Publication VI

Jan Lönnberg, Lauri Malmi and Mordechai Ben-Ari. Evaluating a Visualisation of the Execution of a Concurrent Program. *Proceedings of the Eleventh Koli Calling International Conference on Computing Education Research*, pp 39–48, Koli, Finland, November 2011.

Evaluating a Visualisation of the Execution of a Concurrent Program

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

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

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

ABSTRACT

In this paper we present a study of how students make use of *Atropos*, a new visualisation system that is based upon dependence graphs, while debugging concurrent programs. We examine how students work by identifying their operation foci, and use these as a basis for identifying the students' approaches to debugging concurrent programs. We also identify the types of understanding of a concurrent program Atropos helped them to gain, and the situations in which they did not manage to get the information they wanted from Atropos. We use the latter as a starting point for discussing improvements to make Atropos more useful for students.

Keywords

Concurrent programming, Program visualisation, Operation foci, Atropos

1. INTRODUCTION

Many modern software systems are *concurrent*; they involve simultaneously executing or unpredictably interleaved processes. Finding and correcting defects (*bugs*) is particularly difficult when the program is concurrent. 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.

Debugging is usually done by executing the program with different inputs and examining its intermediate and/or final results, either using additional debugging code or through a program intended to aid debugging (a *debugger*). Some debuggers employ *visualisation* to make the data shown easier to understand. 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 that are 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.

Before taking steps to correct a bug, it must be found. There are a variety of approaches to this, including *testing*, *model checking*, *correctness proofs* and various formal and informal approaches based on reading the code and looking for mistakes.

1.1 Goal of This Study

The goal of this study is to gain an understanding of the behaviour of students writing and debugging concurrent programs. In particular, we want to understand their processes for writing a program and for ensuring its correctness. We want to investigate how their way of working changes when they use *Atropos*[1], a program visualisation tool that we designed to help them with this task. This way, we can identify problem areas in their approaches to developing software in order to devise ways to help them either through software or different teaching. We can also evaluate Atropos and identify the ways in which it can help students. Furthermore, this evaluation also helps us identify aspects of Atropos we can improve on.

Based on these goals, the research questions are:

1. How does the visualisation of program executions help students understand and debug defective concurrent programs?
2. How could the visualisation be improved to further help students in these tasks?

To answer these research questions, we look at how students actually use Atropos. These additional questions are:

3. What information and understandings do students get when using Atropos?
4. What information are they looking for? In particular, what information do they try to get, but fail?
5. How are they using Atropos? What operations are they using? What is their strategy for finding the information they want?

[1]Web site at: `http://www.cse.hut.fi/en/research/LeTech/Atropos/`

1.2 Related Work

Since the work described in this paper can be seen both as an evaluation of a software tool and a study of how students program and debug, work on these two subjects is presented here.

1.2.1 *How Students Debug*

As in many subfields of computer science education, most of the work on how students debug focuses on novice programmers. Fitzgerald et al. [6] used interviews including both a programming exercise and a debugging exercise to investigate the debugging skills and approaches of novice programmers. Strategies used by the students included mental tracing (with and without `print` statements), hand tracing and tracing using the debugger. The students exhibited both *forward* (tracing) and *backward* (causal) reasoning. The latter was less prevalent than the former, possibly because the students were not familiar with the code they were debugging.

Ahmadzadeh et al. [1] examined programs compiled by students working on a similar debugging task. They found that weak debuggers had difficulties applying their knowledge of programming and debugging unless they were working with programs with a familiar structure. While most good programmers were better at debugging than weak programmers, the good programmers who were weak debuggers did not appear to understand the program to debug well enough to find or fix the bugs.

Murphy et al. [18] took a different approach by focusing on the interaction between students doing pair debugging. They hypothesised that pairs of students who engage in *transactive discussion*—"a conversational mode in which participants respond to their partner's statements to clarify their own understanding or reasoning"—are more successful in debugging. They found that pairs that talked more and used critique transactions tended to eliminate more bugs.

