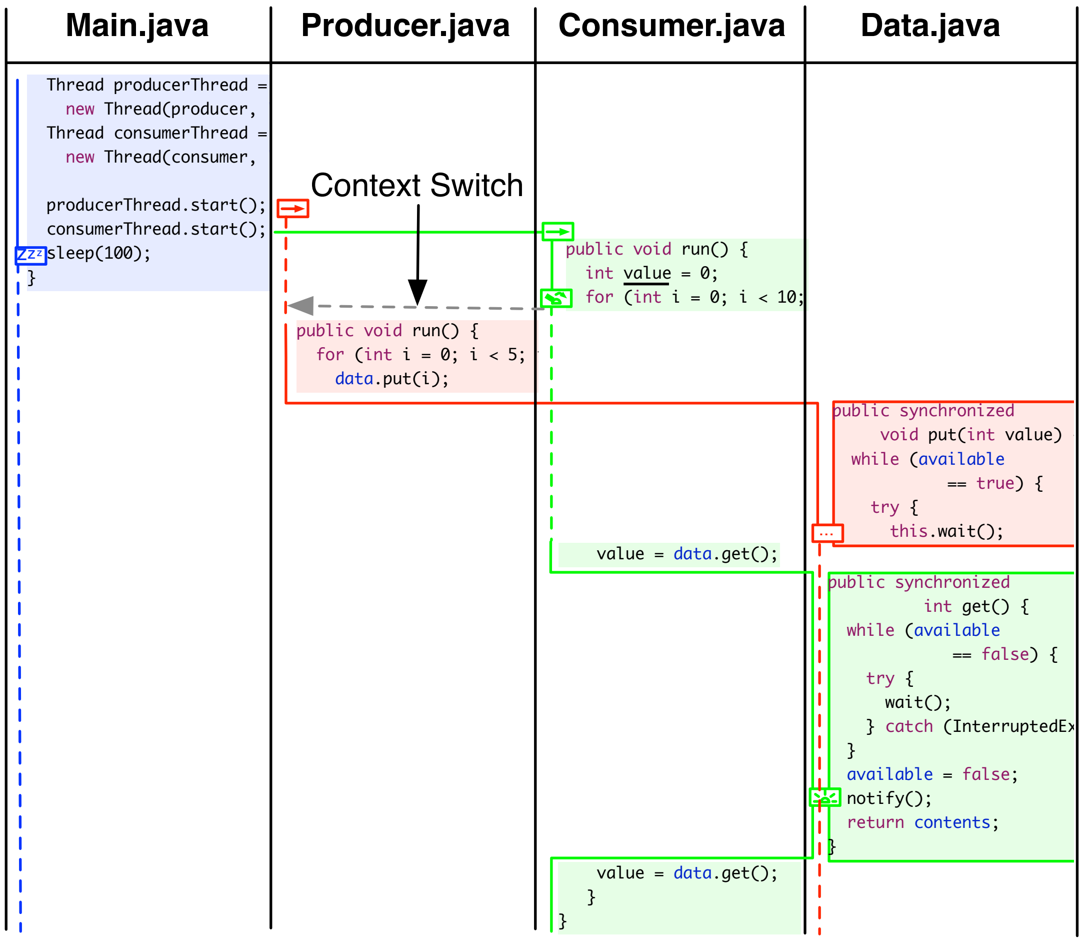
3.7 Context Switches

As discussed in section 2.2.5, a context switch is when the operating system scheduler decides that a thread has had enough time on the processor and needs to be exchanged for another thread or program. Context switches cut across the other elements of our design since they potentially represent an essentially arbitrary transfer of control between two threads in a system. We decided to represent context switches with a dashed grey line running between object columns showing when a context switch occurred in addition to an icon and the use of the dashed line to show when a thread was in active. The context switch design is shown in Figure 3.9.

It was felt that the addition of context switches would help improve understanding and in particular help debugging race conditions.

3.8 State

We considered a number of approaches for integrating state changes into our design. Our first concept was to include additional lines showing the variable that had been changed and what it had been changed to, with some method of highlighting the change, this



```
  Thread producerThread =
    new Thread(producer,
  Thread consumerThread =
    new Thread(consumer,

  producerThread.start();
  consumerThread.start();
  sleep(100);
}
```

```
public void run() {
  for (int i = 0; i < 5;
    data.put(i);
```

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10;
```

```
    value = data.get();
```

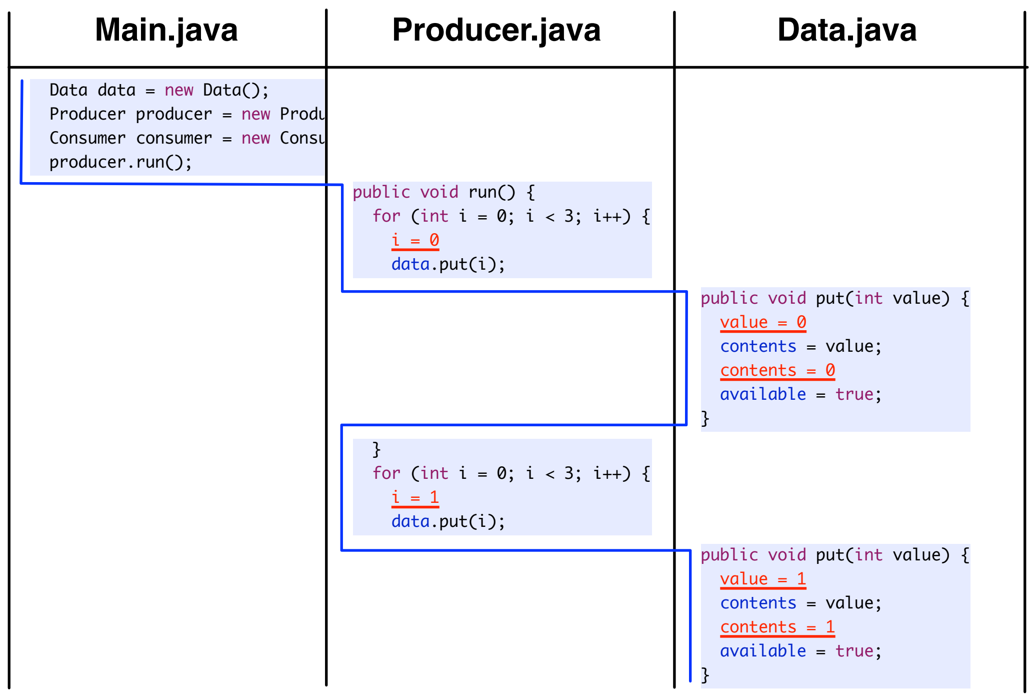
```
    value = data.get();
  }
}
```

```
public synchronized
     void put(int value)
  while (available
           == true) {
    try {
      this.wait();
```

```
public synchronized
          int get() {
  while (available
           == false) {
    try {
      wait();
    } catch (InterruptedEx
  }
  available = false;
  notify();
  return contents;
}
```

Figure 3.9: Context switch shown with switch icon and grey line running between thread lines.

can be seen in Figure 3.10.



```
Data data = new Data();
Producer producer = new Produ
Consumer consumer = new Consu
producer.run();
```

```
public void run() {
  for (int i = 0; i < 3; i++) {
    i = 0
    data.put(i);
```

```
public void put(int value) {
  value = 0
  contents = value;
  contents = 0
  available = true;
}
```

```
  }
  for (int i = 0; i < 3; i++) {
    i = 1
    data.put(i);
```

```
public void put(int value) {
  value = 1
  contents = value;
  contents = 1
  available = true;
}
```

Figure 3.10: Changes in program variables represented by additional lines showing what variables have been changed to.

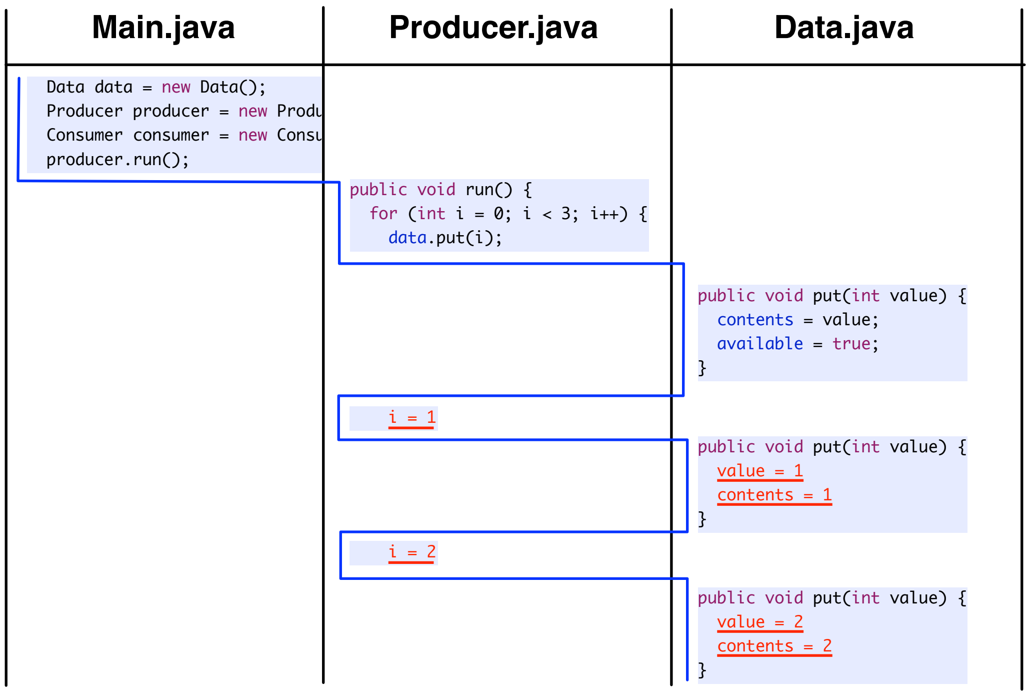
Our second concept was to have two views for method blocks; the source code and the state changes, as shown in Figure 3.11 below.

One of the challenges of visualizing state changes was that if an object field was changed in methods of other objects that weren't being visualized, it would mean that changes wouldn't be visualized. A method to fix this would be to do a before and after on an objects fields or local variables when they were passed to methods that weren't being visualized.

In addition, there has to be a way of filtering the state data to show just the important variables or changes. Another useful feature is to provide a method of displaying complex objects like arrays and hashes in a textual form.

3.9 Concurrency Errors

Some method of highlighting concurrency errors would be useful, so we considered placing a box with a jagged border around the location of the concurrency error, as shown



```
Data data = new Data();
Producer producer = new Produ
Consumer consumer = new Consu
producer.run();
```

