Publication IV

Jan Lönnberg, Mordechai Ben-Ari and Lauri Malmi. Java Replay for Dependence-based Debugging. In *Proceedings of PADTAD IX — Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pp 15–25, Toronto, Ontario, Canada, July 2011.

Java Replay for Dependence-based Debugging

Jan Lönnberg
Aalto University
School of Science
P.O. Box 15400
FI-00076 Aalto, Finland
jlonnber@cs.hut.fi

Mordechai Ben-Ari
Weizmann Institute of Science
Department of Science
Teaching
76100 Rehovot, Israel
moti.ben-ari@weizmann.ac.il

Lauri Malmi
Aalto University
School of Science
P.O. Box 15400
FI-00076 Aalto, Finland
lma@cs.hut.fi

ABSTRACT

In this article, we present a system intended to help students understand and debug concurrent Java programs. The system instruments Java classes to produce execution traces. These traces can then be used to construct a dynamic dependence graph showing the interactions between the different operations performed in the program. These interactions are used as the basis for an interactive visualisation that can be used to explore the execution of a program and trace incorrect program behaviour back from a symptom to the execution of incorrect code.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent Programming; D.2.5 [**Software engineering**]: Testing and debugging—*Debugging aids*; K.3.2 [**Computers and education**]: Computer and Information Science Education—*Computer science education*

General Terms

Design, Performance

Keywords

Dynamic dependence analysis, execution replay, program visualisation, Atropos

1. INTRODUCTION

Finding and correcting defects in concurrent programs is well known to be difficult. Approaches that involve re-executing a program are problematic due to nondeterminism, and reasoning about what happens in a concurrent program also becomes much harder. This suggests a need for methods and software that would make it easier to find (and then eliminate) these defects. Our long-term goal is to improve the correctness of concurrent software by developing better tools and teaching methods.

PADTAD '11, July 17, 2011, Toronto, ON, Canada

Most of the previous work on debugging has concentrated on studying the current state of the program and examining the execution of individual processes. Concurrent programs, however, are likely to have defects that involve unexpected interactions between concurrent processes; these interactions make it hard to debug using these methods. A promising solution to this—which complements existing verification and debugging techniques—involves visualisations designed to aid the programmer in understanding the interactions between the operations performed in a program.

Students are, for several reasons, both an obvious target audience and group of subjects for research. One is that they do not yet have ingrained ways of working, so it is easier to introduce new methods and tools to them. Second, they have more difficulties due to incomplete knowledge and lack of experience. Third, they have most of their career ahead of them, so they will presumably have more chances to benefit from new methods and tools. Finally, university researchers have easy access to students and are familiar with the courses taught and their subject matter.

1.1 Background

The work described in this paper is part of a larger project to help students become more effective at working with concurrent programs. This involves both identifying difficulties students have with understanding concurrency [18] and producing correct concurrent programs [17, 20], as well as finding ways to assist them [21]. This project is centred around the Concurrent Programming course[1] at Aalto University. The goal of this course is to teach students the principles of concurrent programming:

- Synchronisation and communication mechanisms
- Concurrent and distributed algorithms
- Concurrent and distributed systems

To enable students to apply this knowledge in practice and assess their skill, the course includes a set of programming assignments and exercises in Java. Students can either work alone or work on one solution as a pair.

1.2 Research Questions

Our long-term research questions are:

1. Does visualisation of incorrect program executions help students find the underlying defects in their concurrent programs?

[1]Course web site at: `https://noppa.tkk.fi/noppa/kurssi/t-106.5600/etusivu`

2. Does visualisation help them learn from their errors?

Effectively answering these questions requires studying the effects of students using such visualisation. Clearly, the quality of the visualisation will significantly affect the results, so we first sought to answer two other questions:

3. What is a "good" visualisation of a incorrect executions of concurrent programs, where by "good" we mean a visualisation that can enable students to find and to understand the underlying defects in their programs?

4. How can such a visualisation be created algorithmically from an execution of a program?

As discussed below, we decided to approach research question 3 using a visualisation based on exploring a dynamic dependence graph. Hence, research question 4, the focus of this paper, became: "How do we produce a dynamic dependence graph representation of the execution of a Java program?" The resulting visualisation tool, called *Atropos*[2], is presented in this paper. Research questions 1, 2 and 3 will be addressed in more detail in other papers.

Section 2 presents relevant related work, Section 3 describes our answer to RQ 4 and Section 4 evaluates whether the design in Section 3 is a good answer to RQ 4.

2. RELATED WORK

Many existing fields of research are relevant to this research. The most relevant aspects are summarised in this section. As our setting is heavily Java-dominated, the following discussion focuses on methods relevant to examining Java programs.

The problem of generating effective visualisations of failing executions of a program can be divided into three subtasks: finding a failing execution, collecting the information needed to visualise it and visualising an execution. These subtasks can mostly be solved independently, and doing so makes it easier to combine solutions in different ways to address the requirements of different situations.

The third subtask involves issues both of finding a good visualisation for our purposes (RQ 3) and generating the visualisation (RQ 4). The other two subtasks are necessary to create the visualisation; solving them is also part of answering RQ 4.

2.1 Finding Failing Executions

There are two main approaches to finding ways a program can fail: *dynamic* (in which programs are executed and the results analysed) and *static* (in which the analysis is done on program code without executing it). Here, we focus on dynamic methods, as most work with a focus on concurrency belongs to this category and the focus of static methods is typically on finding patterns in the code that correspond to common mistakes [27].

Model checkers find execution sequences that violate specified properties in software by systematically generating all possible executions of a program, especially all the different ways concurrent operations can interleave. Unfortunately, since the entire execution environment must be contained within the model that is checked, verification must typically be applied to a simplified model of the program. *Java PathFinder* [30] attempts to avoid this problem by using Java as its modelling language. However, not all of the Java standard library nor all behaviour of the Java virtual machine is implemented. Model checkers are also hard to apply when the state space of a program is large.