1.2.2 *Evaluating Software Visualisation in an Educational Context*

Many empirical evaluations have been made of software visualisation, and it is clearly beyond the scope of this article to present them all. Hundhausen et al. [8] describe in their meta-study many different evaluations of algorithm visualisations based on empirical techniques. Many of these are controlled experiments that attempt to determine whether changing e.g. the learning or debugging medium affects a measure of success such as post-test accuracy or debugging time by comparing different groups of students. Of the 24 experiments in their corpus, eleven showed a significant positive effect on students from the use of software visualisation.

Kiesmüller [11] and Yehezkel et al. [20] have examined how the use of visualisation tools in an educational context affects students' activities when defining and testing a program. In both cases, the activities of the students were recorded and the focus of the students' operations (what activities the students performed) and—in the latter study—the students' conversation (what the students talked about) was determined. Yehezkel et al. found that students working without the EasyCPU visualisation primarily used the data input and instant run operations with a strategy of 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.

Isohanni and Knobelsdorf [9] examined how students make use of the program visualisation tool VIP by interviewing them and observing and video recording them completing a short programming assignment. They found that when instructed to use VIP, the students ran their program in VIP. However, not all students used VIP to examine the program execution. Only one used VIP as intended (single stepping).

2. SETTING

This work is centred around the Concurrent Programming course at Aalto University[2]. The goal of this course is to teach students the principles of concurrent programming: synchronisation and communication mechanisms, concurrent and distributed algorithms and concurrent and distributed systems.

Most students have completed a bachelor's degree or a roughly equivalent part of a master's degree. About 50–80 students participate each year.

The course includes a set of exercises based on exercises from the course's textbook by Ben-Ari [2] as well as two programming assignments and an exam. Together, the exercises form 10 % of the course grade, the programming assigments 15 % each and the exam 60 %.

The first programming assignment (*Reactor*) concentrates on the Reactor design pattern [19].

In the second assignment (*Tuple space*), the students implement a simple tuple space [7] containing only blocking `get` and `put` operations on tuples implemented as `String` arrays. They are to do this using Java synchronisation primitives and use this tuple space implementation to construct the message passing section of a distributed chat server.

2.1 Atropos

Atropos is a visualisation tool intended to display information relevant to understanding the behaviour of concurrent Java programs, especially when they do not behave as expected. Atropos is entirely a post-mortem analysis tool; you cannot control program execution in any way through Atropos. The input to Atropos is an execution trace of a Java program, containing all the information Atropos needs to reconstruct the execution.

We have designed Atropos, based on our previous research on students' bugs in concurrent programs [13] and on how students develop and understand concurrent programs [17, 16], to assist in the learning and teaching of concurrent programming in Java with a focus on using basic concurrency constructs to construct simple concurrent programs and classes (on the order of a few hundred lines).

The target audience of Atropos consists of two groups: students of concurrent programming and their teachers. The primary use case is that the program to be visualised was written by students and misbehaves in some way, such as locking up or printing the wrong results. Students can use Atropos to find out why their programs do not work, helping them to correct their programs and possibly learn something about how concurrency works in Java. Similarly, teachers can use Atropos to help identify errors students have made in writing a program, which is useful when assisting and

[2]Course web site at: `https://noppa.aalto.fi/noppa/kurssi/t-106.5600/`

assessing students.

As noted in Subsection 1.2.1, students cannot effectively debug a program which they do not understand properly. In particular, the results of Fitzgerald et al. [6] suggest that their debugging strategies are limited when they do not understand the program properly. Atropos is intended to help students apply a backward debugging strategy even when they do not understand the program fully.

When Atropos is used for debugging, the debugging process starts when a test fails, i.e. an execution of the program resulted in incorrect behaviour. Atropos is based on the idea of debugging backwards; starting from the symptom and working backwards to the fault. Hence, the *starting points* shown by Atropos are at the end of a thread's execution, whether it ended normally, threw an uncaught exception or never terminated.

The visualisation Atropos uses to show the user what happened in the program is based on a *dynamic dependence graph* (*DDG*). A DDG is a directed graph. The vertices of a DDG are executions of lines of code or blocks of code formed by grouping executions of lines together. The edges of a DDG are the *data* and *control dependencies* between these operations. The control dependencies of an operation are why the operation was executed in the first place and the data dependencies are the information the operation acted on. Essentially, the DDG contains every way in which an operation is affected by an earlier one. This enables the user to trace the causal chain, backwards and step-by-step, from a symptom to a cause.

