Publication V

Jan Lönnberg, Mordechai Ben-Ari and Lauri Malmi. Visualising Concurrent Programs with Dynamic Dependence Graphs. In *Proceedings of 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*, 4 pp, Williamsburg, Virginia, USA, September 2011.

Visualising Concurrent Programs with Dynamic Dependence Graphs

Jan Lönnberg
School of Science
Aalto University
Espoo, Finland
Email: jlonnber@cs.hut.fi

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

Lauri Malmi
School of Science
Aalto University
Espoo, Finland
Email: lma@cs.hut.fi

***Abstract*—Atropos is a software tool for visualising concurrent program executions intended to help students debug concurrent programs and learn how concurrency works. Atropos supports a slicing debugging strategy by providing a visualisation of dynamic dependence graphs that can be explored to trace the chain of events backwards from a symptom to its cause. In this paper, we present the reasoning behind the design of Atropos and summarise how we evaluated it with students.**

***Keywords*-Visual debugging; Program visualization; Dynamic dependence graph; Atropos; Slicing;**

I. Introduction

Debugging concurrent programs is difficult for several reasons. Many bugs are hard to replicate due to nondeterminism. They also often involve unexpected interactions between threads that are hard to trace.

These same issues also make it hard to understand and learn how concurrency works in practice. We would like to help our students understand what is going on in concurrent programs so they can fix their programs and form more complete understandings of concurrent programming.

In this paper, we describe our design for a visualisation tool for debugging concurrent programs, *Atropos*.[1] Atropos visualises a trace of a concurrent Java program as a graph that shows how the operations performed during the execution interrelate. Essentially, for every line of code that was executed, Atropos can show why it was executed and where it got the data it used from. The details of how Atropos collects and replays concurrent execution and issues related to performance are discussed in another paper [1]. We also briefly describe how we evaluated Atropos; this will be described in more detail in a forthcoming paper [2].

In Section II we present the work on debugging strategies that underlies our design. We then use this as a basis for describing the design of Atropos in Section III. The results of our evaluation of Atropos and their implications for the design of Atropos are presented in Section IV.

II. Related Work

A. Debugging Strategies

Eisenstadt [3] analysed difficult bugs encountered by professional programmers. The most common reasons he found for bugs being hard to find are:

- Cause and symptom are separated in space or time;
- The incorrect behaviour does not consistently manifest itself;
- The programmer gets stuck by misinterpreting what he sees.

By far the most common approach to tracking down these bugs is to gather information on the execution of the program through a variety of means, including single-stepping, adding `print` statements to selected points in the program, adding conditional breakpoints to the program and inspecting the data when the breakpoint is triggered and comparing dumps of the program's state [3].

Metzger [4] describes a wide range of debugging strategies. Most of the strategies focus on dividing a program into parts and ruling them out as the site of the bug. The *deductive-analysis* and *inductive-analysis* strategies are based on generating hypotheses and checking them. The success of these strategies relies almost entirely on how good hypotheses one generates.

When looking for reasons for a program's misbehaviour, 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 variable value. 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* [5].

Slices can be calculated from 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 [6]. A dynamic slice is based on a *dynamic dependence graph* (*DDG*). A DDG is a directed graph whose vertices are executions of statements and whose edges are the data and control dependencies between these operations. A DDG is essentially 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 operations wrote the variables that the statement read.

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

A slicing strategy is very hard to apply without tool support, since calculating slices by hand can quickly become very tedious. Complex cause-effect relations that can be computed (e.g. data flow links) should be computed by the debugger rather than by hand [3], [4].

B. *Visual Debugging*

Most visual debuggers, e.g. DDD [7], concentrate on individual threads and can only show the current state of the program (which is especially problematic in concurrent programs, as replicating a failure may be difficult).

RetroVue [8], with its tree view of all executed operations, ability to examine all previous states of the program and thread display showing lock interactions between and execution times of threads, is a clear exception to this. However, it does not aid the programmer much in finding interrelated operations.

The Whyline [9], which uses DDGs to explain to novices the reason why a program did something (wrong), addresses the problem of explaining relationships, but is limited to low-level explanations of simple causal chains in a limited beginners' environment.