An alternative approach to finding concurrency bugs is to increase the chance of interleavings that lead to failure. One straightforward and realistic way to do this is to distribute the program's threads over multiple processors. Stress testing can also be used. Since no changes to the execution environment are required, stress testing can be applied to a wide range of programs. Another way to further improve the effectiveness of testing is to introduce random and frequent thread switches, which in the case of Java programs can be done by adding calls to switch threads to the Java bytecode [29].

The execution of a program can also be analysed at runtime to find potential race conditions and deadlocks and specification violations. For example, *Java PathExplorer* [12] (JPAX) instruments Java bytecode to output information on execution that can be used to detect data races (in which two writes or a read and a write to the same variable are made simultaneously), deadlocks and violations of requirements specified in temporal logic. *DBRover* [7] can similarly check requirements expressed in Metric Temporal Logic.

Brat et al. [2] present a detailed experimental comparison of different approaches to verification and validation: testing, runtime analysis (using JPAX and DBRover), model checking (using JPF) and static analysis. Four pairs of developers (one for each approach) each attempted to find as many defects as possible in the control code for a Mars rover (seeded with additional defects found in earlier versions of the code) within the allotted time using one of the aforementioned approaches. Runtime analysis and model checking produced similar results in terms of the number of bugs that were found, but runtime analysis resulted in fewer spurious bugs. Model checking also required more initial work to generate suitable abstractions of the code.

2.2 Collecting Executions

Traditional techniques for debugging rely heavily on repeated execution, stepping and breakpoints. This is problematic when a program does not behave deterministically. Hence, collecting an entire program execution sequence for later examination can facilitate debugging a concurrent program. The typical approach to using this information is to replay the execution of the program for examination in another debugging tool. Two important facets of the quality of replay are *accuracy* (how closely the replay matches the original) and *precision* (how much the execution is changed by collecting information on it) [4].

One approach to collecting and replaying execution information is to replace or change the underlying execution engine. Since model checkers maintain an explicit model of the executed operations, it is easy to collect information on a program execution for further study; typically, model checkers can output a list of the operations that were executed or the state transitions. For Java programs (within the limits of what JPF can check), JPF provides a way to get detailed descriptions of their executions.

Instead of completely replacing the JVM, modifying it to collect the information needed to reconstruct an execution can suffice. For example, *DejaVu* changes the Jalapeño

[2] `http://www.cse.hut.fi/en/research/LeTech/Atropos/`

JVM (which runs on a single processor) to keep track of nondeterministic events (e.g. wall clock time) and thread switches. DejaVu also allows the execution of a program thus collected to be replayed for examination in, for example, a traditional debugger. [4]

jreplay [28] similarly uses a modified JVM to collect information on when switches between threads are made, for replay using a program instrumented to enforce a specifed execution order on any JVM.

An alternative approach is to execute the program using a normal Java Virtual Machine (JVM) and collect information on the program's execution from there through *instrumentation*: the addition of code to collect information, as done by e.g. *JaRec* [10]. Like jreplay, JaRec can then enforce the behaviour required for replay through an instrumented version of the program.

RetroVue [3] and *ODB (Omniscient Debugger)* [15] use instrumentation of Java bytecode to collect information about all operations performed by an executing Java program. Instead of replaying the execution, these tools enable the programmer to examine the execution history through a graphical debugger with the capability to show execution histories as a list of operations and to step through different states of the execution that are shown in a manner similar to that of traditional graphical debuggers. A similar approach is used by *MVT* [19] to trace the execution of Java programs, although it does not provide a list view of the execution like RetroVue and ODB.

2.3 Visualising Concurrent Execution

Identifying the aspects of debugging that require the most effort from programmers is important for the development of useful debugging tools. Both von Mayrhauser and Vans [31] and Eisenstadt [8] have shown that programmers spend a lot of time tracing the data and control flow of programs in order to find causes for bugs. They also show that programmers often require information on the causes of an event and connections between parts of a program or its execution when looking for hard-to-find defects. Eisenstadt in particular emphasises that complex cause-effect relations that can be computed (e.g. data flow links) should be computed by the debugger rather than forcing the programmer to work them out. He also points out that the information needed is often at a higher level of abstraction and granularity than the values of individual variables.

A few debuggers and program visualisation systems have been designed with concurrency in mind. Most of them (e.g. *JAVAVIS* [25] and *JAN* [16]) use sequence diagrams or message sequence charts to display method calls; *JaVis* [23] adds collaboration diagrams to show interactions between objects. These diagrams have a level of detail suitable for debugging, but become cumbersome for complex executions.

Finding a defect from a model checker counterexample is similar to debugging in that the program is known to behave incorrectly and the programmer seeks to find the underlying defect. *Bogor* [26], for example, enables the user to examine the counterexample using visualisations similar to those of DDD [34] and RetroVue). *Spin* [13] can produce message sequence charts that show interactions between processes.

2.3.1 Slicing and Dependence Graphs

When looking for reasons for the incorrect behaviour of a program, many programmers look at the (incorrect) value of a variable and try to find a reason for this value by tracing the execution of the program backwards. In essence, they try to concentrate on the part of the program that could have affected the value of the variable. Parts of programs that could have affected the value of a specified variable at a specified point in the execution of a program are called *slices* and the process of calculating slices is called *slicing* [32, 33].

In order to understand slicing, let us first consider slicing of a sequential program. The original form of slicing, *static slicing*, is done through static analysis of control and data flow in a program. Each statement (such as assignment, read, write or conditional branch statements[3]) *uses* and *defines* a set of variables (including all containers for data values that are passed between statements) and has a set of successor statements, forming a flow graph.

A statement has a *control dependency* on another statement if whether it executes or not depends on the other. More formally, i has a control dependency on j if j has successors k and l and all paths to the end of the program from k go through i while at least one path from l to the end of the program does not contain i.