Previous studies of the use of DDG-based visualisations to support debugging in an educational context (albeit at an introductory level) have shown a dramatic reduction in debugging time [12], and we expect that the DDG-based visualisation can be particularly useful in finding and understanding incorrect program behaviour involving unexpected interactions between concurrent threads.

The design of Atropos is described in more detail in [15] and its implementation in [14].

2.2 Exercises

The exercises in the Concurrent Programming course focus primarily on examining the possible behaviour of concurrent programs and on writing programs to solve specific tasks. To allow for more concrete work on the exercises, Java is used in addition to the more abstract concurrency model used by Ben-Ari.

In the Autumn 2010 instance of the course, there were five rounds of exercises, one each week. Two or three sessions with at least one teaching assistant present were arranged for each round. Students were allowed to choose which sessions to participate in. The topics of the rounds were: Concurrency models and critical sections, Semaphores, Monitors, Channels and Tuple spaces.

Each exercise session consisted of four tasks, most of which were exercises from (or adapted from) the textbook [2].

In each session, the assistants presented the exercises and any applicable tools, and then left the students to work on the exercises in pairs; when called on by the students, they assisted the students and evaluated their work. The students presented their solutions to the assistants who then provided feedback on the students' solutions which the students could use to improve their solutions.

3. METHODOLOGY

The study uses a mixed-method research design [10] that is primarily qualitative, with some quantitative analysis to support the conclusions of the qualitative analysis.

3.1 Setup

The study was performed during the two sessions for the last round of exercises in the Autumn 2010 instance of the course, in which students were to perform the following tasks:

1. Implement a general semaphore using just one tuple (P and V operations are sufficient) (Exercise 9.1 [2]).
2. The students were given an incorrect concurrent sorting program that deadlocks; they were asked to produce a failure, explain what caused the failure and to fix the problem.
3. Implement a bounded buffer using a tuple space (Exercise 9.4 [2]).
4. The students were given a concurrent matrix multiplication program that may generate incorrect results; they were asked to produce a failure, explain what caused the failure and fix the problem.

In order to compare students using Atropos and not using Atropos, the students were divided into two groups: A and B. However, since only 21 students participated in the sessions (in part, due to students dropping out of the course; 79 students registered for the course, but only 37 students took the exam at the end of the course), the quantitative comparison was left out of the study.

The first and third tasks were intended to evaluate what the students know. The second and fourth tasks were intended to compare how the students work with and without visualisation. In task 2, group A used Atropos and group B did not; this was reversed in task 4.

Also, even though the sessions were extended from the original two hours to allow students to complete their work (in one case almost up to four hours), none of the pairs completed task 4. Hence, we focus in this study on a qualitative analysis of task 2, which is the task most relevant to our research questions.

3.2 Data Collection

All the students were recorded throughout the entire session. Their conversations and the contents of the screen of the computer they used were recorded using a microphone and video screen capture software. Two selected pairs in each session were recorded using a video camera to provide additional information on what they are doing and to help determine how the use of screen capture instead of a video camera affects the accuracy of the analysis. These pairs were essentially self-selected as several pairs of students did not agree to the additional recording. The students were required to work in pairs which the students formed themselves. This was done in order to encourage them to verbalise their thoughts and to discuss their plans and approaches to the tasks.

Ten pairs of students and one lone student participated in the sessions. The lone student and two of the pairs have been left out due to lack of usable recordings of conversations to analyse. Four of the remaining pairs of students were in group A and four in group B. The following analysis focuses on the qualitative aspects and on evaluating the visualisation. Hence, we have analysed the second task as performed by the three pairs in group A who worked on task 2 during the sessions: Charles and Ada (46 minutes), Peter and John (45 minutes) and Brian and Dennis (51–53 minutes; these students worked independently on different tasks at some points). They all spoke Finnish. The students' names have been changed.

3.3 Analysis

The *operation foci* are what the students are doing in terms of concrete activities that are part of constructing a solution to their problem. Operation foci are determined by the subgoal students are trying to achieve at a given time.

The first version of the operation foci was created before the data were collected, based on the operation foci described by Yehezkel et al. [20] and the activities of Kiesmüller [11]. During the analysis, the categories were refined (by adding categories to cover activities not in any other category and by removing empty categories) to form the categories shown in Table 1. The foci are identified by numbers; in the case of subfoci, a letter is appended.

