ABSTRACT
The introduction of concurrency within emerging languages such as Java brings challenging new concepts to the user. Owing to the inherent non-determinism of threads and multiple flow of control in concurrent programs, traditional debugging and comprehension techniques, such as source code analysis, do not suffice. We believe that visualisation can assist in expediting comprehension of concurrent programs. We present, Jacot, a visualisation tool to depict the execution of concurrent Java programs. Jacot has two views based on the Unified Modeling Language (UML) Sequence diagram paradigm. It is implemented in Java and uses the Java Debug Interface (JDI) for event gathering. It depicts the interaction between objects and the interleaving of threads in a timely fashion, using method invocations.

Categories and Subject Descriptors
D.2.6 [Programming Environments]: Graphical environments; D.1.3 [Concurrent Programming]: Parallel Programming; D.2.2 [Design Tools and Techniques]: State Diagram; D.2.5 [Testing and Debugging]: Monitors, Testing tools, Tracing.

General Terms
Design, Human Factors, Languages.

Keywords
Java, Concurrency, Visualisation, UML.

1. INTRODUCTION
Concurrency is emerging among the IT community. This is partly due to programming languages such as Java having concurrency inbuilt in the language. Furthermore, the need for more processing power and development of distributed systems, make concurrency more attractive. However, the introduction of concurrency into the university curriculum and the IT community brings challenging new concepts to the user. The interleaving and non-deterministic traits of threads during execution are some of these issues. Threads also compete among each other for shared resources, such as memory or processor cycles. Control of flow during execution is constantly being shifted from one thread to another. These issues add significant burden on the shoulders of a novice programmer.

Visualisation uses a blend of colours, space, text and shapes to provide the user with a model of the dynamics of the software. We believe that visualisation can be of tremendous assistance in expediting comprehension of concurrent Java programs. We develop Jacot (an acronym for JAva Concurrent Object Tool), a prototype of a visualisation tool that takes a concurrent Java program, as input, and dynamically visualises its execution using views based on the Unified Modeling Language [7] (UML) paradigm.

This paper focuses on the use of the Java Standard Development Kit (SDK) version 1.4.1 to implement Jacot. More specifically, we use the Java Platform Debugger Architecture (JPDA) and its underlying interfaces to collect and process events generated by an executing concurrent program. We look at the advantages of using Java as an implementation platform in terms of language support for concurrency, visualisation and event gathering. We also provide some comments on some shortcomings and how we addressed them.

This paper is organised as follows. In the next section, we present the views comprising Jacot. This is followed by an outline of the architecture of Jacot. We proceed with a brief discussion of the implementation decisions and present related work in the field. We then summarise and conclude our paper.

2. THE JACOT INTERFACE
Jacot is a visualisation tool comprising of two views: a Sequence Diagram and a Thread State Diagram. We present the interfaces and design decisions relating to those two views below.

2.1 Sequence Diagram
The Sequence diagram is based on the UML Sequence Diagram paradigm. A Sequence diagram is well suited for visualising software execution as it has an explicit time axis. It is also powerful in demonstrating the interaction between objects by illustrating method invocations. Furthermore, the combination of the time dimension and message-passing capabilities allow for the interleaving of events to be clearly depicted.
2.1.1 Objects
Depiction of objects in Jacot is exactly as prescribed by the UML specifications [7]. Objects are represented by rectangles, with the object’s name and its unique identifier centred in the rectangle. A vertical dashed-line (the lifeline) denotes that the object is alive but not currently active. When an object becomes active, the lifeline is substituted for a vertical lane (the Activation diagram). As the execution of the program proceeds, the object’s lifeline or activation diagram is incremented from top to bottom. When an object ceases to exist, its lifeline or activation diagram is halted and a red “X” is drawn below it. Figure 1 illustrates object creation, activation and deletion. It also depicts method invocations as well as thread interleaving and termination.

2.1.2 Method Invocation
We also make no changes to the UML specifications for method invocations. Method invocations are depicted as directed arrows from the calling objects to the called objects. A self-call is represented as a curved directed arrow. A synchronised method call is shown as a filled solid arrowhead, while an unsynchronised method call has a stick arrowhead. A dashed arrow with a stick arrowhead is used to illustrate a return from method call. Figure 2 illustrates the different method invocations. The first processString method is an example of an unsynchronised method call. The delay method shows a self-call, while the updated methods depict unsynchronised and return from method calls respectively.