A statement i has a *data dependency* on statement $j \neq i$ if j defines a variable that i uses and a path in the flow graph from j to i exists in which the variable is not defined by any other statement on the path than j and, possibly, i.

Control and data dependencies define the edges of a *program dependence graph*. Given a statement i and a variable v in the program, a *static slice* with respect to i and v is the set of statements reachable from the statements that define a value of v that can be used in i through the program dependence graph with its edges reversed. It contains all the statements that can affect the value of the variable at the specified point; essentially, it is a (hopefully) smaller program that always gives the same value of v at i. [1, 32]

Slices can also be calculated using dynamic analysis of a specific execution history of a program, in which case the slice (a *dynamic slice*) contains all the operations (executions of statements) in the execution history that could have affected the value of the variable at the end of the execution history. In other words, the static slice is based on any possible execution history, while the dynamic slice is based on a specific execution history. [1, 32]

A dynamic slice is based on a *dynamic dependence graph* (*DDG*). A DDG is a directed graph whose vertices are the statements that have been executed (one vertex for every time a statement is executed) and whose edges are—as in a static dependence graph—the data and control dependencies between these operations. Essentially, the DDG contains every way in which one operation is affected by a previous one (data values, control flow). Specifically, a data dependency exists from j to i if a variable used by i was defined by j and not redefined until it was used by i. A control dependency exists from j to i if the statements s_j and s_i have a control dependency and j is the last execution of s_j before i. A dynamic slice is the set of executed statements that are reachable (traversing edges backwards) from a specific statement; intuitively, the part of the program execution that affected the specified statement (e.g. the value obtained when a variable is read) [1, 35].

[3]There is some leeway in how far statements must be broken down for this analysis and some languages may use the term 'statement' differently.

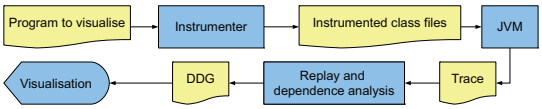

Figure 1: Information flow between parts of Atropos

For our purposes it is more useful to see a dynamic dependence graph as a series of explanations for why each operation in a program was performed and where the data it used came from. A control dependency shows which earlier decision ensured an operation was executed, while the data dependencies of a statement show which statements defined the variables that the statement used.

DDGs can be generalised to concurrent programs by adding *synchronisation* and *communication* dependencies. The communication dependencies represent data values that are transferred between threads, similarly to data dependencies; in Java, this occurs through shared memory. Synchronisation dependencies occur when a thread cannot proceed until an action is taken by another thread; in Java, a `wait` depends on the `notify` that woke it. [24, 36]

2.3.2 Visualisation and DDGs

DDGs can be used for visualisations that aid in finding and understanding cause-effect chains and answering queries about the reasons for events and states in a program. The *Whyline* visualisation has been found useful in some types of debugging situations in educational visual programming environments. The Whyline uses a DDG-based visualisation (together with the other elements of the Alice IDE) to answer queries such as "Why was this statement (not) executed?" or "Why does this variable have this value?"; this is called *interrogative debugging*. The answer is (part of) a slice of a dynamic dependence graph (DDG). In many cases, this DDG enables the programmer to find the reason for incorrect behaviour of a program very quickly by tracing the cause-effect chain from the bug to the symptom backwards along the DDG. The DDG can further be used to quickly navigate to sections of the program and its execution relevant to the bug. [14]

3. DESIGN

A replay system typically generates an execution trace file that can be replayed at a later date. This is particularly useful in our context: a teaching assistant can send a student an execution trace file that demonstrates defects that the assistant found in the student's code, allowing the student to examine the failing execution in more detail.

The execution traces are collected through bytecode instrumentation of class files. For simplicity, compatibility and ease of debugging the instrumenter, this is done by creating instrumented copies of the class files before execution.

The trace files are then used to direct the replay and dependence analysis of program; the result of this is a dynamic dependence graph of the execution. This graph can then be explored through the visualisation.

This process is summarised in Figure 1.

3.1 Generating Execution Traces

In this section we describe the process of creating an execution trace from a user's program. This includes the process of instrumenting the user's program to generate an execution trace and running the instrumented program.

3.1.1 Applicability of Existing Software

In order to decide what information to collect for an execution trace and how to do it, one must determine what sort of program behaviour one is interested in. The systems described in Subsection 2.2 have different advantages and disadvantages in our situation.

Collecting the information needed for program execution visualisation from a model checker is mostly a matter of parsing its output. Collecting the execution trace does not affect how the program is executed, either. In a context where model checkers can be effectively applied, it would be easier to use a model checker to collect executions for visualisation. However, there are several reasons why using a model checker is problematic in this context. One is that model checkers, due to the sheer size of the state space of many programs, cannot be directly applied to many programs; in many cases, they will simply run out of memory without producing any useful results. In our preliminary attempts to use JPF to check students' programs (in the hope of being able to exhaustively check all interleavings, given a certain test input), we failed to get any meaningful results; JPF invariably ran out of memory even when test sizes were cut down to a minimum. For this reason, the model checking approach was abandoned at an early stage. For model checking to be effective in this context, students would have to design their programs with model checking in mind; regardless of the merits of this approach, we did not want to make such extensive changes in how we teach concurrent programming.

As we are interested in understanding and explaining concurrent programs that behave in unexpected ways, unexpected interactions between threads are of great importance. In particular, we want to be able to trace and visualise data races in order to see whether they are benign (i.e. they do not affect the correctness of the program) or not. We also want to be able to examine and explain the consequences of data races. JPF does not appear to support data race simulation; it merely detects situations in which they have occurred.