The descriptions of operation foci form a large part of the answer to RQs 3, 4 and 5. The operation foci related to Atropos highlight issues that answer RQs 1 and 2.

Based on the work of Yehezkel et al. [20], we also examined the students' *conversation foci*; the subjects of the students' conversation at a specific time. Since conversation foci and operation foci overlap extensively and since the operation foci are more relevant to answering the research questions, we have left the conversation foci out of this paper. We initially also analysed the transcripts for transactive discourse, as in the study by Murphy et al. [18]. We found too few instances of transactive discourse for a meaningful analysis.

3.3.1 *Performing the Analysis*

In preparation for analysis, the usable parts of the recordings were transcribed and translated into English. Actions performed on the computers by the students and their results were added to the transcript to aid in the analysis.

Initially, the analysis was done by attempting to place each statement in one category for each of the above classifications, using a spreadsheet with the transcript split into rows corresponding to statements. This approach proved to be cumbersome and error-prone by emphasising individual statements outside their context and by providing little support for effectively expressing structure within categories. Hence, a switch was made to expressing the transcript in a form similar to a script with quotations being formed from contiguous stretches of text and linked to codes that were then organised into the categories presented below. This analysis was done by the first author using ATLAS.ti.

Based on the data, the categories changed somewhat during the analysis. In particular, foci that do not fit into the above categories were used to refine the definitions of the categories and to add additional ones. Hence, the operation foci and the coding of one group of students were checked by the second author and adjusted until a consensus was reached.

4. RESULTS

Table 1: Operation foci

1 Understand Program Code Any activity with the intent of understanding program code in static terms, such as reading the source code and looking at how parts interrelate **1A Understand the Program to Debug** Understanding the program that the students should debug (in static terms) **1B Understand Other Available Code** Understanding other code involved in the task (e.g. library code, concurrency constructs available to them) and how to make use of it
2 Add Debug Code
3 Modify Code **3A Fix Bugs**
4 Formulate Hypotheses Formulate and discuss hypotheses about what a buggy program does and how it differs from the intended behaviour
5 Run the Program
6 Observe Program Behaviour Observe the program behaviour e.g. by reading console output or log files or by looking at the visualisation **6A Explore a Trace** Decide what to look at next in an execution trace (when using Atropos), including asking questions about what to look at next and trying out different commands in Atropos to find something useful **6B Interpret Observations** Discussion of representations of program behaviour and what they mean in terms of the program's execution **6C Unexpected Atropos Behaviour** Discussion of how Atropos does not behave as expected by the students and working to get it to act as expected
7 Determine Correctness Check or prove that a program works correctly **7A Compare Desired and Observed Behaviour** **7B Create Test Cases** Plan and set up test cases and executions **7C Analyse a Class of Scenarios Statically** Discuss and examine how the code reacts to a type of situation
8 Determine Goals Any activity related to understanding the desired behaviour of the program
9 Understand Atropos All activities with the goal of understanding Atropos, not directly trying to achieve anything task-related (e.g. reading Atropos's manual)
10 Present Solution to Assistant Convincing the assistant that the task has been successfully completed
11 Prepare Prepare for other operation focus **11A Create a Trace** Work on and discuss how to create a trace, assuming the test case has already been decided **11B Prepare Code** Prepare code or software for another operation focus **11C Set Up and Start Atropos** Get Atropos running, up to and including loading a trace

In this section, we present the results of our analysis in the form of a justification of the categories of the operation foci, using quotes and observations and in the form of diagrams of operation foci over time for groups of students (see Figure 1). In the diagrams, the Y axis shows time from the start of the recording (in hours and minutes). The X axis is labelled with the number of the operation focus (as shown in Table 1).

Sections of student activity for which an operation focus has been found are marked as vertical lines with start and end markers. To aid in understanding the diagrams, the students' activities have been divided into parts marked with red boxes labelled with a description of the activities.