```
public void run() {
  for (int i = 0; i < 3; i++) {
    data.put(i);
```

```
public void put(int value) {
  contents = value;
  available = true;
}
```

```
  i = 1
```

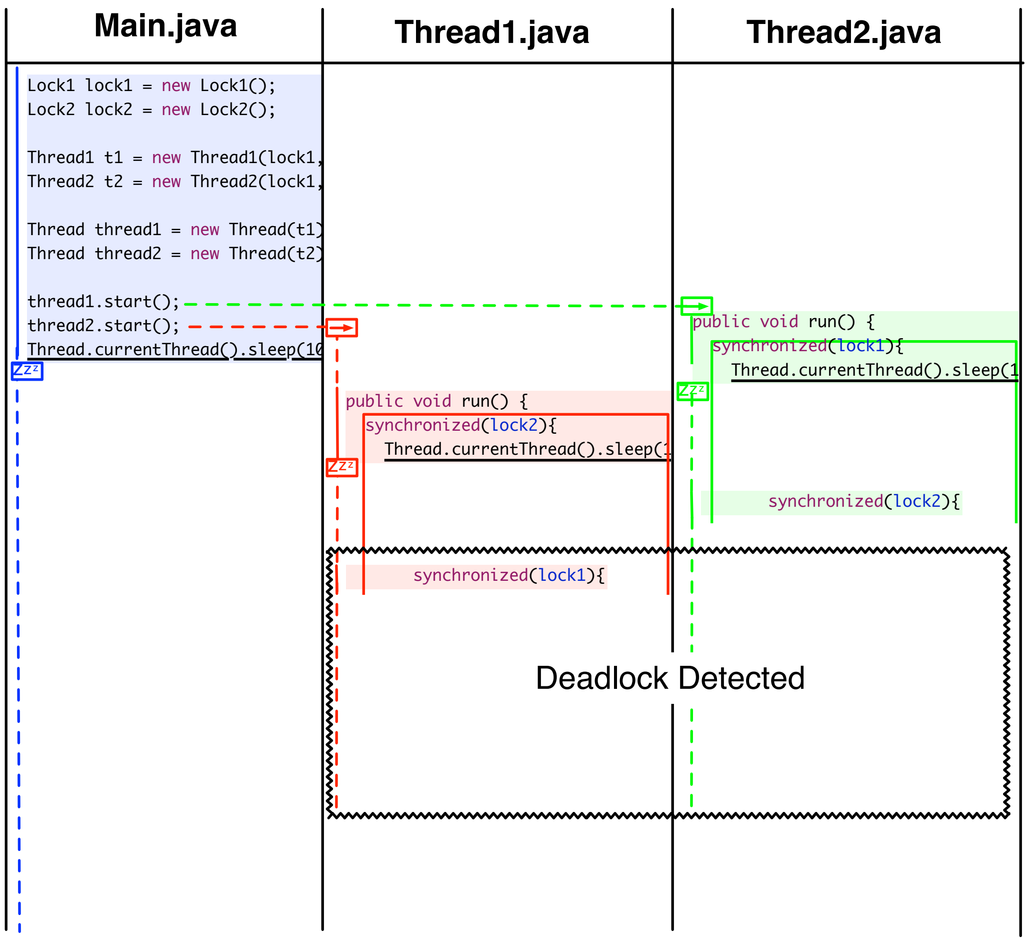
```
public void put(int value) {
  value = 1
  contents = 1
}
```

```
  i = 2
```

```
public void put(int value) {
  value = 2
  contents = 2
}
```

Figure 3.11: State only mode for method blocks. Source code is hidden and only changes to program variables are shown.

in Figure 3.12. We were unable to complete this as we will discuss in Chapter 4.



```
Lock1 lock1 = new Lock1();
Lock2 lock2 = new Lock2();

Thread1 t1 = new Thread1(lock1,
Thread2 t2 = new Thread2(lock1,

Thread thread1 = new Thread(t1)
Thread thread2 = new Thread(t2)

thread1.start();
thread2.start();
Thread.currentThread().sleep(10
```

```
public void run() {
  synchronized(lock2){
    Thread.currentThread().sleep(1

      synchronized(lock1){
```

```
public void run() {
  synchronized(lock1){
    Thread.currentThread().sleep(1

      synchronized(lock2){
```

Figure 3.12: Possible method of highlighting a program deadlock.

3.10 Interaction

It was realized early on that the amount of data we were trying to represent would require a scrolling view and that even with scrolling the design of Beat would require large screen sizes to be effective. We felt comfortable with this as large screens have become much more affordable.

Even with a larger screen there was still going to be a great deal of data so we considered a number of methods for dealing with this. The most basic was to allow resizing, rearranging and hiding of columns to organize the data better. Another option was to expand or contract columns when the mouse was over it or clicked on the column.

A final option was to provide improved filtering to select which methods of an object and maybe even which branches and loops to display.

To provide the macro/micro or context+focus we considered creating a zoomable interface that displayed more or less information as the user zoomed in or out. For example, when zoomed out the interface would only display method names and no source code, but when zoomed in it would show source code or state information.

In the next section we will discuss how we implemented the design and the choices and compromises that were made.

Chapter 4

Implementation

4.1 Implementation Choices

Based on the choices we had made for our design we knew there were a number of pieces of functionality we would have to implement to create the software:

- A graphical user interface to allow the user to generate the visualization from code they had written.
- A component to gather data from the user's running program.
- An infrastructure component that ran the program and processed the resulting data.
- A display component that shows the visualization.

To create these components we had to consider a number of different options such as what language we should visualize, whether to create a standalone application or integrate with an existing application such as an IDE, how to gather the low level data needed by our design and what display technology to use to generate the visualization.

In this section we will discuss the choices we made and then provide some detail about how Beat is implemented.

4.1.1 Language

We had to choose both a language to implement our design and a language to target for visualization. It was decided that using the same language for both would be the simplest approach.

A number of criteria were set for the implementation and target language. The language had to be something we were very familiar with, including details about how to gather runtime data from it. It had to meet our goal of being useful to the average programmer, so a language that had widespread familiarity was important. Since a language doesn't stand alone it had to have good support from tools, documentation and libraries. Built-in support for concurrency would be a plus.

We had extensive experience with the Java [4] and Ruby [21] languages and understood in general how to approach data gathering in each. Java is extremely well known by our target audience, much more so than Ruby which is still relatively obscure. Ruby has a small number of useful libraries, tool support for low level manipulation of objects and functions, reasonable documentation, but limited Integrated Development Environment (IDE) support. Java has a vast array of libraries, good tools support for parsing and manipulating the language, generally excellent documentation (although some areas can be difficult to find), and excellent IDE support particularly from the Eclipse [2] and Netbeans [25] IDEs.

Java and Ruby both have downsides in that both have relatively complex syntax and as a result can be complex to parse, which is necessary to perform instrumentation to add the data gathering probes, as discussed in section 2.2.6. In contrast, languages like LISP and Smalltalk have simpler syntax, but are probably even more obscure to a general programming audience than the Ruby language.

In the end all these reasons meant that Java was the best choice to meet our goals.

4.1.2 Standalone program versus IDE

We had a choice in how we wanted to implement our design, either as a standalone application or as a plugin of another program such as an IDE. Our goal of being useful to programmers meant that we wanted it to be available on all platforms (or at least the ones which have a Java Virtual Machine (JVM) implementation).

An advantage of implementing a standalone application would be that it would give us complete control over how the application worked. However, there were many disadvantages to writing a standalone application. It meant writing a lot of boilerplate or "plumbing" code that was necessary for an application, but irrelevant to the actual task of implementing our design. Toolkits for writing GUIs can be fraught with difficulty and incompatibility when creating cross-platform applications, although toolkits such as QT or Java's Swing library have reduced this somewhat.

The advantages of using an IDE is that we could avoid writing a lot of the boilerplate code for the GUI and also for things such as support for running and monitoring an executing program. IDEs also tend to be familiar to a wide range of programmers, which helps meet our goal of being useful to a wide range of programmers. Finally, IDEs such as Eclipse are cross-platform, which largely solves the problem of portability for us. One disadvantage of IDEs is that the process of developing a plugin can be complex and sometimes lacking documentation.

Given the advantages of IDEs we decided to use the Eclipse IDE which we were familiar with.

4.1.3 Gathering Trace Data

The most difficult implementation choice for Beat was deciding how we would gather data from the running concurrent program. From our design it was obvious that we would need to gather a great deal of information including low level information such as loops and if/switch statements, but it wasn't immediately obvious exactly how to go about this.

There were two possible directions to take, one was to use the debugging system that is built into the JVM and simply receive debugging events from the running program. The other was to instrument the source code or byte code of the program with additional "probe" methods that record information from the running Java program.

The Java Platform Debug Architecture [16] (JPDA) comes with the JVM and is used to implement debuggers for the Java language. JPDA is composed of three parts: Java Virtual Machine Tool Interface [17] (JVMTI), which is part of the JVM and allows C code modules to be written which interact with the JVM at runtime to gather data such as method entries and exits, and The Java Debug Wire Protocol (JDWP) which defines the format for sending events from the JVM to a remote debugger. Finally, the Java Debug Interface (JDI) is a set of Java libraries for implementing Java debuggers in Java. The JIVE system mentioned in section 2.4.6 uses the JPDA to gather data for its visualizations.

Implementing event receiving was relatively straight-forward and just required the implementation of a number of classes and the use of some of the classes that were part of the JDI. Listing 4 shows the program to be debugged being launched using some of the built in classes for launching programs. Listing 5 shows registering for method entry events. Listing 6 shows receiving events from an event queue and passing them to a handling function.

The approach based on the JVMTI didn't work out for three reasons. JVMTI doesn't

```
// mainArgs contains a command to launch the program to be debugged
void launchTarget(String mainArgs) {
    List connectors = Bootstrap.virtualMachineManager().allConnectors();
    Iterator iter = connectors.iterator();

    LaunchingConnector commandLineConnector

    while (iter.hasNext()) {
        Connector connector = (Connector) iter.next();
        if (connector.name().equals("com.sun.jdi.CommandLineLaunch")) {
            commandLineConnector =  (LaunchingConnector)connector;
        }
    }

    Map arguments = connectorArguments(connector, mainArgs);

    VirtualMachine =  connector.launch(arguments);
}
```

Listing 4: This example shows how to launch a program using the JDI library [15].

```
public void requestEvents(VirtualMachine vm) {
    EventRequestManager mgr = vm.eventRequestManager();

    // register for method events
    MethodEntryRequest menr = mgr.createMethodEntryRequest();
    menr.setSuspendPolicy(EventRequest.SUSPEND_NONE);
    menr.enable();
}
```

Listing 5: Registering to receive method entry events from remote program [15].

```java
void run(VirtualMachine vm) {
    EventQueue queue = vm.eventQueue();
    while (connected) {
        try {
            EventSet eventSet = queue.remove();
            EventIterator it = eventSet.eventIterator();

            while (it.hasNext()) {
                handleEvent(it.nextEvent());
            }

            eventSet.resume();
        } catch (InterruptedException exc) {
            // Ignore
        } catch (VMDisconnectedException discExc) {
            handleDisconnectedException();
            break;
        }
    }
}
```

Listing 6: Method to receive registered events from the debugger [15].

allow monitoring of low level events like loops, branches and exceptions, which were crucial to our design. Secondly, JVMTI offers no way to tie a method call to a specific object without stopping the JVM and analyzing the stack to find what instance of the object had had the method call executed on it, stopping the JVM to get this information caused serious performance problems. Finally, there was no way to uniquely identify an object instance across the entire execution of a program, since object ID in Java isn't guaranteed to be unique.

For these reasons we chose to try the second approach based on instrumenting code. Our first approach to instrumenting the code was to use the tools provided by the AspectJ [10] Aspect oriented programming toolkit. Aspect oriented programing is designed to deal with "cross cutting concerns" in a program, these are "concerns" or pieces of functionality that are part of a wide range of a system's modules, but are not directly related to any one module. A common "cross cutting concern" is providing logging for a program without needing to scatter calls to logging methods throughout, the source code which was similar to what we were trying to do.

In aspect oriented programming you provide pointcuts, which are points within the source code at which to attach "advice" to method calls or method bodies. Basically, this allows you to add extra code to methods before runtime to provide various information about when a method is entered or exited. Pointcuts are created using a Java like syntax, as shown in Listing 7.

AspectJ suffered from the same problem as the JVMTI – we couldn't gather the low level details about loops, branches and exceptions. Additionally, AspectJ didn't properly support the synchronized keyword that is part of the Java language.

To fix these problems we used the ANTLR [7] parser generator tool and an existing Java grammar that comes with the ANTLR software. This works somewhat like a filter, the source code is passed from input to output, with additional methods being inserted into the code at important points. A sample of this can be seen in Listing 8, which shows a small part of an ANTLR grammar file with statements to output code included.

Although the combination of AspectJ and ANTLR worked, it was prone to breaking and was very complex to extend, making it difficult to add extra features without problems.

While working on other parts of the implementation we came across a better solution to this problem. We found that part of the Java Development Toolkit [1] (JDT), which is the part of Eclipse that implements the various parts of the IDE that allow Java programs to be written, run and debugged, has a built-in lexer and parser that can parse Java source to an abstract syntax tree that can then be examined, modified and

```
public void aspectProbe(EventType event, JoinPoint joinPoint){
    // code to create probe event goes here
}

// only add pointcuts within objects that have the
// @BeatTrace annotation
pointcut tracedObject() : within(@BeatTrace *);

// point cut for methods
pointcut method() : tracedObject() &&
   !execution(public void Runnable.run()) &&
   execution(!synchronized !static * *());

// add probe before method call
before() : method() {
    aspectProbe(EventType.methodStart, thisJoinPoint);
}

// add probe after method call
after() : method(){
    aspectProbe(EventType.methodEnd, thisJoinPoint);
}
```

Listing 7: Probing methods using AspectJ pointcuts.