The replay and runtime analysis-based programs proved problematic due to their modifications to the JVM, instrumentation or their assumptions on the programs they collect information on. JaRec [10] cannot handle data races. DejaVu [4] only runs on one processor and does not seem to handle data races, either. jreplay [28] is based on thread schedules and assumes that it is running on a single processor; again, data races are assumed not to occur. Both of these issues limit the possible executions in such a way that many educationally relevant types of failure can not be examined. The instrumentation used by ODB [15] effectively prevents data races through the synchronisation it uses. RetroVue was ruled out because its source code cannot be accessed. JPaX does not appear to collect all the data values needed for visualisation, nor does the source code seem to be available.

Another problem with most of the existing replay software is that it is geared toward reproducing execution to allow it to be examined in a traditional, state-oriented, debugger. For example, the Java Platform Debugger Architecture is oriented toward examining the current state of a program

and detecting events such as method calls or reaching a specific line for breakpoint purposes, which is inadequate for more complex visualisation of, e.g. data flow or inter-thread interaction.

Although ODB would be appropriate as the basis for implementing the tool we need, we chose to extend our previous tool MVT. In addition to the advantage of familiarity, it has two additional advantages: (a) its instrumenter had already been adapted to generate additional interleavings in concurrent programs for use in the test packages provided to students in our course; (b) it was developed with the intention of extending it to generate DDGs.

3.1.2 Collecting Execution Information

MVT uses BCEL [5] to instrument Java programs for execution history collection; its instrumentation was modified to produce an execution trace of a Java program, consisting of a partially ordered sequence of executed JVM operations and any data manipulated by these other than operand stack values and local variables (i.e. any data that cannot be easily reconstructed by executing simple and deterministic operations). This distinction between deterministic and nondeterministic operations is similar to that of DejaVu, but reading the value of a variable that could have been written to by another thread is treated as a nondeterministic operation, allowing the execution of the program to be correctly reconstructed even in the case of data races.

The test cases are executed using the instrumented code. Execution information is collected in thread-local lists of operations, each identifying which operation was executed in terms of which method it belongs to and its address in the method's bytecode as well as any non-local values read or written, objects referenced and operations known to have happened-before this operation. Operations that only involve a thread's stack are not instrumented, since the limited set of data accessed by these operations allows them to be easily and reliably reconstructed by the visualiser.

When a thread's list of operations exceeds the size limit or the thread or JVM terminates, the thread's list of operations is dumped to disk. At this point, object references are resolved to unique id numbers using a global table of weak references, allowing objects that are no longer in use to be garbage collected. This table is protected by a `synchronized` lock; hence, for reasons described in Subsection 3.1.4, it is desirable to minimise accesses to it while a thread is still running. After the JVM terminates, the lists of operations are compressed in a ZIP file.

The instrumentation is implemented as an additional feature of the MVT-derived instrumenter for the test packages for the programming assignments in our Concurrent Programming course. This allows the assignment-specific test packages to be easily used together with Atropos and also makes it easier to ensure this instrumentation does not conflict with the other instrumentation performed by the system to change how threads interleave.

3.1.3 Partial Ordering and the Happens-before Relation

There are several reasons why the trace is partially ordered. On a multiprocessor system, it is obvious that operations may be performed simultaneously. Hence, it may be impossible to define a total order for a trace. Many operations cannot interact with other threads (for example, any code that touches only local variables); the order of such operations is irrelevant. It also seems useful to get students used to the idea of program execution not being totally ordered.

The execution trace will be generated using bytecode instrumentation of the user's code and, optionally, some unit test code that calls the user's code. The execution trace is primarily intended to explain the user's code, so it must contain the operations performed by the user's code. To help the user see how the calls made to the user's code fit together, the test case code used by the test framework must also be included in the execution trace.

The memory model of Java 1.5 [11, §17.4] describes the circumstances under which inter-thread actions performed by operations (such as reading or writing non-local variables, or acquiring and releasing locks) are guaranteed to be visible to each other. All synchronisation actions (acquiring and releasing locks, starting and joining threads, reading/writing `volatile` variables) are totally ordered in each execution. However, many actions that can be affected by other threads (such as reading non-`volatile` variables) do not have a total order. The *happens-before* relationship is a partial order between inter-thread actions. If an action happens-before another, the effects of the first are visible to the second. Naturally, operations in a thread happen-before each other in the execution order of the thread. Synchronisation between threads induces happens-before relationships between threads; most importantly, releasing a lock in one thread happens-before it is next acquired. Similarly, starting a thread happens-before its first operation.

Even if only one thread is running at some time, it is not practical to determine a total order for all operations, especially when using bytecode instrumentation.

3.1.4 Avoiding Precision Loss from Instrumentation

It is desirable for the instrumentation to not induce additional happens-before relationships between existing actions, as this could eliminate data races that we wish to examine. To do this, the instrumentation must use thread-local data structures to collect information on thread-local operations. Obviously, shared data structures are necessary to collect information on inter-thread interaction. If a happens-before b, the instrumentation can record this by storing an identifier for a in a variable associated with the mechanism used to induce the happens-before relationship with b. The instrumentation can then, after b is executed, safely read this identifier, thanks to the happens-before relationship, at which point the instrumentation for b has identified the happens-before relationship.

For example, for each lock, the identity of its last release operation is maintained. Before the lock is released, this is updated. The updated value can then be read safely after the lock has been acquired to determine the happens-before relationship induced by the lock. In practice, this identifier takes the form of additional fields to track which thread held an object's lock last and when (in terms of its number of operations executed) it released the lock. Before a lock is released, the fields are updated, and after the lock is acquired, they are read to determine the previous owner of the lock and which unlock operation happened-before the lock operation.