2.1.3 Thread Depiction
Each thread executing within the program is shown in a different colour. Thread activation is commonly associated with message passing paradigms. Incidentally, thread creation and termination are shown in the Sequence view as a method invocation with a directed arrow in the colour of the thread and a wavy symbol, to indicate thread creation and with an “Ω” to indicate thread termination. However, the current colour association of threads does not allow the tool to scale well. The next version will address this issue by associating a colour to each object. Threads created by this object will then share tones of this colour at increment of 25. This addresses several design limitations of software visualisation. Firstly, it minimises the difficulty associated with assigning a different colour to each thread. Secondly, it enables the clustering of threads from a particular object to be made possible.

2.2 Thread State Diagram
We propose a Thread State Diagram, based on the UML Statechart diagram. One of our objectives is to fully utilise UML diagrams and concepts. During its lifetime, a thread can be in any of seven states, namely created, waiting, runnable, running, waiting on a monitor, sleeping and terminated. We choose, however, not to represent the created and running states because they are, essentially, conceptual states of threads in concurrency and in reality, threads only transit briefly in those two states. We instead depict an “unknown” state, primarily because the JVM supports such a state but more importantly because it depicts a thread that has been suspended by the debugger. The Thread State Diagram consists of six interconnected states, each depicting one of the above states including the “unknown” state, together with an initial pseudostate and a final transition, as prescribed by UML. The threads are then depicted inside the states as filled circles in the thread colour. We also include a number inside each
circle to indicate the thread’s priority. Furthermore, in the next version we will provide vital statistics about the thread by right clicking on the circle. These threads migrate from one state diagram to the other as a result of a change in their state, as shown in Figure 3.

Figure 3: Thread State Diagram

3. ARCHITECTURE
Jacot is based on a four-layer model built on top of the three-layer JPDA architecture as shown in Figure 4. It is modular allowing it to be extensible and accommodate for more views to be added in the future. This tool is the successor to the COOPE [3] environment as our objectives have evolved.

Figure 4: Jacot Architecture

Java was chosen as the implementation platform for many reasons. Firstly Java offers platform independence. This allows us to evaluate Jacot under different operating systems.

Secondly, Java uses a virtual machine. More specifically the JPDA architecture provides mechanism to significantly assist in the gathering of events from the program under study. The three layers comprising the JPDA are: (i) the Java Virtual Machine Debug Interface (JVMDI), a native interface that defines the debugging services that each VM must provide; (ii) the Java Debug Wire Protocol (JDWP), which defines the communication format between the program being debugged and the debugger and (iii) the Java Debug Interface, a high-level pure Java interface that defines information and requests at the user level in pure Java code.

Jacot uses the JDI to obtain traces from the program under study. The main advantage of using the JDI is that these traces can be obtained in a non-invasive manner, thus eliminating the need to modify the source code of the “debugged” software. The two benefits of this approach are that (i) it reduces the risk of inclusion of errors in the code; and (ii) it allows for a much wider range of software to be visualised, even if the source code is unavailable.

Thirdly, Java offers a rich graphical interface inbuilt in the language, thus allowing powerful graphical user interface to be implemented without the need to import external libraries. We make use of the Swing and Java2D graphics methods in our implementation.

Finally, our aim is to visualise concurrent Java programs and it seems rather logical to implement the visualisation tool using the same platform. Furthermore, this gives us the potential to visualise the inner-workings of Jacot in the future.

3.1 Event Gathering Layer
The Event Gathering layer is responsible for requesting and collecting events from the “debugger” VM. The events are gathered in a “polling” fashion. Once started, the tool waits and listens for events from the “debuggee” VM to be placed on the Event Queue. Once an event is generated, the event set is removed from the queue. The program then iterates through the set of events wrapping each event in a message that is then forwarded to the next layer for processing. These steps are repeated indefinitely until either an event notifying that the debuggee VM has been terminated or the program exits.