A few debuggers and program visualisation systems have been designed for concurrency. Several use sequence diagrams to display method calls. JaVis [10] adds collaboration diagrams to show interactions between objects. These diagrams have a level of detail suitable for debugging, but become cumbersome for complex executions.

C. *Bugs in Students' Concurrent Programs*

Most of the literature on students' errors in programming assignments focuses on novice programmers working with sequential programs. We analysed the defects in the programs our students wrote for the programming assignments in our concurrent programming course. We found that most bugs involved nondeterministic behaviour and that many difficult bugs involved unexpected interactions between two different parts of the student's program or unexpected interleavings of threads [11].

D. *How Students Debug*

Fitzgerald et al. [12] found that students use mental tracing (with and without `print` statements), hand tracing and tracing using the debugger and that students exhibit both *forward* (tracing) and *backward* (causal) reasoning.

Kiesmüller [13] and Yehezkel et al. [14] examined how the use of visualisation tools in an educational context affects students' activities when defining and testing a program. Yehezkel et al. found that students working without the EasyCPU visualisation primarily used the data input and instant run operations with trial and error, while the students using EasyCPU tended to run the program step by step and investigated the program's execution more. Kiesmüller identified different problem-solving strategies, such as bottom-up, trial and error and hill climbing.

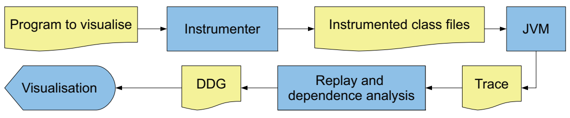


Figure 1. Information flow between components of Atropos

III. The Design of Atropos

Atropos is primarily intended for a scenario in which a student's code has been observed to fail by a student who is developing or testing her code or by a teaching assistant who is assisting her or assessing her code. In either case, the student's code (including test code) has been run with instrumentation that collects an execution trace in a file that can be used immediately or saved for later use by another person.

The expected goal of the user of the visualisation tool is assumed to be to find the defect in the student's code that caused the program failure. The user can be a teaching assistant who needs to find the defects in a student's program to decide how the student erred in order to give the student a meaningful grade and useful feedback (which may include the trace file). Alternatively, the user is a student trying to find a defect in her own program in order to fix it and learn from her mistakes.

In any case, the visualisation is intended to support its user in finding a failure in a trace of an execution of the student's code using a known symptom as a starting point and hence identify the defect in the code that caused the failure.

How the different parts of Atropos fit together and what information they exchange is summarised in Figure 1.

How Atropos can be used in teaching is discussed in more detail in [15].

A. *Why DDG?*

As noted in Section II-A, the slicing strategy provides a clearly-defined systematic way to trace backwards through a program from a symptom to the failure that caused it and hence identify the underlying defect. It also requires tool support to be effective.

DDGs are of particular interest for concurrent programs, as interactions between threads are clearly shown as edges. This means that a DDG can help students identify unexpected interactions between threads. By showing how different parts of a program interact, they can also help bridge cause-effect chasms and isolate relevant information from a large trace. Execution traces also remove the need to re-execute the program.

Previous studies of the use of DDGs to support debugging in an educational context (albeit at an introductory level) have had encouraging results [9].

B. *Visualising the DDG*

Since a DDG is a graph, one can make use of well-known visual representations for graphs in visualising a DDG. Operations and dependencies must be labelled so that the user can identify them. Directed labelled graphs are often represented as labels (often surrounded by a rectangle or similar container) representing vertices connected by arrows representing edges.

Operations are by default grouped together by lines, as is traditional in debuggers. Each operation is labelled with the number of the line of code and the line itself. The shape of the line of code makes a rectangle a natural choice of container.

In order to show execution order in an intuitive fashion, vertices are arranged chronologically from top to bottom as a *layered graph* [16]. If one operation happened before another, it will be above it. Two vertices being next to each other means that neither is known to have preceded the other. The vertical layout uses space more efficiently than a horizontal one when there are many vertices in a thread and labels are long.