While it would seem to be a good idea to add these fields to `java.lang.Object`, this does not work in practice, since

most JVMs (including Sun's HotSpot VM) do not allow additional fields to be added to `java.lang.Object` and our attempts to do so led to crashes, similarly to those described by Georges et al. [10]. Generally speaking, attempting to instrument classes in `java.lang` is problematic, as these classes are often built in to the JVM or the JVM itself makes assumptions about their contents. Hence, these additional fields are added to the classes that are instrumented, as in JaRec. Similarly, each `volatile` field is replaced with a `volatile` field referencing an object containing the value of the original field and additional fields to identify which operation wrote the value; this allows the happens-before relationships to remain the same between the thread while ensuring that both the value and the information collected by the instrumentation is retained.

This approach assumes that `volatile` fields in the instrumented code are only accessed from instrumented code; as long as all the users' code is instrumented this should be true in all relevant scenarios (the standard library code has no business changing fields in user code it has no knowledge of). Similarly, it assumes that non-instrumented code does not lock instances of instrumented classes. In practice, as long as the Reflection API is not used to modify variable values, this should not be a problem. However, it is not unreasonable for a user to create `Objects` to use as locks. Luckily, in most cases (again, the Reflection API is the most obvious exception), Java code cannot tell that an object of a class X has been replaced by another object that is an `instanceof` X with the same state except for some additional fields. This means that all object construction done by the users' code that is of non-`final` non-instrumented classes can be replaced with an instance of a subclass of the original intended class with the fields required for lock use tracking mentioned above added.

`wait`, `notify` and `notifyAll` are handled much like in JaRec: `wait` involves releasing the lock, waiting and reacquiring it, and is instrumented accordingly. As Georges et al. [10] note, `notify` and `notifyAll` do nothing that needs to be recorded to replay the program execution.

3.2 Replay and Dependence Analysis

Rather than performing straightforward replay of a concurrent execution, Atropos replays the program in its own interpreter, which constructs a dynamic dependence graph of the execution. Any thread is allowed to execute for which everything that should have happened-before the current operation has already been executed; in other words, everything is executed in an order consistent with the happens-before order. In the absence of data races, this ensures that whenever a variable is read, it has an unambiguous value and the last write that was performed is the value that was read.

The replay in happens-before order allows a vector timestamp for each operation to be created as described by Mattern [22], with happens-before relationships forming the messages of Mattern's vector clock algorithm.

However, since we allow data races to occur, many different values for a variable may be available for reading at a time. When a read is performed, the corresponding write operation must be found from the set of writes that it is *allowed to observe* [11, §17.4]. A write w can not be observed by a read r if r happened-before w (in which case w would not have been re-executed yet) or if there is an intervening write w' to the same variable such that w happens-before w' and w' happens-before r. In other words, the last write that happened-before the read and any later write may be observed. This set of observable writes can easily be determined by using the vector timestamps described above to find the last operations in each thread that happened-before the read.

Re-ordering of operations may cause writes that are later than reads [11, §17.4.5]; the data dependency must then be determined at a later time. This does not affect the replay itself; it merely means a placeholder must be left for replacement with the correct data dependency after the right write operation has been performed. In this case, it becomes necessary to make sure the write does not happen-after the read, but otherwise the process is the same as above.

Control dependencies are traditionally calculated statically, and in any case, as explained in the following subsection, they are not relevant to our visualisation.

3.3 Visualisation

The primary visualisation around which this tool is based is a dynamic dependence graph that explicitly shows how the results of executed operations depend on other, previously executed, operations.

The DDG representation consists of executed operations, represented by vertices, interconnected by their dependencies, shown as edges. The DDG is intended to support *backward debugging* strategies; in other words, this visualisation is supposed to help the user find the point where a program failed by backtracking from a failure to the point where the program diverged from its expected behaviour.

As the full DDG of a program execution is likely to be very large, only a small portion that the user has explicitly requested is shown. Essentially, the visualisation explains what happened in an operation in terms of previous operations. Once the user has found an operation that does the wrong thing despite being executed at the right time and operating on the right data, he has found code involved in a defect.

As an example in this section, a modified version of an example of an incorrect mutual exclusion algorithm for two threads will be used. The steps to produce Figure 2 from an execution of the following code will be explained.

```
1  /* http://www.pearsoned.co.uk/HigherEducation/
2        Booksby/Ben-Ari/ */
3  /* Second attempt; Modified to exit if critical section
4     counter shows something other than 0 or 1. */
5  class Second {
6   /* Number of processes currently in critical section */
7   static volatile int inCS = 0;
8   /* Process p wants to enter critical section */
9   static volatile boolean wantp = false;
10  /* Process q wants to enter critical section */
11  static volatile boolean wantq = false;
12
13  class P extends Thread {
14   public void run() {
15    while (true) {
16     /* Non-critical section */
17     while (wantq)
18      Thread.yield();
19     wantp = true;
20     inCS++;
21     Thread.yield();
22     /* Critical section */
23     System.out.println("Processes␣in␣critical␣section:␣"
```

```
24          + inCS);
25        if ((inCS > 1) || (inCS < 0)) System.exit(1);
26        inCS−−;
27        wantp = false;
28       }
29      }
30     }
31
32     class Q extends Thread {
33      public void run() {
34       while (true) {
35        /* Non−critical section */
36        while (wantp)
37         Thread.yield();
38        wantq = true;
39        inCS++;
40        Thread.yield();
41        /* Critical section */
42        System.out.println("Processes␣in␣critical␣section:␣"
43          + inCS);
44        if ((inCS > 1) || (inCS < 0)) System.exit(1);
45        inCS−−;
46        wantq = false;
47       }
48      }
49     }
50
51     Second() {
52      Thread p = new P();
53      Thread q = new Q();
54      p.start();
55      q.start();
56     }
57
58     public static void main(String[] args) {
59      new Second();
60     }
61    }
```

3.3.1 Starting Points

The visualisation uses the termination of the threads in the program as starting points. It is assumed that at least one of these corresponds to a failure. This could be:

- A thread terminating abnormally due to an uncaught exception;
- A thread detecting incorrect behaviour and aborting itself or the whole program; unit tests and other assertions usually behave like this;
- A thread being stuck in a deadlock.