```
statement
  : block
  |  ASSERT {a(" assert ");} expression (':' {a(":");}
       expression)? ';' {a(";");}

  |  'if' {a(" if ");} parExpression
     {
       probeStack.add("beat.collector.EventType.ifStatement");}
       statement {probeStack.removeLast();}
       (
       options {k=1;}:'else' {a(" else ");}
        {
          probeStack.add("beat.collector.EventType.elseStatement");}
          statement {probeStack.removeLast();
        }
      )?

  |  'for' {a(" for ");} '(' {a("(");}  forControl ')' {a(")");}
      {
         probeStack.add("beat.collector.EventType.forLoop");}
         statement {probeStack.removeLast();
      }

  |   'try' {a(" try ");} block
      ( catches 'finally' {a(" finally ");} block
      | catches
      |   'finally' {a(" finally ");} block
      )

  |  'switch' {a(" switch ");} parExpression '{' {a("{");}
       switchBlockStatementGroups '}' {a("}");}

  |  'synchronized' {a(" synchronized ");} parExpression block

  |  'return' {a(" return ");} expression? ';' {a(";");}

  |  statementExpression ';' {a(";");}

  |  id3=Identifier {a($id3.text);} ':' {a(":");}  statement
  ;
```
.

Listing 8: Example of part of an ANTLR grammar used for processing a block of Java code to add tracing probes.

written back to a source file.

A simple recursive descent parser was used on the abstract syntax tree to examine the details of classes in the system and add method calls to probe methods by simply adding nodes to the AST before writing it out to a separate source file. More information about this process will be given in section 4.3.

4.1.4 Visualization display choices

The decision to use an IDE constrained our choices about what display technology to use; we investigated three main approaches to solving this problem.

Eclipse uses SWT as its native toolkit for creating widgets like buttons and text fields. Although we investigated this a bit we were concerned that all our time would be spent writing low level code for drawing lines and positioning text. Additionally we knew of a library called the Graphical Editing Framework (GEF) which was designed for creating things such as UML editors and "boxes and wires" visual languages, which seemed more relevant to what we were trying to do.

The GEF is essentially a Model View Controller framework for representing and modifying an applications model/data objects. To actually draw the visualizations the GEF relies on a 2D drawing framework called Draw2D. Since the GEF was primarily oriented towards creating "boxes and wires" visual languages, which wasn't relevant to what we were trying to achieve, we decided to try to use Draw2D independently to create our visualization. Draw2D is certainly capable of creating our design, however we found the documentation so poor that it was difficult to understand how to go about implementing our design and after making little headway we decided to try a different approach.

Using HTML embedded in a browser may seem like a strange choice, however it has a number of advantages. HTML has good support for text and with the canvas object drawing vector images included with modern web browsers would provide the support we needed to create our design. Probably the most important thing that influenced our decision was our experience with HTML and developing complex graphical and Javascript- driven pages.

4.2 Plugin Infrastructure

Plugin Infrastructure is a catch-all term for the parts of the plugin that coordinate the data gathering, data processing and visualization display.

Developing a plugin for Eclipse is relatively straight-forward thanks to the Plugin Development Tools (PDT) that are available as part of the Eclipse "classic" distribution. The PDT provides plugin templates and simple tools to perform tasks such as running another copy of Eclipse with the plugin that is being created enabled, making testing and debugging relatively easily.

To extend and integrate with Eclipse there are a number of integration points that a plugin can plug into. This is done by implementing an interface and registering the plugin class by using an xml file to set that integration point of Eclipse to use the class that you create.

Beat uses four of these extension points. The first is a launcher extension, which Eclipse uses to launch programs for an IDE project (note: this is not necessarily a Java program). The second is an extension of the run configuration menu that is used to configure the launcher, we just use the options provided by the JDT for consistency and ease of use. Thirdly, we have a toolbar shortcut to run the program. Finally, we have a view extension which adds our visualization view to Eclipse which is simply a subclass of an Eclipse ViewPart with a child web browser object. Listing 9 shows the xml code for the launcher extension.

```xml
<extension
      point="org.eclipse.debug.core.launchConfigurationTypes">
   <launchConfigurationType
         delegate="beat.BeatLauncher"
         id="Beat.RunLauncher"
         modes="run"
         name="RunLauncher">
   </launchConfigurationType>
</extension>
```

Listing 9: Configuration of a launcher extension for Eclipse in the XML plugin configuration.

4.2.1 Launcher and the JDT

At the heart of the plugin infrastructure is the launcher, which extends Eclipse's built-in mechanism for launching programs. Rather than develop all our own code for running a Java program we simply extended the existing Java launcher that comes with the JDT. Since we had to make fairly extensive modifications to the launch behaviour we found the source code for the launch method and modified it extensively to provide the launch functionality. We also used the run menu configuration panels that are part of the JDT, these allow the configuration of a Java program by providing an interface that lets the user specify things like program arguments and environment variables.

A brief overview of the process that Beat goes through to load and run a program and produce a visualization is:

1. Verify the name of the class to be run and the working directory to run in.
2. Load the program and virtual machine arguments from the launching menu system.
3. Run the source code preprocessor.
4. Create an object to handle running the program.
5. Run the program.
6. Load the data from the program and render the HTML visualization.
7. Show the Beat visualization view and set the Browser.java instance to the generated html page.

4.3 Data Gathering

Gathering data to generate the Beat visualization involves a number of parts'. Selecting which Java classes to trace, adding instrumentation to the source code, recording data at runtime, loading the data and processing it into a form useful for display.

4.3.1 Trace data annotations

Tracing every single instance of every single object in a running Java program was deemed impractical due to the large amount of unnecessary data it would produce. This meant that a way of selecting which classes to include in the visualization was needed. There were two ways we considered to allow the user to select which classes to trace. The first approach was to use a GUI to select the classes from within the existing project, The second was to use Java's source code annotation mechanism.

We chose to use annotations mainly for the sake of expediency; by implementing it this way we could simply detect the annotations in our instrumentation code, rather than implementing additional GUI code passing the information from the GUI to the instrumentation processor. One possible problem with annotations is that to identify what class needs to be instrumented requires examining all the source code of a project for the annotation, which could be inefficient in large programs, GUIs wouldn't face this problem.

4.3.2 Source Code instrumentation

As mentioned above we chose to use the JDT to implement the source code instrumentation. The basic process is as follows:

1. Find the files that need to be checked for the annotation from the current project.
2. Check the files for the @BeatTrace annotation used to mark classes to be traced.
3. For each file, parse the source code into an abstract syntax tree.
4. Record the method names and types of each file being traced for use later on.
5. Process the source code of each of the methods of the files (this will be explained in more detail below).
6. Add the interface used to get a unique object identifier for the traced object to each file.
7. Write the modified abstract syntax trees out to a source file in a directory called preprocessor-src.
8. Compile the modified source using the project options.

The fifth step is done using a recursive descent parser that takes the block of code of each method and goes through every statement in a method recursing as necessary. In Java a statement can be things such as an assignment, for loop, if statement or another block. Listing 10 below gives a basic outline of the tree structure of a Java source file.

There are a number of different instrumentations that we perform on the code as shown below.

- Add a method to the class to return the object identifier for the instance of that class.
- Methods have a probe inserted at the start of the method.
- If statements are handled by inserting probes at the start of each branch and at the start of each else and else if block.

```
class
        method
                block
                        statement
                        statement
                        statement

        method
                block
                        statement
                        block
                                statement
                        statement
```

Listing 10: Diagram showing basic tree structure of Java source code. A classes is made up of methods which contain blocks of statements.

- Do, while and for loops are handled by inserting probe methods at the start and end of a loop.
- Synchronized blocks have a probe before the block is entered, after the block is entered and after the block is exited.
- Expression statements can include a wide variety of different expressions, most importantly method calls. Calls to methods are handled by inserting a probe before and after the method call.
- Return statements are handled by inserting a probe before the statement. Since not all methods have a return statement (i.e. void methods) we check if a method has a return statement and add a method exit probe at the end if it doesn't. This is actually a bug since a void method could have both return statements and the possibility of exiting a method at the end.
- Exceptions are handled by putting probes before throw statements and at the start of catch blocks.
- The main method of a program and the run method of a Thread or Runnable are handled slightly differently from regular methods. A major difference is that these methods are where a program or thread starts and are not called by another object. Because of this they have special probe names to distinguish them and special probe methods, which will be explained in the next section. Another problem is

that RuntimeExceptions (which are unrecoverable errors) can cause a program or thread to exit unconditionally, so these methods have their contents wrapped in a try, catch block to catch these exceptions and ensure that the data recorded for that thread is written out to disk.

To perform the actual instrumentation we use code like the code in Listing 11 which inserts a probe before and after a method call and produces output like Listing 12 which shows a method call being traced.

4.3.3 Runtime method probes

Data is recorded on a per thread basis using a thread local ArrayList.java object to store the probe data for that thread. We decided to record data like this in the hope that it would avoid excessive disk IO compared to just writing every event straight to disk. When the thread or program exits it writes the data out to a file named after the thread in a subdirectory of the Eclipse project directory called "beat_thread_data".

The probe method stores the following information; the event type, the random object id generated for that class, the name of the thread, the name of the class, the name of the method and finally the time that the event occurred.

4.3.4 Data loading and processing

To make the data useful for visualization we need to process the raw text data into a form that can be used to draw both the object columns and the thread lines. We decided to do this by taking the raw event data and creating a number of model objects representing different parts of the execution. The class diagram in Figure 4.1 shows the basic attributes and relationships between the different model objects that were used.

After the program is run the event data files are loaded and each line is processed to generate a RawEvent.java object that contains the basic information about an event.

The object and thread information that each event contains is used to create a single ThreadData.java object to represent the thread that the event is from and a single ObjectData.java object to represent the instance that the event is from. The ThreadData and ObjectData objects also have lists of the RawEvents and the RawEvent contains a reference to the ThreadData and ObjectData objects.

To make it easy to ensure that threads are separated correctly when two or more threads are executing in a single column, the data is also organized by thread and by object in a separate object called ThreadDataObject, which contains a list of RawEvent.java objects from that Thread and Object instance.