3.2 Event Processing Layer
The Event Processing layer gathers detailed information about each event and encapsulates it into an object that is then sent to higher layers to be displayed. After careful considerations regarding the level of abstraction that need to be provided by the visualisation interface and taking into account the four overlapping guidelines for visualisation as discussed in [2] we arrive at the following set of desirable functionalities for Jacot. These are: (i) object creation and destruction; (ii) method invocations; (iii) thread creation, termination and status; (iv) dynamic interleaving of threads; (v) object’s monitor; (vi) exception handling and (vii) depiction of critical sections.

The main function of this layer is to determine the type of the event received and querying the JDI for information about the event. For our purposes, this can be any of a Method Entry, Method Exit, VM Start, VM Death, Caught Exception, Thrown Exception or Class Prepare event. The type of the event further determines what additional information can be gathered as well as their types. For example from a Method Entry event object, we can obtain information relating to the name of the method making the call; the ID of the thread executing the method; the name and group name of the thread; the ID and name of the “calling” and “called” objects and more specific information about the method itself, such as whether it is synchronised, static, final or abstract.

We identify the objects associated with an event by pairing the object’s name and object’s ID together. The object ID alone is not sufficient in the case of a static method call since the ID returned by the JDI, in this case, is undefined as this in fact refers to a call to or from a class. We cater for this by combining the object ID with the object name.
Furthermore in the case of method invocations, only one object is readily available in the event trace obtained from the JPDA. In the case of method entry, the object that is obtained from the JVM is the “called” object. We obtain details of the “calling” object by traversing the stack of method calls made by this thread. The call prior to the current call provides the identity of the “calling” object.

3.3 View Formatting Layer
The “View Formatting” layer is mainly concerned with arranging the information passed on from the previous layer in a coherent manner on the screen. Decisions such as placement location, shape, size, colour, shade are dealt with here. The two views within Jacot handle the formatting of their respective views independently and differently.

3.4 Display Layer
The Display layer comprises of a dozen classes each responsible for drawing a graphical component on the screen.

4. IMPLEMENTATION DECISIONS
The Jacot visualisation tool is implemented as a stand-alone application running on the same platform as the program under study. This poses the problem of contention of memory and processor cycles between the program and Jacot. However, competing for the shared resources on the computer may bring forth hidden errors in the program that may never have been uncovered if the program was not contending for resources. We also decided to build the application from scratch rather than rely on an existing open tool that supports UML and allows the user to change the underlying UML metamodel [7]. Our decision was motivated by the fact that Jacot is a prototype and our main goal is to validate our concepts.

Jacot executes by starting a new JVM to run the target program. From this point forward, Jacot controls the execution of the program. The main window provides the user with the ability to play, pause, slow down and stop the execution of the program using the “play/pause” and “stop” buttons and the slider on the menu bar. The executing program retains all of its functionalities though, opening the same windows, performing the same tasks and displaying the same information as it does when executing on its own.

4.1 Visualising concurrency
Jacot currently focuses on Liveness issues and provides mechanisms to visualize Starvation (Contention), Dormancy and Deadlocks. We describe how these are visualized in Jacot below.

4.1.1 Contention (Starvation)
Visualising contention involves monitoring the activity of threads within the Sequence Diagram. The user may conclude that a thread is starved of CPU time by observing long periods of inactivity by the contended thread in the Sequence Diagram. However, Contention is better illustrated in the Thread State Diagram. In the case of contention due to priority, glancing at the pool of waiting and monitor threads should indicate whether the contended thread has a lower priority than other running threads. In the case of a fairness problem causing starvation, the user can observe the time spent for that the thread in the above two states.

In the next version, the user will be able to query the thread and obtain an estimate of the time spent therein.

We choose a simplified version of the SingleLaneBridge example [4] to illustrate contention. An example of the execution of an unsafe and unfair version of the SingleLaneBridge program within Jacot can be seen in Figures 1 and 3. From the diagram, we can see that all the six cars have entered the Bridge. And RedCar 538 has already exited the bridge even though RedCar 541 was the first one to enter.

The Thread State Diagram gives a more accurate depiction of Contention by showing four of the six car threads waiting on a monitor.