By choosing the right thread to examine, the user can work backwards from the thread's termination to the fault that caused it.

Once the user has opened a trace, the list at the top of the window shows the starting points for the DDG. In the example, the final lines of code executed in each of the three threads in the example program, labelled with the thread they occurred in and the description of the operation (see Subsection 3.3.3): the `main` thread constructed a `Second` object and terminated, while the other two threads fail to exclude each other. At least one of these will have terminated on the check of the number of threads in the critical section:

```
Last operation in Thread 5:
  25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
```

Since we are expecting the program never to end at this line and it did, this is clearly a failure and hence a good starting point for exploring what went wrong.

3.3.2 Names

While local variables and classes have names that are unique in context, other entities do not have an obvious name by which they are identified. Objects are referred to as ⟨class⟩-⟨id⟩, where ⟨id⟩ is a positive integer that makes the name unique for each object. Fields and methods of objects and classes are referred to as ⟨object/class⟩.⟨field⟩.

3.3.3 Operations

The operations shown are limited to the operations in the instrumented code (i.e. the user's program and, if applicable, the test from the test package). These operations are all performed by Java bytecode (i.e. there is no native code).

The unit of code usually shown in Atropos as a vertex in the graph is the execution of a line of code. Each vertex is shown as a box. The box contains a textual description of the operation. The information shown in a vertex consists of the number of the executed line and the line of source code, if the vertex corresponds to a line of source code. This is followed by the number of operations executed in the thread at the time the operation was executed. For example, the line that ends the execution of our example in the example trace is:

```
25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
```

The user can choose a vertex and request that the previous line be shown. This enables the user to display the whole program execution one step at a time, although this is seldom a convenient way to do so.

Vertices are arranged in chronological order from top to bottom as a *layered graph* [6]. Specifically, if one operation happened before another, it will be above it. Two vertices being next to each other does not imply they were executed simultaneously except in the sense that neither is known to have preceded the other. Vertices are horizontally positioned by thread (i.e. operations executed by a thread form a column). Threads are ordered by creation time from left to right. Above the vertices that belong to the execution of a method call, the name of the method and the object or class on which it was called is shown, together with the arguments to the call. Indentation is used to indicate nesting of method calls; the further to the right within a thread a vertex is, the deeper nested the call is.

3.3.4 Dependencies

Dependencies between operations are shown as arrows between vertices:

- Data/communication dependencies: data produced by one operation and used by another are labelled with the variable name (where applicable) and the value.
- Control dependencies: The last branching operation, such as `if` or `while` is shown. This is easier to understand and calculate than the traditional definition.

By default, to keep the visible graph manageable, all dependencies of vertices are hidden. To show a dependency, the user must explicitly request that it be shown. This may add a vertex to the graph.

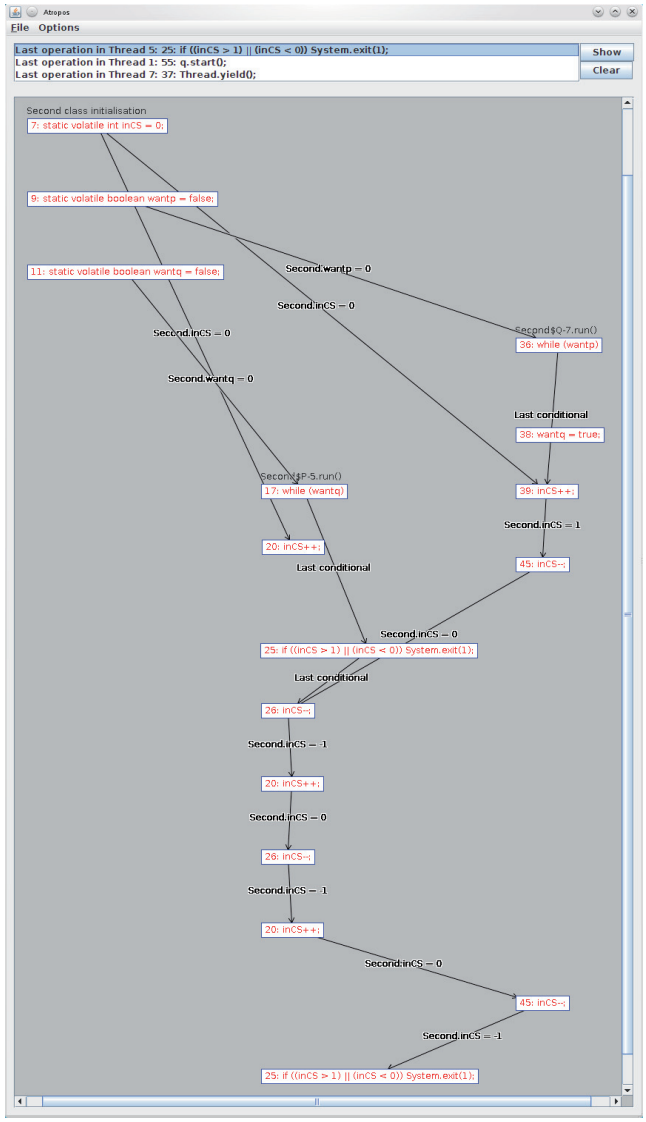

Figure 2: A screenshot of Atropos at the end of the exploration of the example trace

To show where a value read by an operation came from, one can choose the relevant value from the list of data sources of the vertex. Similarly, to show where a value written by the operation was used, one can select the value from the list of data uses of the vertex. There are also corresponding commands to show all data sources and uses at once. All of these commands show data dependencies of the specified operation.