```
// processes a method invocation to check if we want to trace it.
private void processMethodInvocation(ListIterator x, String[] names,
  Expression expression, String className, String methodName,
  Statement statement) {
        if(names == null || names[0] == null)
                return;

        if(names[0].equals("threadStart")){
                processThreadStart((MethodInvocation) expression,
                                   className, methodName,  statement);
                return;
        }

        int lineNumber =  compilationUnit.getLineNumber(
                                expression.getStartPosition() +
                                expression.getLength());

        ExpressionStatement beforeExp = ast.newExpressionStatement(
                        makeProbe(names[0], className,
                            methodName, lineNumber));

        // add trace probe before method invocation statement
        x.previous();
        x.add(beforeExp);
        x.next();

        // add trace probe after method invocation if necessary
        if(names[1] != null){
                ExpressionStatement afterExp =
                            ast.newExpressionStatement(
                                makeProbe(names[1], className,
                                methodName, lineNumber));

                x.add(afterExp);
        }
}
```

Listing 11: Code to insert method probes around a method call. The lines containing ExpressionStatement are the code for the methods to insert. The lines like x.add(beforeExp) add the method probes.

```
beat.collector.TimestampCollector.probe(
                beat.collector.EventType.methodCall,
                ((beat.collector.ObjectId) this).getObjectId(),
                    Thread.currentThread().getName(),
                    Producer.class.getName(), "run", 17);
data.put(i);
beat.collector.TimestampCollector.probe(
                beat.collector.EventType.methodCallExit,
                ((beat.collector.ObjectId) this).getObjectId(),
                    Thread.currentThread().getName(),
                    Producer.class.getName(), "run", 17);
```

Listing 12: Diagram showing the result of inserting probe methods before and after method call.

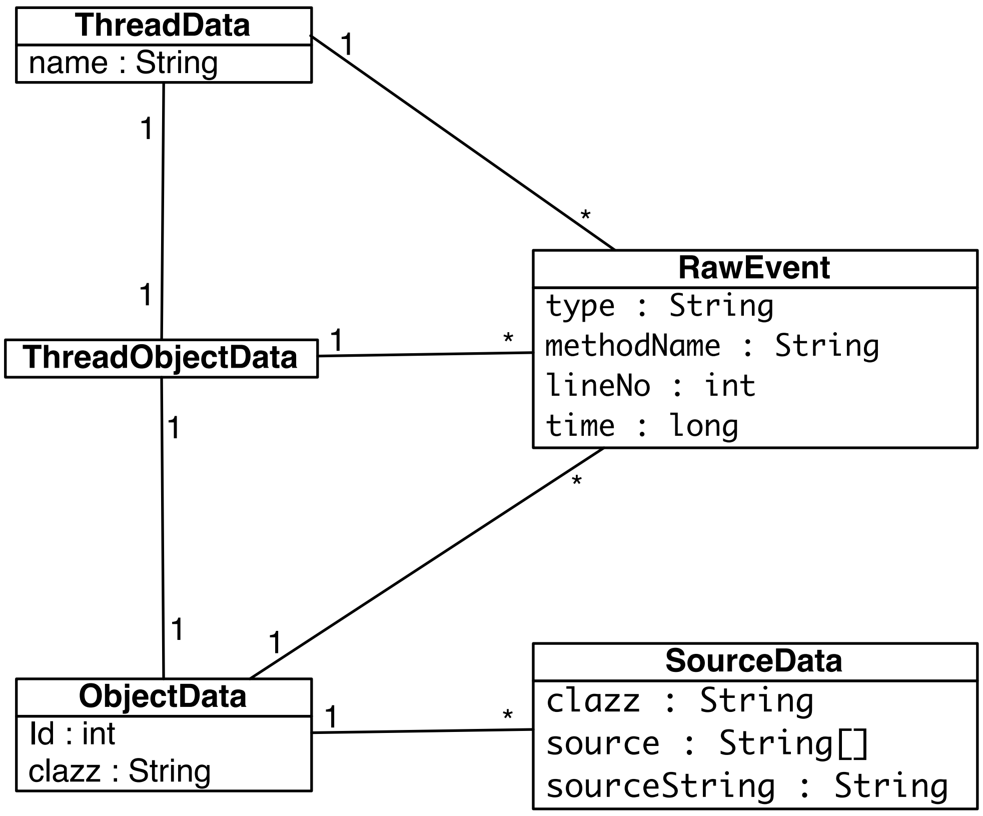


Figure 4.1: Class diagram of the relationship between the various model objects. These objects are how we structure the results we receive from the running program to use for processing and displaying the results.

After this, the raw events for each type are sorted by the timestamp and some adjustments to the data are made. A thread start event is added to a newly created thread at the point in time that Thread.start() is called, since a thread may not be scheduled to run as soon as Thread.start() is called. There were many more minor adjustments made to the data to ensure precise positioning and to correct other problems with the raw data.

4.3.5 Context Switches and DTrace

Our design included the visualization of context switches, unfortunately due to a lack of time and technical constraints we were unable to complete this aspect of data gathering, although we will give a basic outline of how we intended to do it and some challenges of implementing this.

A simple method is to simply to check if there are long gaps between events that aren't due to things such as method calls to other objects, the problem with this is that it isn't completely reliable and becomes more difficult on multiprocessor systems.

Another way to gather data about when context switches occur requires information from the thread scheduler, in the kernel of the operating system that the program is executing on. As we mentioned in section 2.2.6 DTrace is a tracing framework that allows the tracing of events within an operating system kernel. Listing 13 shows a couple of methods for probing a kernel scheduling function in the Mac OSX kernel.

An obvious limitation to DTrace is that it is only available on Solaris and Mac OSX, with similar tools available for Linux. Unfortunately nothing like this seems to be available for Windows, which limits the platforms that tracing context switches could be performed on.

While working on this we found a number of technical problems. We found that we had to find a unique thread identifier available in both kernel space and user space. Thread ID in Java doesn't correspond to kernel thread identifiers, so to solve this a kernel extension that passes a kernel internal ID back to user space was necessary.

Another technical limitation is that we can't get the exact line at which the context switch occurred. It could possibly be done by analyzing the stream of byte codes that make up the program and looking for debugging byte codes indicating a line number.

Finally, support on Mac OSX was incomplete particularly, in the area of support for tracing the scheduler. Support on Solaris seems better, but we lacked the time to investigate this fully.

```
fbt::thread_timer_event:entry
/pid == $target/
{
        self->swout = (uint32_t)curthread;
        /* uint64_t tstamp sits in arg0:arg1.
        timer_t new_timer is arg2! */
        self->swin = ((uint32_t)arg2 -
           offsetof(struct thread, system_timer));
}

fbt::thread_timer_event:entry
/self->swout != self->swin && pid == $target/
{
        printf("%d %d %u\n", self->swout, self->swin, walltimestamp);
        self->swin = 0;
        self->swout = 0;
}
```

Listing 13: Probing the entry of a scheduling function in the Mac OSX kernel.

Without detection of context switches, visual anomalies are produced in the visualizations - threads will appear to be executing when in fact the thread is switched off the processor. With additional time and better support for DTrace on our targeted platforms, implementing visualizing context switches should be feasible.

4.3.6 Program State and Concurrency errors

Due to time limitations we were unable to complete the tracing of program state or the detection of concurrency errors. This had some consequences for how effective our software was, as we will see in the next chapter.

4.4 Visualization Generation

4.4.1 Ruby and Templating

Because we weren't entirely sure how we were going to generate the HTML file for display we decided to use the JRuby language to help us while prototyping. We did this for a couple of reasons. When working with HTML you are mostly working with text, which JRuby is very good at. Also, we wanted something flexible so that we could change things easily before settling on an approach, which would have required more work in

Java. In the longer term the Ruby prototyping code will be removed and replaced with pure Java code to generate the template and other parts of the code.

As mentioned above, a template is used to generate an HTML file for display. The template is an ERB template, which is the standard templating system that comes with the JRuby language. Like PHP or JSP, ERB allows code snippets to be embedded in a text file that get run when the template is rendered. A short JRuby script is run from Eclipse to render the template. The script also performs the syntax highlighting using the CodeRay syntax library.

Data is passed to the script through setting variables which the JRuby interpreter makes available to the script. Aside from the model objects such as RawEvent and ThreadObjectData mentioned above, other information such as the source code for each object, various bits of positioning information and a helper object that provides various drawing functions are included.

The template simply loops through the list of objects and generates the object columns by checking the first and the previous event and seeing if they are part of a block of source code and outputting an HTML div element and placing the generated source inside. The positioning of a code block is stored inside the RawEvent and is generated using the algorithms discussed in the next section. Listing 14 shows part of the templating code.

4.4.2 Text Positioning

One of the things we discovered early on was that there was either a large amount of space where nothing was happening in a thread or method boxes were too small or large for the source code. We created an algorithm that would remove blank space while maintaining the overall order of events relative to the previous and subsequent events, as well as ensuring that the boxes for text would be large enough to contain the text. Figure 4.2 shows an overview of the results of the time compression.

Another problem was ensuring that when two threads were executing inside a single thread that the code boxes for threads were placed side-by-side, not overlapping. An algorithm was created to space the threads correctly when two or more were threads were executing at the same time. A challenge with this was to ensure that a thread stayed in the same horizontal position if there were several other events in another thread in between. Figure 4.3 gives an example of this.

```
<div id="visualization">
  <div id="thread-colors">
    <%= threadColorBox %>
  </div>
  <% viewHelper.getObjectOrder.each do |object| %>
    <div id="<%= object.oid %>"
         class="object-column object-column-<%= object.oid %>"
         style="height: <%= headerSpace + columnHeight + 50 %>px">
      <div class="object-header">
        <%= object.clazz %><br />
        <span><%= object.oid < 0 ? "static" : object.oid %></span>
      </div>

      <div class="column-header"></div>
      <% viewHelper.getTODS(object).each do |tod| %>
        <% (1..tod.events.size()-1).each do |n| %>
          <div
               class="<%= eventClass(previous, event, inSync)%>"
               style="<%= eventStyle(previous, event, headerSpace)%>">

            <pre class="code">
              <% removeIndent(previous, event).each do |line| %>
                <%= line %><%= "\n" %>
              <% end %>
            </pre>