Vertices are horizontally positioned by thread. 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. An earlier design used nested boxes to represent stack frames, but this was deemed to be too cluttered. Dependencies between operations are shown as arrows between vertices. Data produced by one operation and used by another are labelled with the variable name (where applicable) and the value.

Since students work with concurrent data structures such as tuple spaces at multiple levels of abstraction [17], we provide a way for users to raise the level of abstraction and remove implementation details from the visualisation by grouping together lines executed as part of a method call to a single vertex.

C. Navigating the DDG

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. The visualisation uses the termination of the threads in the program as starting points. It is assumed that at least one of these is a symptom, such as a deadlock, an uncaught exception or failed assertion. By choosing the right thread to examine, the user can work backwards from the thread termination to the failure that caused it.

The list at the top of the Atropos window shows the starting points for the DDG. To keep the visible graph manageable, all dependencies of vertices are hidden unless the user requests that they be shown, which may cause more vertices to be shown. 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 commands to show all data sources and uses. The last branching operation, such as `if` or `while` can be shown. This is easier to understand and calculate than a control dependency. Figure 2 shows an example trace.

IV. Evaluation

We summarise the results of our evaluation of Atropos [2] that are most relevant to the future development of Atropos here. We analysed the activities of students in our Concurrent Programming course performing a debugging task using Atropos.

The students succeeded in extracting some useful information from Atropos that helped clear up some misunderstandings of e.g. what threads exist in a running program, but they failed to effectively navigate the DDG due to difficulties caused by the visualisation displaying implementation details, and, in one case, the sheer number of data dependencies of a method execution. Grouping operations by method execution is specifically intended to address the former case, but is apparently not something students can easily discover, especially since the examples of using Atropos did not require this. In both cases, one can argue that the problem is that the students are being shown what happened in a data structure instead of how it was used.

We have three proposed approaches to helping students navigate execution traces. One is to complement the DDG with an overview of the program's execution trace such as the tree view of RetroVue [8]. Another is to provide the ability to examine all operations done on a particular variable. Both approaches would provide an overview of program execution that some students tried to achieve by repeatedly requesting the previous line or branch. A third option would be the ability to examine the data structures in the program at a certain time as in most debuggers. This would enable the user to check, for example, whether several values form a consistent state, which is hard to do in Atropos.

References

[1] J. Lönnberg, M. Ben-Ari, and L. Malmi, "Java replay for dependence-based debugging," in *Proceedings of PADTAD IX — Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. Toronto, Canada: ACM, Jul. 2011, pp. 15–25.

[2] J. Lönnberg, L. Malmi, and M. Ben-Ari, "Evaluating a visualisation of the execution of a concurrent program," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling 2011)*. Koli, Finland: ACM, Nov. 2011, in press.

[3] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[4] R. C. Metzger, *Debugging by Thinking*. Elsevier, 2004.