4.1.2 Dormancy
Dormancy is illustrated by monitoring the invocation of synchronisation primitives in the Sequence Diagram together with the activity of the threads executing. This allows the developer to identify the simple case of a wait command not followed by a notification. However, it is also useful in elucidating the more complex synchronisation problem whereby a thread gets notified before the wait process has even started. The thread state diagram may not provide an explanation as to why the thread is dormant. However, it assists by showing the waiting threads and providing information regarding the time spent in that state.

4.1.3 Deadlock
Deadlock is more accurately depicted in the executing program by observing the interleaving of thread invocation, for which the Sequence Diagram is particularly powerful. The occurrence of deadlock can be portrayed in the Sequence Diagram by following the offending threads along the “crime trail” while observing the locking sequence of objects by threads at differing times during the execution. The order of the locks sequences, the frequency of the locks, the duration of the locks and the activities performed while the locks are in place provide crucial information in analysing the deadlock. The thread state diagram can shed light on the order of transition of the thread state. It can also provide clues about the time spent in different states. Combined, these two diagrams can assist in portraying the four conditions for the occurrence of deadlocks as discussed in [1].

We choose the Dining Philosophers problem, one of the classic examples in the literature, to illustrate deadlock in a multithreaded program. The problem is as follows. Five Philosophers are seated around a table with a bowl of spaghetti in the middle and only one chopstick between each pair of Philosophers. However, because the spaghetti is long and tangled, each Philosopher needs two chopsticks to eat.

Figure 5 and 6 show the execution of the Dining Philosophers problem using Jacot. From the view, we notice that a deadlock has occurred and execution of the program has been halted. Each philosopher has successfully acquired one chopstick and made an attempt at grabbing a second one.

The problem arises when the philosophers attempt to grab another chopstick. All the objects are deadlocked because each Philosopher has grabbed one chopstick but cannot obtain another one, and yet, no one releases his chopstick.
Figure 5: Sequence Diagram depicting the deadlock of the Dining Philosopher program

Figure 6: Thread State Diagram depicting the deadlock of the Dining Philosopher program

This is more accurately depicted in the Thread Status view where all but one threads are shown in the Waiting pane.

5. RELATED WORK
JaVis [5] is a tool for visualising and debugging concurrent Java programs. The visualisation is performed using Sequence and Collaboration diagrams, based on the UML [7] paradigm. JaVis uses the UML Case Tool Together as its implementation platform. The main objective and strength of the tool is in the discovery and notification of deadlocks. JaVis supports abstraction, depicting only the objects and method calls directly involved in the deadlock. JaVis shares many of its functionalities with Jacot. However, at the time of writing, unlike Jacot, JaVis only focuses only on deadlock issues.

The Parade [9] environment uses the POLKA [10] animation system as an implementation platform to visualise concurrent programs. The visualisation is made up of four views: the Threads view, the Function view, the History view and the Mutex view. Two of its views have similar functionalities to Jacot. The Threads view is similar to our Thread State Diagram in that it shows the threads in a unique colour. However, it only depicts the status of the threads as runnable (coloured), blocked (half coloured) and terminated (black). The History view provides a listing of the method invocations since the start of the program. It is very similar to our Sequence view but not as detailed. However, Parade does not target specifically Java programs.

Jinsight [8] can be regarded as a visual debugger for visualising the run-time performance of Java programs. It does not aim directly at addressing concurrency issues. Instead it is mainly powerful at uncovering memory leaks.

Javavis [6] is an educational visualisation tool aim at supporting the teaching of object-oriented concepts in Java to novices. The tool is quite similar to JaVis and Jacot with the exception that it does not yet support concurrency concepts. Javavis also uses UML Sequence diagrams as medium for the visualisation. Despite not supporting concurrency, it shows good promise by displaying an object diagram for each active method on the call stack of threads, with interesting navigational features.
6. CONCLUSION
This paper has presented a visualisation tool, called Jacot, to represent the execution of concurrent Java programs. Its main objective is to assist the user in understanding challenging new concurrency concepts, such as synchronisation, non-determinism and deadlock. Jacot is implemented in Java and uses the JDI to collect traces in a non-invasive manner from the JVM of the executing program. It comprises two views: a Sequence view, which is based on the UML Sequence diagram paradigm, and a Thread view.

7. REFERENCES