            <%= eventDetails(previous, event)%>
          </div>
        <% end %>
      <% end %>
    </div>
  <% end %>
</div>
```

Listing 14: Snippet of ERB templating code used to create visualization. The “<%” elements are used to insert data into the template either directly from the data structure shown above or further processed through functions that are called from the template such as “eventClass”.

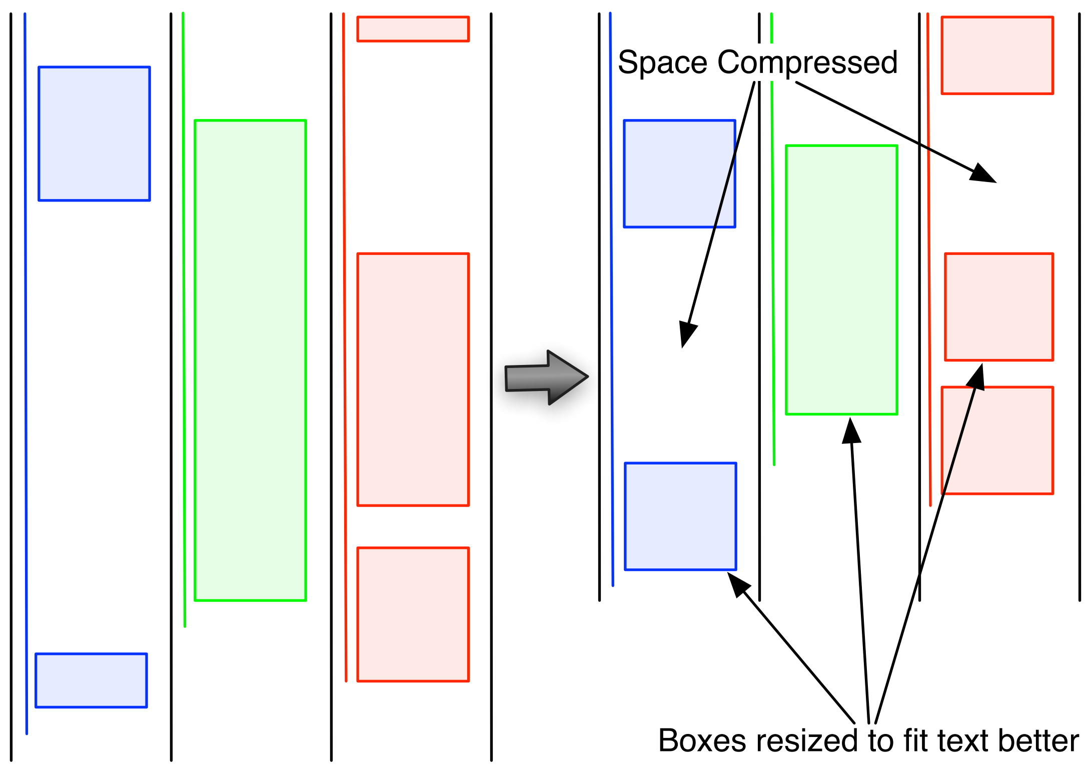


Figure 4.2: Method blocks get unnecessary space between them removed while remaining order while also ensuring that the blocks fit the source code text.

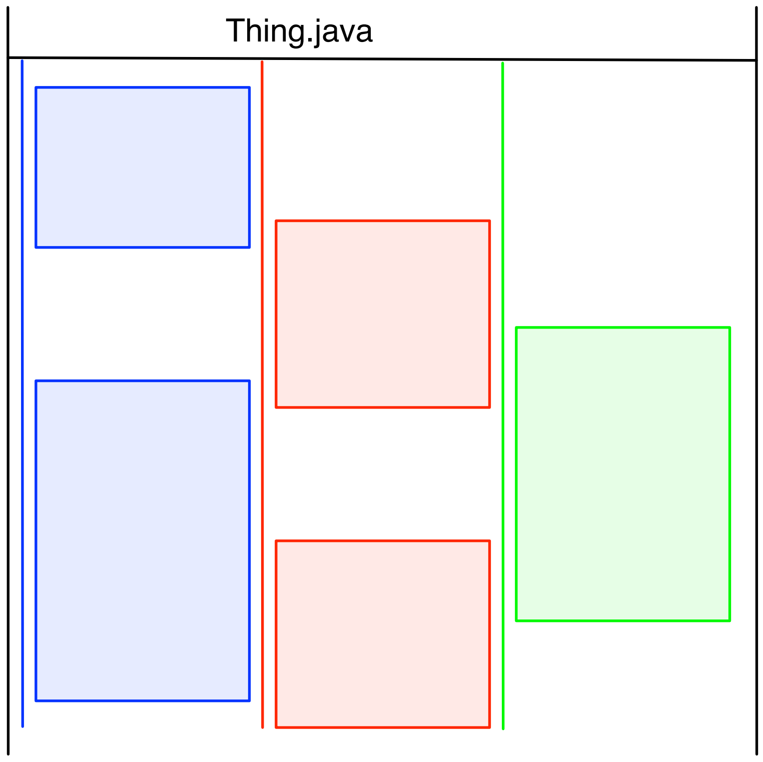


Figure 4.3: When multiple threads are executing within an instance once Beat ensures that the method call blocks remain in line horizontally.

4.4.3 Canvas Object and Threads

The thread lines for our application were drawn using the canvas tag that is part of the HTML5 standard [3]. The canvas tag allows the creation of vector graphics such as lines and boxes similar to other vector drawing software packages. A variety of Javascript functions are used to create allow lines, shapes, fills and other vector elements. Listing 15 has an example of some of the code used for drawing thread lines in Beat.

```
ctx = canvas.getContext('2d');

ctx.clearRect(0,0, width, height);

if(type == "notify" || type == "thrown" || type == "exception"){
    drawIcon(ctx, line[0][0], line[0][1], type, color);
    ctx.moveTo(line[0][0], line[0][1] + icon_height+2);
}else{
    ctx.moveTo(line[0][0], line[0][1]);
}

ctx.lineTo(line[1][0], line[1][1]);

ctx.stroke();

drawThreadName(ctx, color, line, name, y, height);