4.1 Understand Program Code

Charles, Ada, Peter and John tried to **Understand the Program to Debug (1A)** before trying to **Observe Program Behaviour (6)** (see Figures 1a and 1b). Many cases of this involved one of the students talking at length about how he understood the program. In others, one student asked the other, for example (Figure 1b, 00:36):

> **John:** Do they all have that? Do those threads have a different id that they get, or?
> **Peter:** Yeah. The threads get an id number there when they're created.
> Peter points to `ConcurrentSelectionSort` constructor.
> **John:** Where's it given here?
> **Peter:** It's here on the second screen when you create those Workers: `new Worker(i).start();`

Students may in some cases first **Observe Program Behaviour (6)** and then return to trying to **Understand the Program to Debug (1A)** when they realise that there is part of the program they do not understand. For example, Brian and Dennis switched between exploring the execution of the program and the source code, initially with Brian exploring the code and analysing it statically and Dennis examining an execution in Atropos. An observation about what happened in the program led Brian several times to look for an explanation in the source code. This also involved trying to **Understand Other Available Code (1B)**, as in e.g. (Figure 1c, 01:12):

> Dennis shows the last operation in one of the deadlocked threads.
> **Dennis:** Yep, `readRemove`.
> **Brian:** Yep, it's here.
> **Dennis:** This is the space, apparently.
> **Brian:** This is the space. Here it waits; where's the `notify`?
> Brian scrolls through `Space.java`. Dennis shows the last operation in the main thread.
> **Brian:** When it posts, it notifies.

This strategy was somewhat problematic, as evidenced by Brian's comment (Figure 1c, 01:31): "*This algorithm is sort of too hard. If only I could figure out what it's supposed to do, I'd understand where the problem is.*"

In one case, an off-hand remark from a teaching assistant encouraged Peter and John to discuss the `Thread` class from the Java standard library, clarifying the difference between creating a thread and creating a `Thread` for John (Figure 1b, 01:04):

> **John:** What does `start` [in class `Thread`] do?
> **Peter:** It really creates a new thread and runs its `run` function in the new thread.
> **John:** OK, and this just... I mean it creates a thread, but not until the previous one was run... right?
> **Peter:** No, it doesn't. Actually, the `start` method creates the thread.
> **John:** But it says `new Worker` [a subclass of `Thread`] here.
> **Peter:** Yeah.
> **John:** It does create an object, though?
> **Peter:** It creates an object, but it's still not its own thread. `start` finally creates it.
> **John:** OK.

4.2 Add Debug Code

While one would expect students to **Add Debug Code (2)** to generate failing executions for debugging, we found no evidence of this, probably because the example data included in the incorrect program was enough to get it to fail.

However, Brian decided to examine the program using Eclipse while Dennis used Atropos. Rather than use the debugger to examine intermediate states, he added (and later removed) `print` statements to determine whether certain lines of code had been executed and what values they used. When doing this, it is natural to **Run the Program (5)** afterwards, which he did.

4.3 Modify Code

The task in this study was simple enough that the only times students had to **Modify Code (3)** were to **Fix Bugs (3A)**. The students had to do this (at least) once to complete the exercise, but they also discussed and, in one case, tried out other possible fixes (Figure 1c, 01:31).