Figure 2 shows how one can go from an operation that uses `inCS` to the operation that wrote that value; repeatedly doing this allows one to find the points where the critical section was entered and exited, and shows how the value of `inCS` changes throughtout the execution. In particular, this shows that `inCS` was (somewhat surprisingly) -1 when the program decided to terminate and shows how this was caused by `inCS` being incremented in two threads. This also shows that P has toggled it between -1 and 0 instead of 0 and 1; furthermore, on the sixth step back, we see that Q has left its critical section at line 45 and correctly given `inCS` the value of 0. The very fact that we have both threads leaving their critical section without one of them entering in between show that they both were in the critical section at the same time; therefore, mutual exclusion is not being maintained. This also means that `inCS` still has an unexpected value.

Continuing back two more times shows us where `inCS` got its original value of 0. The initial value of `inCS` was obviously correct, but there is no sign of the expected increment by P when it first entered the critical section.

To examine why an operation was executed, one can backtrack to the previous conditional. This can be used to find, for example, when the decision was made to allow a process into its critical section. In particular, we want to know how P and Q got into their critical sections at the same time. Requesting the previous branch will show where they exited their busy-wait loops.

Naturally, the reason for both threads unexpectedly entering their respective critical sections can be found by examining the source of the data used as a loop condition. Examining where the values of `wantp` and `wantq` involved in these branches came from will show that both variables were false at the time they were read. In other words, what is wrong in the program is that both threads may see that the other thread does not want to proceed; the defect is in the condition that allows the threads to enter the critical section (or in the setting of the flags). Hence they both enter the critical section simultaneously, allowing two concurrent changes to `inCS`, resulting in this variable having an unintended value. One can indeed find the missing increment of `inCS` by working forwards from the value of `inCS` it read (the initialisation) by checking where this value was used. This confirms that two increments of `inCS` (one in P and one in Q) read the same value of `inCS` (i.e. they executed simultaneously) and the increment in Q overwrote the value written by P, causing `inCS` to have a value one lower than expected.

4. EVALUATION

Although a thorough evaluation of the effectiveness of Atropos is beyond the scope of this article, a few observations that are relevant for its future development can be made. We also want to evaluate whether the system described in Section 3 actually succeeds in creating the intended visualisation, i.e. whether it satisfactorily answers research question 4. In order to do this, we must determine whether the system can feasibly be used to trace the failures of the students' programs. This involves both issues of performance (trace size and processing overhead) and whether the failures still manifest when a trace is produced. We also touch on research question 3; whether the visualisation enables students to find and understand defects.

The tests described in this section were all done on a 64-bit Ubuntu 10.04 workstation with an Intel Core 2 Quad Q9400 CPU and 4 GB of RAM.

4.1 Size of Traces

The execution traces are, unsurprisingly, very large in the case of stress tests. In the first programming assignment of our concurrent programming course in 2010, the stress test traces from the programs submitted by our students had an uncompressed mean size of 1.7 GB, a maximum size of 13 GB and a median size of 0.93 GB. When compressed, the mean was 110 MB, the maximum 460 MB and the median 85 MB. Although this means some temporary disk space is necessary (roughly speaking, the amount of disk space that can be filled within the time limit for the test execution; in this case 1000 seconds), the final file size is manageable. Also, most of the executions that were aborted due to incorrect output had traces of only 2–3 MB (uncompressed), since the stress test caused a failure early in the execution. This suggests that trace size is unlikely to be a problem if stress tests are divided into smaller parts to avoid having to store and replay several minutes of correct behaviour that is of little relevance when debugging.

Performing the dependence analysis is often problematic since Atropos constructs the entire DDG in memory. In practice, this means that traces may not be larger than a few megabytes. Again, this can be mitigated by finding test cases in which failures occur and are detected as quickly as possible.

4.2 Performance Loss Caused by Instrumentation

Using the same programs as in the previous subsection and considering only the executions that completed successfully (since the failures were not necessarily in the same place), instrumenting the programs to trace the execution caused the time used by the stress test to increase on average to 10.2 times the original execution time; the median slowdown factor was 5.27. One test was only 25% slower; the Reactor implementation in question is otherwise efficient but creates a new thread for each event, causing an overhead that dwarfs that of the instrumentation. The two worst slowdowns were by factors of 69.5 and 35.4, both with Reactor implementations that allow each thread to feed unlimited amounts of data into the buffer without ever waiting for it to be processed. While there is very little overhead from switching between threads in this type of solution, they run the risk of running out of memory.

We conclude that, while the instrumentation introduces noticeable overhead, much of this is masked by overhead from e.g. creating or switching between threads.

The mean time for the instrumented stress tests was 127 seconds, the median 75.5 seconds, the minimum 44.9 seconds and the maximum 739 seconds. Without instrumentation, the mean was 17.9 seconds, the median 15.0 seconds, the minimum 3.8 seconds and the maximum 59.9 seconds. This means that for most students, the stress test will be completed within a reasonable time despite the instrumentation; we doubt that students will have new versions of their code to test every few minutes.

4.3 Effect on Failure Occurrence

Evaluating the precision of concurrent program replay, i.e. how much the execution changes when the program execution is traced, is a very complex issue. However, in our setting, the clearly most important facet of precision in our case is whether the execution tracing prevents failures from manifesting in incorrect programs. To evaluate this, we reran the stress test 10 times with and without instrumentation on the Reactor implementations in which a failure caused by a race condition was detected by the test package (without the instrumentation). All three programs exhibited race conditions that consistently caused the stress test to fail both with and without the tracing. One of the programs, a very inefficient implementation, was slow enough that the overhead from the instrumentation consistently caused the test to time out before a failure occurred. This can be remedied by adjusting the timeouts to compensate for the instrumentation overhead.

4.4 The Use of Atropos by Students

Preliminary results from the evaluation of the use of Atropos by students in our course suggest that they were able to use Atropos to extract information to support their own debugging process and help clear up some misunderstandings of, for example, what threads exist in a running program. This happened even though the students often used a debugging style that relied heavily on examining source code, trying to reason about it statically and rewriting suspicious parts of the program (cf. [9]). This suggests that while Atropos was designed for backward debugging, it can also support other debugging strategies.