if(end_type == "exit" || end_type == "threadDeathException"){
    drawIcon(ctx, line[1][0], line[1][1], end_type, color);
}
```

Listing 15: Snippet of line drawing code in Javascript using the canvas tag. The context is an object with methods to draw on the canvas.

In the beginning we tried creating a single large canvas object that covered the entire visualization, but this caused significant performance problems so we abandoned this approach. To draw the lines we settled on updating the position and the contents of the canvas object every time the user scrolled or resized the browser view. This also meant that we had to redraw the lines with new data on every change.

The browser object allows functions to be registered which can be called by JavaScripts running in the browser. To implement a function you simply create a Java class and register it with the browser object. To update the canvas, handlers for the HTML resize and scroll events were created, which called the Java functions to get new data. The

Java functions would work out what data was necessary for the canvas to display then pass it to the canvas drawing function.

4.4.4 JQuery and CSS

To provide features such as column reordering and resizing we used the JQuery library [18]. JQuery is a simple library that makes writing code that manipulates HTML documents using Javascript easy.

The styling and colouring of the template has three parts. The first is a simple static css file loaded by the browser when the page is loaded. This file contained all the static information and basic styles, such as basic positioning.

The second part are the styles for positioning the source code boxes inside the columns, these are applied directly to the div elements that contain the source code.

The third part are styles that are specific to a thread, basically things like the colours of the code boxes. These are generated dynamically as an inline style sheet inside the template, each time the visualization is run.

Chapter 5

Evaluation

5.1 Evaluation Introduction

Ideally, we would have conducted a study to compare our software with users using software that was "non-integrationist" or used separate windows for data, but given our limited time and resource this was deemed impractical. Instead, we decided to conduct a smaller pilot study testing how useful Beat was compared with just using program source code. This would help us continue the development of Beat, which we would test against other systems in the future.

5.2 Evaluation Method

Our test involved participants solving two simple concurrent programming problems, one using Beat and one with just the source code. Participants would be timed performing the tasks and answer a short evaluation questionnaire giving their impressions of it. Appendix A contains the full questionnaire.

We chose to test our software with 300-level and post-grad students, because they were more likely to have experience with concurrent programming and because we didn't have access to a large enough sample of working concurrent programmers. Participants were found by placing advertisements around the computer science department at Massey University in Palmerston North and by approaching students that we knew were 300-level or post-grad. We were only able to test 20 participants in total, which should be kept in mind when we review the results of the test.

As part of the test-taking participants were given help by the test giver if requested, we felt this was appropriate given that, like music software, we expect the tool to not

Group	Task 1	Task 2
1	Deadlock With Beat	Race Condition With Source
2	Race Condition With Beat	Deadlock With Source
3	Deadlock With Source	Race Condition With Beat
4	Race Condition With Source	Deadlock With Beat

Table 5.1: Table showing the different groups and the tasks they performed.

be used in isolation from an instructor.

5.2.1 Tasks and Timing

The two tasks were simple find-the-concurrency-bug tests where the participants simply had to point out how to fix the bug in the program. The first task was a program with a simple race condition bug where an object was shared between two threads. The second task was a simple deadlocked program where the order of two locks was reversed. Appendix B contains the source code of the two examples.

To perform the test participants were divided into 4 groups as follows as shown in Table 5.1. Four groups with different orders of performing the two tasks were used to ensure that the order of the tasks and the order that Beat was used in didn't effect our results.

5.2.2 Previous Experience

Our questionnaire had a number of questions related to the participants previous experience.

- Have you have taken 159.355 Concurrent Systems or another concurrency related paper?
  Yes / No

- How would you rate your level of experience with concurrency and knowledge of concurrency issues such as race conditions and deadlocks?
  Low 1 2 3 4 5 High

- How much experience do you feel you have with using the Eclipse IDE?
  Low 1 2 3 4 5 High

Also we had a number of questions related to users subjective experience of using Beat.

- Which task was easier?
  Race Condition / Beat / About the same
- Did Beat help you solve the task that you used it for?
  Yes / No
- Do you think Beat would have been useful for solving the other task?
  Yes / No

5.2.3 Timing and Experience Questions

Timing participants and asking questions about their level of experience was designed to address a number of subquestions about Beat to help answer the broader question of how effective it was:

- Was using Beat significantly faster?
- Was Beat helpful in one not the other?
- Was using Beat significantly easier?
- What effect did experience have?
- Did the order that Beat was used in have any effect?
- Was one of the tasks harder than the other?

5.2.4 UI Questions and Comments

There were also a number of specific questions we wanted to answer about parts of our visualization which weren't covered by the timing data. These generally related to parts of our visualization that were particularly novel.

- Was the software user interface responsive enough?
- Were the visualizations produced easy to relate to the source code of the example program?
- Did the layout of the visualization and the use of color and space make it clear how the program was working?
- Do you feel that the inclusion of source code in the diagram helped you to relate to how the test program was executing to your source code?

Test	**DeadLock**	**Race**
P-Value	0.0173	0.0029
P-Value $>$ 0.05	False	False
Mean With Beat	6.7min	6min
Mean With Source	3.8min	1.6min
Standard Deviation With Beat	3.917min	2.944min
Standard Deviation With Source	2.700min	1.506min

Table 5.2: Comparison data on the underlying distribution and averages of the two tasks.

Finally, we wanted to gather any comments about the software, both positive and negative, and also to make observations about participants when taking the test.

- Please comment on any things you liked about the software.
- Please comment on anything you disliked about the software.
- Do you have any recommendations for future changes?

5.3 Timing and Quantitative Questions Results

Full results are available in Appendix C.

5.3.1 Was using Beat significantly faster?

To compare whether using Beat was significantly faster we compared the timing data of each of the tasks using Beat to the corresponding task using just the source, without reference to the order of the tasks. The comparison was made using the mean time of each task.

To ensure that the distributions of the two tasks were different we used the common statistical function the t-test which tells whether data from two tests is distinct or from the same underlying distribution. We made the assumption that the underlying population was normally distributed and acknowledge that our small sample size could lead to bias in our results, in spite of these limitations we feel the t-test was the best option available for performing the comparison. The results can be seen in Table 5.2.

The results of the table show that on average using Beat was slower than using just the source. They also show that the data from using Beat and just using the source were from separate distributions, showing that the comparison is valid. It isn't unexpected that Beat took longer since as we mentioned in our discussion of software of this sort of software is designed to be used by experienced users and we didn't have time to let users become experienced enough to conduct a truly effective test.

5.3.2 Was Beat helpful in one not the other?

As can be seen in Table 5.2, using Beat took longer than using the source alone, although it seems that in the case of the deadlock task the difference in performance was less compared to the race condition task.

5.3.3 Was using Beat significantly easier?

The results in Table 5.2 suggest that Beat wasn't easier to use since it took participants longer to complete the task using Beat. However, a majority of users did report in the questionnaire that Beat helped them solve the task they used it for and felt it would help with the other task if they had been using it for that, as shown below:

- Did Beat help you solve the task that you used it for?
  15 / 19 answered Yes
- Do you think Beat would have been useful for solving the other task?
  18 / 19 answered Yes

5.3.4 What effect did experience have?

To check whether experience had any effect we compared participants' reported experience levels to the time taken to perform the task. We used Pearson's correlation in the form of the Excel CORREL function to check if there was any relationship between the two variables. Pearson's correlation produces a value between -1 and 1, -1 indicating a perfect negative linear correlation, 1 indicating a perfect positive linear correlation and 0 indicating no correlation. Table 5.3 shows the correlation values for the different tasks and concurrency and Eclipse experience levels.

The results indicate a small negative correlation between concurrency experience and the tasks, in other words those with more experience in concurrency performed the task in a shorter time. Experience with Eclipse only seems to have effected the time taken to complete the Race Condition test and doesn't seem to have effected the time taken to complete the Deadlock test.

Because only one participant had never taken the a concurrency course at Massey University or elsewhere we considered it impossible to make a valid comparison based

Comparison	**Correlation**
Concurrency vs Deadlock	-0.401
Concurrency vs Race	-0.321
Eclipse vs Deadlock	0.082
Eclipse vs Race	-0.315

Table 5.3: Correlations between difference experience levels in concurrency and Eclipse and time taken to complete a task.

Task	**Mean First**	**Mean Second**	**Std. Dev. First**	**Std. Dev. Second**
Deadlock With Beat	9.5min	5.8min	1.0min	3.701min
Race With Beat	8.0min	4.0min	1.225min	2.828min
Deadlock With Source	3.0min	4.6min	1.414min	3.578min
Race With Source	1.8min	1.4min	1.789min	1.342min

Table 5.4: Comparison of using Beat first and second and using Source first and second.

on whether experience in a formal educational environment had an effect on the time taken.

5.3.5 Did the order that Beat was used in have any effect?

We had tested using Beat both first and second to test whether the order that Beat is used had an effect on the data, the results are in Table 5.4.

As the results show the order that Beat is used does seem to have an effect on the outcome of the test, with those who used it second showing better average times. The order that the source was used in doesn't seem to have had as great an effect on the times for using the source as it does when using Beat.

5.3.6 Was one of the tasks harder than the other?

To check if one task was harder than the others we compared the mean of the times taken to complete the two tasks without reference to the use of Beat or the order of the tasks. The results are in Table 5.5 and show that the deadlock task took slightly longer to complete on average.

We also asked participants what task was easier, the results are shown in Table 5.6. Again, the deadlock task seems to be slightly harder than the race condition task.

Task	**Mean**	**Standard Deviation**
Deadlock	5.526min	3.470min
Race	4.0min	3.162min

Table 5.5: Mean times for the two tasks without reference to use of Beat.

Task	**Count**
Race	10
Deadlock	5
Same	4

Table 5.6: Participants rating of what was task was easiest.

5.4 Visualization Design Questionnaire Results

- Was the software user interface responsive enough?
  19 / 20 answered Yes
- Were the visualizations produced easy to relate to the source code of the example program?
  18 / 20 answered Yes
- Did the layout of the visualization and the use of colour and space make it clear how the program was working?
  17 / 20 answered Yes
- Do you feel that the inclusion of source code in the diagram helped you to relate to how the test program was executing to your source code?
  19 / 19 answered Yes

The results show that participants felt that the user interface was responsive enough and that the layout, colour and use of space was adequate. Participants also felt that the inclusion of source code was useful, which is encouraging given that the addition of source code is one of the novel parts of our design.

5.5 Comments From Questionnaries

In this section we will review some of common comments made about Beat and provide a brief discussion of them. We have divided it up into sections on what people liked, minor and UI problems and major dislikes and additional features.

5.5.1 Like Comments

In general participants liked the colour and layout, reordering and expanding of lanes and generally found the approach to displaying threads and objects useful as indicated above in the questionnaire results and in many of the comments such as:

> *"It's clarity (good contrast, soft colouring). The ability to reasize and reorder the threads. The rubber banding while modifying. The timeline approach to visualize the thread that are currently executing and the transfer of control"*
> - Participant 7
>
> *"A great tool for demonstrating concurrency. Representation is very clear."*
> - Participant 9
>
> *"Logical layout of objects, time, code and process life"* - Participant 14

5.5.2 Minor UI problems

> *"A method of disabling a selected thread."* - Participant 7

We should also have methods for easily disabling object instances as well.

> *"Thread key hides some col info."* - Participant 12

> *"Legend for the symbols used in Beat."* - Participant 9

Beat needs a separate area for meta information like this.

> *"order the columns in a more intuitive manner"* - Participant 2

This probably means order things automatically, which may be possible but not easy. Adding a feature that maintains column order between runs would be relatively easy however.

> *"Some codes are cut (column width for Threads' codes are not wide enough)"*
> - Participant 9

This could be addressed by making a column expand on hover or simply clicking on it.

> *"Use of Lock, Semaphores etc."* - Participant 16

It would be useful to handle other concurrency primitives that are part of Java.

5.5.3 Major Dislikes/ Needed Features

> *"A method of highlighting changes to a selected variable or variables."* - Participant 7

Adding some type of visualization of state is one of the most important things that was left out of the implementation. Also, the lack of a method to highlight concurrency errors is a crucial inadequacy of the software.

> *"Debugger integration, step through a program."* - Participant 17

A common demand was to integrate Beat with the debugger to allow stepping through the program e.g. "Debugger Mode". This was a good point and one we will expand on in the Future Work section of our Conclusion.

> *"having the line of code twice is a little confusing."* - Participant 17

Due to the fact that method calls can sometimes span large amounts of time due to calling other methods, threads sleeping or threads waiting we included the text of the method calls after the return from the called method. This was intended to help the user follow what was happening, however it seems to just generate confusion, so it will be removed in future versions. Some other type of indication of a returning method could possibly be developed.

> *"Visualization: too much information present."* - Participant 17
>
> *"filters for content, being able to restrict the visualization."* - Participant 17

The implementation of the interactive features mentioned in Section 3.10 would help solve this problem.

5.6 Observations of Participants

We made a few general notes about test participants during the testing. One thing we noted was that to understand the visualizations produced you already had to have a good understanding of concurrent programming. Participants, even ones who we knew had recently completed a course on concurrent programming, had difficulty identifying a deadlock and often needed additional guidance when solving the problems given.

Another important thing was that participants were often confused about the colouring of the thread lines - that they were coloured to differentiate them and that the colour has no meaning beyond the need to differentiate the threads. In future versions of the software it may be useful to allow users to select what colour they want for a thread or rather than using colour to differentiate threads use some other mechanism (maybe just names or icons) to avoid confusion. As mentioned in the discussion about saUML, colour could be used for another purpose such as to show thread state.

5.7 Evaluation Limitations

There were a number of problems, technical, procedural and structural with our testing.

The first technical limitation was that the built-in synchronized locks of Java are non-interruptible. This meant that when a deadlock occurs the only thing that can be done is to kill the program, which given the implementation of data collection meant that all the data up to the deadlock was lost. Because it was too late to fix this in the implementation code, a test for the deadlock was added to make sure that when a deadlock occurred the trace data was recorded automatically.

Another limitation to our design was that it lacked good support for identifying which object instance was being locked in synchronized blocks (not synchronized methods). To fix this, additional method calls to the lock object were added before the lock was acquired to try to indicate the order of locking to the participant in the test. In hindsight our deadlock example could have been better designed to take into account the limitations outlined above, in future we will create a better deadlock test.

There were also several problems with how the test was run. The first was that parts of the deadlock code were very confusing so they were changed after the first few participants to ensure that it was clear how the program worked. Earlier participants had been given more help by the test supervisor to help them understand the problem, less help was given after the changes.

Second was that the timing of the participants was to the nearest minute, which didn't give as much timing detail as we would have liked.

Chapter 6

Conclusion

6.1 Conclusion

We introduced our thesis by noting the great shift away from uniprocessor architectures towards multicore systems. This change has led to a great deal of effort being expended attempting to make programming these new systems easier for developers. To address this challenge visualization tools have been created, although they have only had limited success in the market. In contrast we noted that music notation and music software have had tremendous success in the market, in many cases entirely displacing older manual methods. This interested us because of several similarities between the two, most importantly that both dealt with time and that both used a timeline-based approach for visualization.

This led us to ask the following questions:

> *Is there something about existing visualizations that make them unsuitable for the task of programming and debugging concurrent programs?*
>
> *What is it about music notation and music recording software designs which make them effective?*
>
> *Can we draw any conclusions from existing visualizations, music notation and music editing software and apply the lessons learned from them to the design of a visualization based concurrent debugger?*
>
> *Will the resulting design be effective for the task of debugging concurrent programs?*

Our investigation of existing literature highlighted the problems for programmers that arise from concurrency and some of the existing non-visual solutions to these problems. A review of existing visualization and software visualization showed a field heavily focused on performance debugging and not the kind of error debugging that we would argue is a much more common task for programmers. The central insight that we drew when comparing software visualizations and music notation and technology was that existing software visualizations lacked a single highly integrated view of the data and instead relied on multiple separate subviews.

We feel our investigation of existing systems has answered our first two thesis questions, though we would note that without a full study comparing our software with one of the "non-integrated" pieces of software mentioned in the literature it is hard to give a definitive answer to whether this is because of the lack of integration between different kinds of data in the software.

In addition, it should be said that there are numerous reasons why visual debugging tools could have failed to have achieved wide use that have nothing inherently to do with the technology. This can include the fact that there is a wide gulf between the products of the academic world and the details and finesse needed for success in the market. More provocatively, it could be argued that there hasn't been enough of a pressing need in an economic sense to require tools like this, despite the fact that multiple cores have become a lot more common.

Applying the insights gained from studying existing systems to the design of a piece of software produced the design for Beat, which attempts to integrate the various different pieces of data that make up an executing concurrent program. In particular, this included low level data to assist programs debugging for error correction purposes.

This effectively answers our third thesis question, since our design combines the lessons we have learned from all the different systems we have investigated.

Implementing Beat was complex at times, as we had to evaluate a number of technologies and approaches, many of which seemed promising but ended up being dead ends in terms of our goals. We ended up with a system that used the Java language as its target, with the visualization being integrated in to the Eclipse environment. Instrumenting the source code was performed using part of the Java Development Tools library and using HTML to display the generated data.

Unfortunately,we didn't get to completely finish implementing our design, in particular the state tracing that would have been helpful for users during our testing.

The evaluation of our software suffered from a number of flaws, including some technical problems and problems with the way the test was run. Perhaps the greatest flaw was that we were testing software design for experienced operators in a test that was too short. This possibly makes our timing data less useful since it doesn't reflect a realistic usage scenario. Ultimately, the most useful part of our evaluation was the comments by users, which confirmed much of what we thought was right with our software and also what was wrong with it. Some future directions in testing will be discussed in the Future Work section below.

Overall, we feel we have struck on a new approach and that we are just at the beginning of investigating the ideas present in this thesis, however our present work gives us cause for optimism about the future direction of the work.

6.2 Future Work

Not surprisingly our work left us with more questions than answers and many ideas about possible directions with regards to the software. We will start by addressing some better methods of testing Beat, then some of the basic core and UI improvements that need to be made to complete Beat as designed and then address some possibilities for making Beat more useful.

6.2.1 Testing

There were two different future directions we considered for improving the testing of Beat. The first was a longer term study where participants would use the software several times over a period of time, with each time being longer than the 20 minutes or so we had for testing participants. This would be a more realistic test of Beat since Beat is designed for experienced operators who have been given more than a brief introduction and a short amount of time to become accustomed to the software.

Second it would be interesting to answer the question of whether an integrated data approach was better than a non-integrated data approach. This would require a comparative study comparing our system to an existing ”non-integrationist” system to answer this question.

6.2.2 Core Improvements

The most basic improvement is the addition of data gathering of state information which will allow the visualization of state information.

Currently Beat is deeply tied to the Java language, a possible future direction is to create a framework to enable other languages to be easily integrated into the software. There would be a number of challenges with this, for example how would we visualize languages that aren't object orientated such as C or Haskell? Another challenge would be languages that don't use shared memory and locks for concurrent communication such as Erlang.

Completion of gathering context switch information is important to complete Beat as it was designed. An open challenge is whether this could be done on Windows. While there are tools similar to DTrace on Linux we are not aware of any directly comparable tools for Windows. For tracing to work on Windows we would have to have a way of tracing the kernel scheduler, which might be difficult without documentation of the kernel scheduling functions or access to the source code.

We haven't fully tested Beat in the context of multicore and multiprocessor systems, more work needs to be done especially with machines with large numbers of chips, cores and large numbers of threads.

6.2.3 UI improvements

The completion of the design and implementation for visualizing state changes is probably the most important thing that needs to be implemented or improved. The second most important thing is the addition of some mechanism of filtering the results, such as hiding threads or object instances, specific methods and maybe state information for specific objects. Another solution to the problem of too much information would be the implementation of a zoomable interface, as mentioned in the design section

An interesting idea that we considered and which came up in our evaluation was the creation of an automatic system to order the columns in an intuitive manner. We developed a solution to this problem by trying to save the order of columns between executions so that the user would be presented with the same layout they had before, but we were unable to complete this.

One design aspect that wasn't considered was how to highlight concurrency errors that occur during and execution. This would require some improvements in the core software to help analyze the stream of events to find these errors.

HTML has served well as a mechanism for prototyping our design, however to achieve integration with the debugger will probably require us to rewrite the visualization using the SWT graphics library.

Another part of the UI that needs writing is a GUI to select classes and parts of classes such as method calls and variables, this ties in with the tools to filter output as well.

6.2.4 Debugger and Static Analysis

Adding debugging functionality to Beat or integration with an existing debugger is probably the most likely future direction of Beat. This could be done by simply using the existing debugging system of languages like Java or potentially a much deeper set of additions or modifications to the Java Debugging system. This seems to be an under-investigated area, with most frameworks being either for debugging or for tracing alone and not a combined or integrated approach.

As we mentioned, work has been done on using visualization to display possible interleavings of a concurrent program using information drawn from an analysis of the program. Beat would be excellent at displaying these interleavings and would be well suited to integration with these tools.

6.2.5 Education

We considered the use of Beat as part of a course on concurrency. Beat may be useful in this capacity, but more investigation of the needs of students learning concurrency needs to be done and testing of the software in an educational environment.

6.2.6 Visual Language

Developing a visual editing language based on the visualization principles underlying Beat is an interesting possibility. This would involve using the position of blocks of code to represent when events could occur in time. Unlike many other visual languages this would be implemented as an extension to an existing language rather than an entirely new language of the box and wires type.

This would be a completely new direction requiring a considerable amount of research and development to see whether it is feasible.

Appendix A

Questionnaire

Beat Evaluation Questionnaire

Q1. Have you have taken 159.355 Concurrent Systems or another concurrency related paper? Yes / No

Q2. How would you rate your level of experience with concurrency and knowledge of concurrency issues such as race conditions and deadlocks? Low 1 2 3 4 5 High

Q3. How much experience do you feel you have with using the Eclipse IDE? Low 1 2 3 4 5 High

Q4. Which task was easier? Race / Deadlock / About the same

Q5. Did Beat help you solve the task that you used it for? Yes / No

Q6. Do you think Beat would have been useful for solving the other task? Yes / No

Q7. Was the software user interface responsive enough? Yes / No

Q8. Were the visualizations produced easy to relate to the source code of the example program? Yes / No

Q9. Did the layout of the visualization and the use of colour and space make it clear how the program was working? Yes / No

Q10. Do you feel that the inclusion of source code in the diagram helped you to relate to how the test program was executing to your source code? Yes / No

Q11. Please comment on any things you liked about the software.

Q12. Please comment on anything you disliked about the software.

Q13. Do you have any recommendations for future changes?

Thank you for your participation.

Appendix B

Evaluation Source Code

B.1 Source Code for Deadlock Task

B.2 Source Code for Race Condition Task