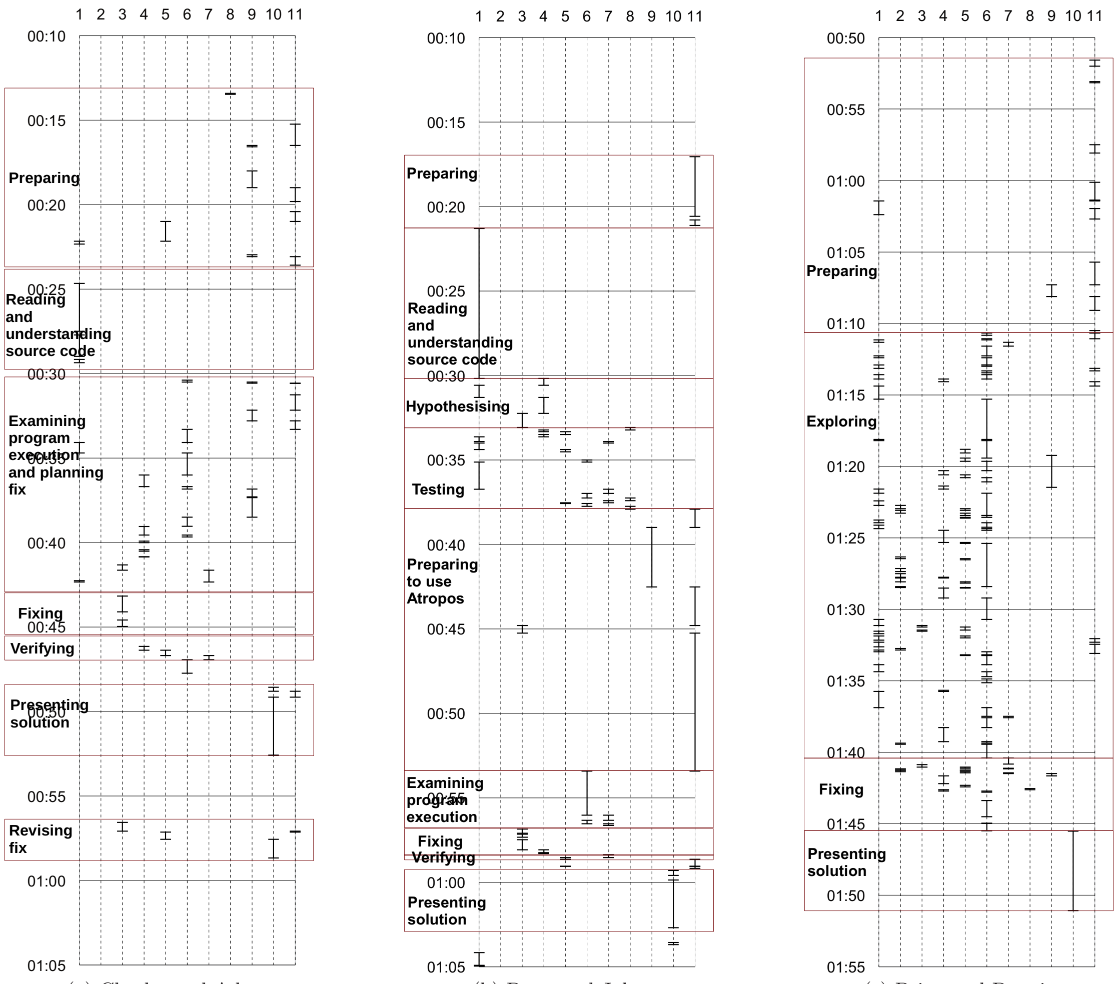
It is interesting but hardly surprising that students are adept at finding ways to quickly get a program to produce the right result without regard for whether their fix has undesirable side-effects. Consider the following exchange illustrating how two students **Fix Bugs (3A)** (Figure 1b, 00:32):

> **John:** Could that be fixed just by writing `synchronized` here?
> John points to `Worker.run` declaration.
> **Peter:** Sort of, but this can be fixed with just the tuple space and without us adding any new synchronisation, so just one thread at a time can be here, so we add a tuple that says "now we're counting something". Then, when we get here, the thread tries to take the tuple away. Then when it's done calculating, it puts the tuple back.

First, John suggests only running one worker at a time by making them mutually exclusive using Java's built-in locking mechanisms. Then, Peter counters John's suggestion with an alternative: use a tuple to implement a lock, avoiding the need to use any other inter-thread communication than the tuple space. This effectively converts the concurrent sorting algorithm into a sequential one, albeit one that processes elements in a random order.

4.4 Formulate Hypotheses

In a debugging task, students will hopefully **Formulate Hypotheses (4)** about what the program does and how it differs from the intended behaviour. For example, Ada observes that several threads are waiting for a tuple and then hypothesises (correctly, as it turns out) that the deadlocked threads are waiting for a tuple to be put in for them to



(a) Charles and Ada (b) Peter and John (c) Brian and Dennis

Figure 1: Operation foci over time for each group

remove (Figure 1a, 00:36): "*They're never put in the space, those... values. Because now it's just waiting for it, or it's removed the note, and now it's waiting for a message that it would remove.*"

Brian constructs several hypotheses about what is going on in the program based on what he sees in the program code and execution, e.g. (Figure 1c, 01:20): "*So it waits for there to be a... So the last tuple is not removed, so the sorting goes in the other direction, so that is apparently not called.*"

4.5 Run the Program

One would think that it is commonly accepted that, having written or modified a program, one should **Run the Program (5)** to make sure it works. In the debugging task, the students executed their modified programs before deeming them complete and going on to **Present Solution to Assistant (10)** (see Figure 1). For example, Charles and Ada run their fixed program once (Figure 1a, 00:46–00:47):

Recompiles `ConcurrentSelectionSort`, runs, gets correct result immediately.
Ada: Well, the sequence is there. Pretty good. How many times do we need to see if it still works?
Restarts Atropos, reloads trace, fails cryptically due to mismatch between trace and the modified class file.
Charles: Right, just... [inaudible] Ugh.
Dismisses error message.
Ada: Put us in the queue.
Adds self to assistant waiting queue.