The students also turned up several minor usability issues that left them frustrated even before they actually got to see the visualisation. While splitting the testing and visualisation into two separate steps makes it easier to do testing as an unattended batch job in order to examine the interesting traces later, students sometimes change the code they have written after executing it and generating a trace, causing problems in the replay. While it is possible to detect this situation using e.g. checksums, the problem can also be avoided by including the source and class files in the execution trace file, ensuring that the replay is done with the same code as the execution trace was produced from. This has the additional advantage of making the trace file self-contained.

5. CONCLUSION

We have presented Atropos, a system intended to help students understand and debug concurrent Java programs by allowing them to examine execution traces through a dynamic dependence graph-based visualisation. While care must be taken to keep the size of these execution traces manageable, this can easily be done by splitting large test runs into smaller parts. The overhead from the instrumentation causes some minor problems, but they can be worked around by allowing more time for tests to execute. Students are able to use Atropos effectively, even when their debugging approaches do not seem to fit the intended approach. This can probably be improved by teaching students debugging approaches for concurrent programs more explicitly and by integrating Atropos further into our teaching of concurrent programming.

5.1 Future Work

Problems with applying Atropos in an educational context stem from the size of the execution traces. Splitting long-running tests will help mitigate this, but an alternative approach would be to collect the corresponding information from a model checker such as JPF. The traces produced by a model checker would almost certainly be shorter than those produced by stress testing. Another alternative is to allow users to manually specify in which order operations are executed and what happens in data races; this would be especially useful for teachers to demonstrate ways in which a simple program can go wrong.

To assist students unfamiliar with backward debugging, it would be helpful if Atropos itself provided more explicit guidance on how it can be effectively applied. Similarly, it would be worthwhile to investigate how to support forward debugging strategies more effectively, perhaps by supporting forward navigation along dependencies to complement the backwards navigation already available.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 60–73, New York, NY, USA, 1991. ACM. ISBN 0-89791-449-X. doi: http://doi.acm.org/10.1145/120807.120813.

[2] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian Rover software. *Formal Methods in System Design*, 25 (2-3):167–198, 2004. doi: http://dx.doi.org/10.1023/B:FORM.0000040027.28662.a4.

[3] J. Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[4] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, San Fransisco, USA, Apr. 2001. IEEE Computer Society.

[5] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, Apr. 2001.

[6] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[7] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330. Springer-Verlag, 2000.

[8] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/248448.248456.

[9] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, June 2008.

[10] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software — Practice and Experience*, 34(6):523–547, May 2004.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[12] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004. doi: http://dx.doi.org/10.1023/B:FORM.0000017721.39909.4b.

[13] G. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[14] A. J. Ko and B. A. Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the 2004 conference on Human factors in computing systems*, pages 151–158. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985712.

[15] B. Lewis. Debugging backwards in time. In M. Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging*, Ghent, Belgium, Sept. 2003.

[16] K.-P. Löhr and A. Vratislavsky. JAN - Java animation for program understanding. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003)*, pages 67–75, Oct. 2003.

[17] J. Lönnberg. Defects in concurrent programming assignments. In A. Pears and C. Schulte, editors, *Proceedings of the Ninth Koli Calling International Conference on Computing Education Research (Koli Calling 2009)*, pages 11–20, Koli, Finland, 2009. Uppsala University.

[18] J. Lönnberg and A. Berglund. Students' understandings of concurrent programming. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 77–86, Koli, Finland, 2008. Australian Computer Society.

[19] J. Lönnberg, A. Korhonen, and L. Malmi. MVT — a system for visual testing of software. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'04)*, pages 385–388, May 2004.

[20] J. Lönnberg, A. Berglund, and L. Malmi. How students develop concurrent programs. In M. Hamilton and T. Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, pages 129–138, Wellington, New Zealand, 2009. Australian Computer Society.

[21] J. Lönnberg, L. Malmi, and A. Berglund. Helping students debug concurrent programs. In A. Pears and L. Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*, pages 76–79, Koli, Finland, 2009. Uppsala University.

[22] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, Oct. 1988. Elsevier.

[23] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In S. Diehl, editor, *Software Visualization*, pages 163–175, Dagstuhl Castle, Germany, 2002. Springer-Verlag.

[24] D. Mohapatra, R. Mall, and R. Kumar. An efficient technique for dynamic slicing of concurrent Java programs. In S. Manandhar, J. Austin, U. Desai, Y. Oyanagi, and A. Talukder, editors, *Applied Computing — Proceedings of the Second Asian Applied Computing Conference (AACC 2004)*, volume 3285 of *Lecture Notes in Computer Science (LNCS)*, pages 255–262. Springer, 2004.

[25] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, pages 176–190, Dagstuhl Castle, Germany, 2002. Springer-Verlag.

[26] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: A flexible framework for creating software model checkers. In *Proceedings of Testing: Academic & Industrial Conference — Practice And Research Techniques*, June 2006.

[27] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. doi: http://dx.doi.org/10.1109/ISSRE.2004.1.

[28] V. Schuppan, M. Baur, and A. Biere. JVM independent replay in Java. In *Proceedings of the Fourth Workshop on Runtime Verification (RV 2004)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 85–104. Elsevier, Jan. 2005.

[29] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.

[31] A. von Mayrhauser and A. M. Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179, New York, NY, USA, 1997. ACM Press. doi: http://doi.acm.org/10.1145/266399.266414.

[32] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[33] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[34] A. Zeller. Animating data structures in DDD. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 69–78, Porvoo, Finland, 2001. University of Joensuu.

[35] J. Zhao. Dynamic slicing of object-oriented programs. In *Technical Report SE-98-119*, pages 17–23. Information Processing Society of Japan, May 1998.

[36] J. Zhao. Multithreaded dependence graphs for concurrent Java programs. In *In Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23. IEEE Computer Society, 1999.