```java
package evaluation;

import beat.collector.BeatTrace;

@BeatTrace
public class Main {
        public static int number_of_runs = 10;

        public static void main(String[] args) {
                new Main().init();
        }

        public void init(){
                Printer firstPrinter = new Printer("Printer 1");
                Printer secondPrinter = new Printer("Printer 2");

                PrinterThread1 pt1 = new PrinterThread1(firstPrinter,
                                                        secondPrinter);
                PrinterThread2 pt2 = new PrinterThread2(firstPrinter,
                                                        secondPrinter);

                pt1.otherThread = pt2;
                pt2.otherThread = pt1;

                Thread printerThread1 = new Thread(pt1,
                                                   "PrinterThread1");
                Thread printerThread2 = new Thread(pt2,
                                                   "PrinterThread2");

                printerThread1.start();
                printerThread2.start();

        }
}
```

Listing 16: Contains main method of deadlock test.

```java
package evaluation;

import beat.collector.BeatTrace;

@BeatTrace
public class Printer {

        private String name;

        public Printer(String name) {
                this.name = name;
        }

        void print(int value){
                System.out.println("Thread: " +
                                   Thread.currentThread().getName() +
                                   " " + name + ": " + value);
        }

        void acquired(){
                System.out.println(name + " acquired by " +
                                   Thread.currentThread().getName());
        }
}
```

Listing 17: Printer object for deadlock test.

```java
package evaluation;
import java.util.Random;
import beat.collector.BeatTrace;

// a printer thread that requires two printer
// objects to be locked simultaneously to print
@BeatTrace
public class PrinterThread1 implements Runnable {
        // printer objects
        Printer firstPrinter;
        Printer secondPrinter;

        // these are to check for deadlocks - see below
        public PrinterThread2 otherThread;
        public boolean holdsOther = false;

        public PrinterThread1(Printer printer1, Printer printer2) {
                this.firstPrinter = printer1;
                this.secondPrinter = printer2;
        }

        Random rand = new Random();

        public void run() {
                for(int i = 0; i < Main.number_of_runs; i++){
                        synchronized(firstPrinter){
                                holdsOther = true;

                                firstPrinter.acquired();

                                try {
                                        if(rand.nextBoolean()){
                                        Thread.currentThread().
                                                sleep(rand.nextInt(5));
                                        }
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }

                                checkDeadlocked();
```

Listing 18: First printer thread for deadlock test part 1.

```
                                synchronized(secondPrinter){
                                        secondPrinter.acquired();
                                        firstPrinter.print(i);
                                        secondPrinter.print(i);
                                }

                                holdsOther = false;
                        }
                }
        }

        /**
         * Check for deadlocks - we need to do this because of
         * some limitations in how synchronized is implemented
         * in the Java Virtual Machine - ask Paul for details.
         *
         * (This will be fixed in future versions)
         */
        private void checkDeadlocked() {
                if(otherThread.holdsOther){
                        System.out.println("Deadlock in thread " +
                            Thread.currentThread().getName());
                        throw new ThreadDeath();
                }
        }
}
```

Listing 19: First printer thread for deadlock test part 2.

```java
package evaluation;
import java.util.Random;
import beat.collector.BeatTrace;

// a printer thread that requires two printer
// objects to be locked simultaneously to print
@BeatTrace
public class PrinterThread2 implements Runnable {
        // printer objects
        Printer firstPrinter;
        Printer secondPrinter;

        // these are to check for deadlocks - see below
        public PrinterThread1 otherThread;
        public boolean holdsOther = false;

        public PrinterThread2(Printer printer1, Printer printer2) {
                this.firstPrinter = printer1;
                this.secondPrinter = printer2;
        }

        Random rand = new Random();

        public void run() {
                for(int i = 0; i < Main.number_of_runs; i++){
                        synchronized(secondPrinter){
                                holdsOther = true;

                                secondPrinter.acquired();

                                try {
                                        if(rand.nextBoolean()){
                                        Thread.currentThread().
                                        sleep(rand.nextInt(5));
                                        }
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
```

Listing 20: Second printer thread for deadlock test part 1.

```
                                checkDeadlocked();

                                synchronized(firstPrinter){
                                        firstPrinter.acquired();
                                        firstPrinter.print(i);
                                        secondPrinter.print(i);
                                }

                                holdsOther = false;
                        }
                }
        }

        /**
         * Check for deadlocks - we need to do this because of
         * some limitations in how synchronized is implemented
         * in the Java Virtual Machine - ask Paul for details.
         *
         * (This will be fixed in future versions)
         */
        private void checkDeadlocked() {
                if(otherThread.holdsOther){
                        System.out.println("Deadlock in thread " +
                            Thread.currentThread().getName());
                        throw new ThreadDeath();
                }
        }
}
```

Listing 21: Second printer thread for deadlock test part 2.

```java
package evaluation;

import beat.collector.BeatTrace;

// @BeatTrace tells Beat to trace this class
@BeatTrace
public class Main {
        public static int number_of_runs = 10;

        public static void main(String[] args) {
                new Main().init();
        }

        public void init(){
                Shared shared = new Shared();

                Thread printerThread1 = new Thread(
                    new PrinterThread(shared), "PrinterThread1");
                Thread printerThread2 = new Thread(
                    new PrinterThread(shared), "PrinterThread2");

                printerThread1.start();
                printerThread2.start();

        }
}
```

Listing 22: Contains main method of race condition test.

```java
package evaluation;

import beat.collector.BeatTrace;

@BeatTrace
public class PrinterThread implements Runnable {
        Shared shared;

        public PrinterThread(Shared shared) {
                this.shared = shared;
        }

        public void run() {
                for(int i = 0; i < Main.number_of_runs; i++){
                        int value = shared.updateValue();

                        System.out.println(Thread.currentThread().
                                  getName() + " value: " + value);
                }
        }
}
```

Listing 23: Printer thread for race condition test.