At the other extreme, Brian repeatedly runs both the original program and his modified versions to examine their behaviour and verify that the bug has been fixed.

4.6 Observe Program Behaviour

Since one goal of Atropos is to help students **Observe Program Behaviour (6)**, this category is of particular interest in evaluating and developing Atropos.

After running the buggy concurrent sort program and noting it failed to produce any output, John noted (Figure 1b, 00:37): "*All right, it blew up at once. It deadlocked.*"

Similarly, Peter determines from Atropos's list of threads and their final operations that the threads are blocked, and then they use static analysis to determine why they are blocked (Figure 1b, 00:56):

> **Peter:** They're all blocked here in `wait`. Except this one.
> Peter points at one of the threads in the list.
> **John:** When does it enter `wait`?
> Peter switches to source code in Eclipse.
> **Peter:** So if you take `removenote` and it can't be found there.
> **John:** OK, right.
> **Peter:** Yeah, this is totally clear.

After exploring the branches that lead to the `wait` calls in which the program deadlocked using Atropos, Charles and Ada find that `wait` was called by `Space.readRemove` because `Space.searchNote` found no matching note in the space. Ada then asks (Figure 1a, 00:39):

> **Ada:** But where, where, isn't `searchNote` used when we try... or like, what's the method of the previous one? Is that one? If you click those?
> **Charles:** No, but `p` is used to try... yeah, a parameter.

Charles responds to the question by looking at the source code for `Space`; his response seems confused since `p` in this context is a field of the `Note` (i.e. the tuple) that is being checked for a match, not a parameter.

Dennis, after having tried out grouping operations together in Atropos in an effort to avoid getting bogged down in the `for` loops in `searchNote` that repeatedly scan through the tuple spaces, finds himself viewing a list of data sources for the `searchNote` method execution that fills much of the screen (Figure 1c, 01:40). He notes: "*This is what it looks like. I have no way of figuring out what to do now. A horrible amount of data is what I get.*" Shortly after (Figure 1c, 01:41), he opines that "*Someone should tell us how to really use Atropos to figure this out. I've been clicking around and I feel kind of silly.*" The other pairs run into similar problems: their attempts at exploring the dependence graph merely lead them in circles around the `searchNote` method.

4.7 Determine Correctness

In order to get meaningful program executions to **Compare Desired and Observed Behaviour (7A)**, the students must **Create Test Cases (7B)**. In practice, this manifested mostly as running the program using the example data provided to the students. For example, Peter, working on the hypothesis that the concurrent selection sort program can deadlock, suggests after a successful execution of the program (Figure 1b, 00:36): "*Try running once more. I mean, it worked now. It didn't hang.*"

One important part of determining correctness is to **Compare Desired and Observed Behaviour (7A)**. This has two important roles in this task: determining how the incorrect program misbehaves and ensuring that the bug has been fixed. Peter and John first used this approach to test the broken concurrent sort (Figure 1b, 00:37):

> **John:** Should that output have a certain form or order; is there a right or wrong order?
> **Peter:** This should always be in alphabetical order, but...
> **John:** Yeah, it is.

After examining the thread list as described in Subsection 4.6, Peter starts to **Analyse a Class of Scenarios Statically (7C)** and confirms his earlier hypothesis by reasoning from the source code about what must have happened for the execution to have reached the situation shown in Atropos (Figure 1b, 00:56): "*So if you take `removenote` and it can't be found there.*" He then proceeds to convert the sort into a sequential sort to eliminate the deadlock, based on the suggestion described in Subsection 4.3.