[5] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

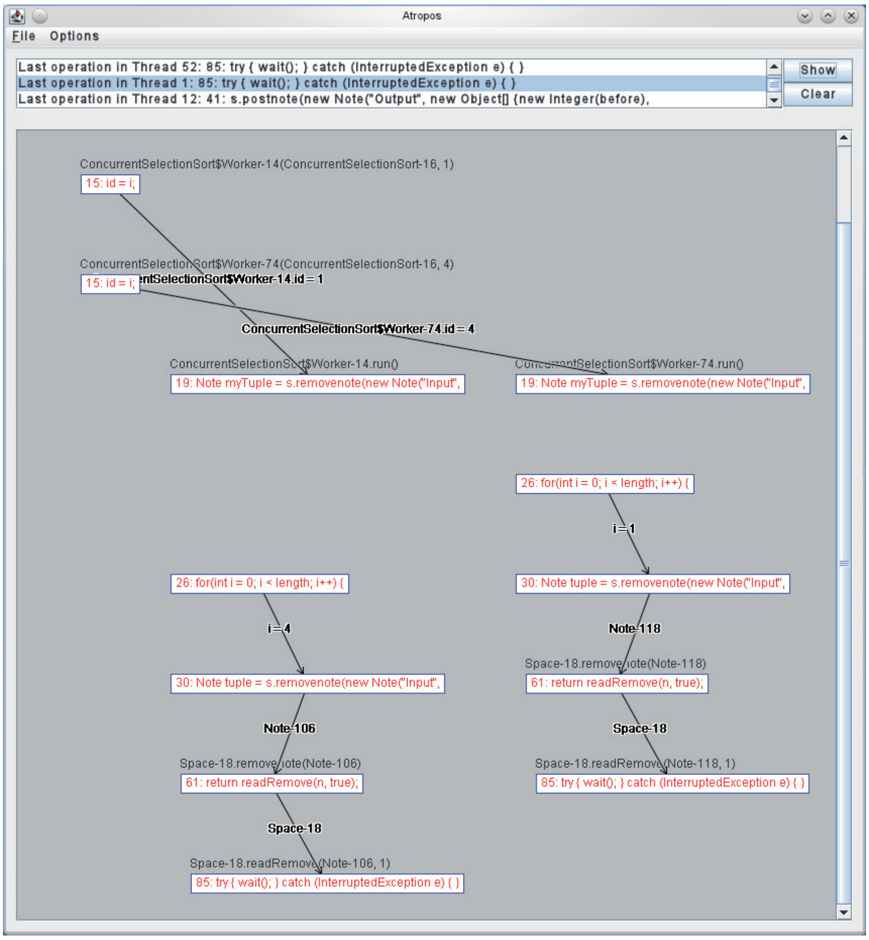
[6] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers," in *TAV4: Proceedings of the symposium on Testing, analysis, and verification*. New York, NY, USA: ACM, 1991, pp. 60–73.

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

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

[9] 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*. ACM Press, 2004, pp. 151–158.

[10] K. Mehner, "JaVis: A UML-based visualization and debugging environment for concurrent Java programs," in *Software Visualization*, S. Diehl, Ed. Dagstuhl Castle, Germany: Springer-Verlag, 2002, pp. 163–175.



```
Last operation in Thread 52: 85: try { wait(); } catch (InterruptedException e) { }
Last operation in Thread 1: 85: try { wait(); } catch (InterruptedException e) { }
Last operation in Thread 12: 41: s.postnote(new Note("Output", new Object[] {new Integer(before),
```

Atropos
File
Options
Show
Clear
ConcurrentSelectionSort$Worker-14(ConcurrentSelectionSort-16, 1)
15: id = i;
ConcurrentSelectionSort$Worker-74(ConcurrentSelectionSort-16, 4)
15: id = i;
ConcurrentSelectionSort$Worker-14.id = 1
ConcurrentSelectionSort$Worker-74.id = 4
ConcurrentSelectionSort$Worker-14.run()
19: Note myTuple = s.removenote(new Note("Input",
ConcurrentSelectionSort$Worker-74.run()
19: Note myTuple = s.removenote(new Note("Input",
26: for(int i = 0; i < length; i++) {
i = 1
30: Note tuple = s.removenote(new Note("Input",
Note-118
Space-18.removenote(Note-118)
61: return readRemove(n, true);
Space-18
Space-18.readRemove(Note-118, 1)
85: try { wait(); } catch (InterruptedException e) { }
26: for(int i = 0; i < length; i++) {
i = 4
30: Note tuple = s.removenote(new Note("Input",
Note-106
Space-18.removenote(Note-106)
61: return readRemove(n, true);
Space-18
Space-18.readRemove(Note-106, 1)
85: try { wait(); } catch (InterruptedException e) { }

Figure 2. A screenshot of Atropos

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

[12] 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*, vol. 18, no. 2, pp. 93–116, Jun. 2008.

[13] U. Kiesmüller, "Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education," *Trans. Comput. Educ.*, vol. 9, no. 3, pp. 1–26, 2009.

[14] C. Yehezkel, M. Ben-Ari, and T. Dreyfus, "The contribution of visualization to learning computer architecture," *Computer Science Education*, vol. 17, no. 2, pp. 117 – 127, Jun. 2007.

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

[16] 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.

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