```java
package evaluation;

import java.util.Random;

import beat.collector.BeatTrace;

// Shared object
@BeatTrace
public class Shared {
        private int value = 0;

        Random random = new Random();

        public int updateValue() {
                int temp = value;

                // pretend to perform complex processing -
                // (really just waste time to cause race conditions)
                try {
                                Thread.currentThread().sleep(5);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                value = temp += 1;

                return value;
        }
}
```

Listing 24: Shared object for race condition test.

Appendix C

Result Tables

C.1 Timing and Experience Results

Participant	**Group**	**Q1**	**Q2**	**Q3**	**Q4**	**Deadlock time**	**Race time**
0	1	Yes	3	3	Same	10	2
7	1	No	N/A	N/A	N/A	N/A	N/A
11	1	Yes	3	4	Race	10	N/A
15	1	No	2	3	Same	8	3
18	1	Yes	2	4	Race	10	2
16	2	Yes	2	3	Deadlock	6	10
1	2	Yes	3	5	Race	10	8
12	2	Yes	4	4	Deadlock	1	7
8	2	Yes	4	3	Deadlock	2	8
4	2	Yes	2	2	Deadlock	4	7
2	3	Yes	4	5	Deadlock	5	6
5	3	Yes	3	4	Race	2	5
9	3	Yes	3	3	Same	4	7
13	3	Yes	4	4	Same	2	1
17	3	Yes	5	5	Race	2	1
3	4	Yes	4	4	Race	11	1
6	4	Yes	3	3	Race	4	1
10	4	Yes	4	5	Race	6	1
14	4	Yes	4	4	Race	1	5

19	4	Yes	4	5	Race	7	1

C.2 UI Questions

Participant	**Group**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**
0	1	Yes	Yes	Yes	No	No	Yes
7	1	N/A	N/A	Yes	Yes	Yes	Yes
11	1	Yes	Yes	Yes	Yes	Yes	Yes
15	1	Yes	Yes	Yes	Yes	Yes	Yes
18	1	No	Yes	Yes	Yes	Yes	Yes
16	2	Yes	Yes	Yes	Yes	Yes	Yes
1	2	Yes	Yes	Yes	Yes	Yes	Yes
12	2	Yes	Yes	Yes	Yes	Yes	Yes
8	2	No	Yes	No	Yes	No	Yes
4	2	No	N/A	Yes	No	Yes	N/A
2	3	Yes	Yes	Yes	Yes	Yes	Yes
5	3	Yes	Yes	Yes	Yes	Yes	Yes
9	3	Yes	Yes	Yes	Yes	No	Yes
13	3	Yes	Yes	Yes	Yes	Yes	Yes
17	3	Yes	Yes	Yes	Yes	Yes	Yes
3	4	Yes	Yes	Yes	Yes	Yes	Yes
6	4	Yes	Yes	Yes	Yes	Yes	Yes
10	4	Yes	Yes	Yes	Yes	Yes	Yes
14	4	Yes	Yes	Yes	Yes	Yes	Yes
19	4	No	Yes	Yes	Yes	Yes	Yes

C.3 Comments

Participant	**Question**	**Comment**
0	Q11	It showed a trace of the threads & what is happening to them
0	Q12	It needs to much screen space & requires me to scroll horizontally which I don't like
0	Q13	Don't output code directly but instead display clickable icons which expand to show code.

1	Q11	It was great and easy to use. I did really like it as it gives me a better visualization of how threads working.
1	Q12	
1	Q13	If we can get the snapshot of screen before and after fixe a bug to see how it effects on visualization (for comparison)
2	Q5	If I knew hot to use beat, needs a user manual
2	Q8	Should have read source code first, ordering of columns is important.
2	Q11	Good colours and layout
2	Q12	needs a comprehensive user manual and training to use
2	Q13	order the columns in a more intuitive manner
2	Q10	I always read the source code, it is how one normally debugs.
3	Q11	Having a visual representation of what the threads are actually doing at different points in time was good because in your head theoretically it can be hard to visualize and System.out.println has its limitations.
3	Q12	Nothing comes to mind at this time
3	Q13	Not at this time sorry.
4	Q11	Knowing how the thread worked
4	Q12	Found Beat a little confusing, as there were parts of code that didn't seem to link together
4	Q13	
5	Q11	It works, thats a bonus
5	Q12	overlapping lines can obscure labels sometimes
5	Q13	Not really.
6	Q11	moving columns. Colour.
6	Q12	can't always see all code if lines too long.
6	Q13	

7	Q11	It's clarity (good contrast, soft colouring). The ability to reasize and reorder the threads. The rubber banding while modifying. The timeline approach to visualize the thread that are currently executing and the transfer of control
7	Q12	
7	Q13	A method of disabling a selected thread. A method of highlighting changes to a selected variable or variables.
7	Q10	Sufficient context to see whats happening.
8	Q11	Very good at showing the actual run order which can sometimes can be difficult to comprehend
8	Q12	Hard to resize column was expecting resize cursor, not the arrow cursor for resize. Hard to pickup cold, took awhile to understand/process visually.
8	Q13	fix cursor. Need more intro\examples of using the diagram.
9	Q11	A great tool for demonstrating concurrency. Representation is very clear.
9	Q12	Some codes are cut (column width for Threads' codes are not wide enough)
9	Q13	Legend for the symbols used in Beat.
10	Q11	Visual representation of threads working in action
10	Q12	No adjustment for columns on the top of the chart
10	Q13	Debbuger Mode
11	Q11	Moveable + resizeable columns
11	Q12	Repeating method names
11	Q13	
12	Q11	Reordering cols for a better view. Showing source execution.
12	Q12	Thread key hides some col info.
12	Q13	
13	Q11	Logical program flow, from top to bottom
13	Q12	zzz and x icons are ugly some nice icons perhaps?
13	Q13	variable highlighting.

13	Q10	Maybe colapsable for larger segments
14	Q11	Logical layout of objects, time, code and process life
14	Q12	Nothing
14	Q13	No
15	Q11	Lanes and colour coding made easy to follow objects and threads.
15	Q12	
15	Q13	Stepping through running code also steps through visualization.
16	Q11	Really liked inclusion of code in diagram for Beat
16	Q12	
16	Q13	Use of Lock, Semaphores etc.
17	Q11	Visualization: Moving of Object instances, being able to organize the flow. Beat: Transparent integration through pre-compilation.
17	Q12	Visualization: too much information present. Beat: lack of debugger integration.
17	Q13	filters for content, being able to restrict the visualization. Debugger integration, step through a program.
18	Q11	It looks very useful, would be good to find concurrancy bugsin software I already understand.
18	Q12	having the line of code twice is a little confusing.
18	Q13	Mabey using braces instead of the line twice, or something other than the line repeated.
19	Q11	Displaying the flow of threads, and when threads are sleeping. Displaying synchronized blocks with rounded rectangles. Different colours for different threads.
19	Q12	Difficult to switch back to Java view.

19	Q13	Link in Java directly e.g. ctrl-click to open method definitions. Add an overview display. Replace tabs with shorter whitespace. Allow content resie (font size). Perhaps different colours for different synchronized objects (not just threads). If hover mouse over a code block, show the entire code block as an overlay.

Bibliography

[1] Eclipse java development tools (jdt) overview. http://www.eclipse.org/jdt/overview.php, 2010.

[2] Eclipse.org home. http://eclipse.org/, 2010.

[3] Html5 (including next generation additions still in development). http://www.whatwg.org/specs/web-apps/current-work/, 2010.

[4] java.com: Java + you. http://www.java.com/en/, 2010.

[5] Abhi on java: Java 5 concurrency: Callable and future. http://java-x.blogspot.com/2006/11/java-5-concurrency-callable-and-future.html, 2011.

[6] Air protocol at ipulse blog. http://shred444.com/blog/?p=31, 2011.

[7] Antlr parser generator. http://www.antlr.org/, 2011.

[8] Apple - logic studio. http://www.apple.com/logicstudio/, 2011.

[9] aram's blog blog archive midterm idea - treemap. http://www.aramchang.com/blog/2008/02/a_to_z/midterm-idea-treemap/, 2011.

[10] The aspectj project. http://www.eclipse.org/aspectj/, 2011.

[11] Debugging with intellitrace. http://msdn.microsoft.com/en-us/library/dd264915.aspx, 2011.

[12] Design codes: Uml sequence diagram: Interaction fragment (alt, opt, par, loop, region). http://aviadezra.blogspot.com/2009/06/uml-fragment-alt-opt-par-loop-region.html, 2011.

[13] Erlang programming language, official website. http://www.erlang.org/, 2011.

[14] Intellitrace - debugging applications with intellitrace. `http://msdn.microsoft.com/en-us/magazine/ee336126.aspx`, 2011.

[15] Java se - java platform debugger architecture - faqs. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/faqs.jsp`, 2011.

[16] Java se - java platform debugger architecture home. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`, 2011.

[17] Jdk 6 java virtual machine tool interface (jvmti). `http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/`, 2011.

[18] jquery: The write less, do more, javascript library. `http://jquery.com/`, 2011.

[19] Nehalem - everything you need to know about intel's new architecture. `http://www.anandtech.com/show/2594/6`, 2011.

[20] Ni labview - improving the productivity of engineers and scientists. `http://www.ni.com/labview/`, 2011.

[21] Ruby programming language. `http://www.ruby-lang.org/en/`, 2011.

[22] Tracing user processes - dtrace user guide. `http://dlc.sun.com/osol/docs/content/DTRCUG/gcgkk.html`, 2011.

[23] Uml 2 communication diagramming guidelines. `http://www.agilemodeling.com/style/collaborationDiagram.htm`, 2011.

[24] visualvm: Home. `https://visualvm.dev.java.net/`, 2011.

[25] Welcome to netbeans. `http://netbeans.org/`, 2011.

[26] Jim Arlow and Ila Neustadt. *Uml and the Unified Process: Practical Object-Oriented Analysis and Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[27] Cyrille Artho, Klaus Havelund, and Shinichi Honiden. Visualization of concurrent program executions. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[29] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[30] S. K. Chang, Margaret M. Burnett, Stefano Levialdi, Kim Marriott, Joseph J. Pfeiffer, and Steven L. Tanimoto. The future of visual languages. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 58–, Washington, DC, USA, 1999. IEEE Computer Society.

[31] Chaomei Chen. Information visualization. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2, 2010.

[32] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[33] Wim De Pauw and Steve Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 143–152, New York, NY, USA, 2010. ACM.

[34] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[35] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23, May 2000.

[36] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 95–104, New York, NY, USA, 2005. ACM.

[37] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[38] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[39] Joel Huselius. Debugging parallel systems: A state of the art report. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, Mälardalen University, September 2002.

[40] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004.

[41] Edward A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[42] Gowritharan Maheswara, Jeremy S. Bradbury, and Christopher Collins. Tie: an interactive visualization of thread interleavings. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 215–216, New York, NY, USA, 2010. ACM.

[43] Mark McGrain. *Music Notation (Berklee Guide)*. Berklee Press, 1990.

[44] Stephan Merz. Model checking: A tutorial overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 3–38, London, UK, 2001. Springer-Verlag.

[45] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. 1 chess: A systematic testing tool for concurrent software.

[46] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[47] Richard Rastall. *The Notation of Western Music: an Introduction*. J. M. Dent & Sons London, 1983. Interesting account of the evolution and origin of common notation starting from neumes, and ending with modern innovations HWN.

[48] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[50] N Shavit and D Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[51] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[52] Ben Shneiderman and Benjamin B. Bederson. *The Craft of Information Visualization: Readings and Reflections*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[53] N. C. Shu. Visual programming languages: A perspective and a dimensional analysis. *Visual Languages*, 1986.

[54] John T. Stasko. The parade environment for visualizing parallel program executions: A progress report. Technical report, 1995.

[55] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[56] CORPORATE SunSoft. *Solaris multithreaded programming guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[57] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[58] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[59] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 1998.

[60] James J. Thomas and Kristin A. Cook. A visual analytics agenda. *IEEE Comput. Graph. Appl.*, 26:10–13, January 2006.

[61] Melanie Tory and Torsten Moller. Rethinking visualization: A high-level taxonomy. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 151–158, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 133–142, New York, NY, USA, 2010. ACM.

[63] Edward Tufte. *Envisioning Information.* Graphics Press, Cheshire, CT, 1990.

[64] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[65] Shaohua Xie, Eileen Kraemer, R. E. K. Stirewalt, Laura K. Dillon, and Scott D. Fleming. Assessing the benefits of synchronization-adorned sequence diagrams: two controlled experiments. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 9–18, New York, NY, USA, 2008. ACM.

[66] Qiang A. Zhao and John T. Stasko. Visualizing the execution of threads-based parallel programs. Technical report, 1995.