In contrast, Charles and Ada only use the example program provided to them to test their solution to a concurrency bug; their only question regarding how to test this, raised by Ada, is how many times to run the example program on the corrected code (Figure 1a, 00:46): "*Well, the sequence is there. Pretty good. How many times do we need to see if it still works?*"

4.8 Determine Goals

Students obviously **Determine Goals (8)** when they read the description of the task they are to perform and try to understand what it entails. Students also seem to decide to use Atropos when they **Determine Goals (8)**. For example, John realises: "*Wait, we were supposed to use Atropos now.*" after they have essentially completed the task (Figure 1b, 00:33). After verifying their solution without Atropos, they again bring Atropos up (Figure 1b, 00:37):

> **Peter:** Should we use Atropos?
> **John:** Well, for the sake of appearances...

4.9 Understand Atropos

Trying to **Understand Atropos (9)** most clearly manifests itself as reading its manual. Sometimes students read relevant parts of the manual out loud, such as when Peter reads "*Generating an execution trace: You need to compile with full debugging information...*" (Figure 1b, 00:42).

4.10 Present Solution to Assistant

When students **Present Solution to Assistant (10)**, they typically explain the code they have written or modified. Some students also provided feedback on Atropos, e.g. Dennis who described Atropos as "*not cooperative at all*", but added that he "*did get something out of Atropos in the previous exercise [round], but apparently, there should be some sort of mini-lecture about it*" (Figure 1c, 01:45).

4.11 Prepare

We want to minimise the time students need to **Prepare (11)**, since it does not contribute to learning. This includes downloading or finding code and importing it into development tools (including Atropos) as well as setting up these development tools. Peter and John spend some time (Figure 1b, 00:19–00:20) trying to **Prepare Code (11B)** by importing it into their Eclipse project for the second programming assignment and modifying the package structure.

Atropos should be designed to minimise the time needed to **Create a Trace (11A)** and **Set Up and Start Atropos (11C)**. Peter and John experienced problems running Atropos as a side effect of putting the concurrent selection sort program in a pre-existing project containing class files that reference external libraries, which Atropos cannot find. Charles and Ada caused Atropos's replay to fail by changing the class files it is using to replay the execution.

5. DISCUSSION

5.1 What Students Try to Do with Atropos

One approach to answering RQ 4 is to look at why students use Atropos; how it fits into their process of solving a debugging task.

Like Isohanni and Knobelsdorf [9], we found that some students do not use the provided visualisation tool even when instructed to do so or only use it in a limited fashion. As shown in Subsection 4.8, students seem to use Atropos because they are told to do so, not because they see a need for it. We found that some of our students did not even start Atropos until after they felt they had identified the bug. When one takes into account that they are clearly trying to **Understand Atropos (9)** for much of the beginning of the task, this suggests that most of the students had never used Atropos before, which would rule out problems with Atropos itself. This is strange, since Atropos was used in one of the tasks in the previous round of exercises. This suggests that Atropos would need to be better integrated with the rest of the course to convince students to use it. Indeed, one of the students explicitly suggested that the course should include at least a mini-lecture on how to apply Atropos effectively. The experiences of Isohanni and Knobelsdorf [9] suggest that this may not be enough, but their experiences may be due to the fact that VIP was not directly connected to debugging strategies. Giving students a visualisation tool that supports a new way of working (for example, a debugging strategy) is probably not sufficient to teach them this way of working unless they are given guidance on how to use it or the visualisation tool itself provides guidance on how it can be used. We return to the question of how the visualisation can guide students in Subsection 5.4.

5.2 Identifying Incorrect Behaviour

Once the students believe they have fixed a bug, they will hopefully **Determine Correctness (7)** of the program. As noted by Ben-David Kolikant and Ben-Ari [3, 4], students may not agree with the professional definition of correctness, which (rephrased in the terminology used here) is that the program is efficient, legible, documented, modular and, most importantly, always produces the right output no matter what input it is given and what interleavings occur. Some students' behaviour seems to be more consistent with seeing a correct program as one a teaching assistant will accept as correct or simply not, as suggested by Ben-David Kolikant and Ben-Ari [4], considering the possibility that they may have made an error. For example, at the end of the task, Charles and Ada make a change to the program, run it and then decide to present their change to the assistant without discussing whether it is correct at all. In terms of the purposes of programming assignments described by Lönnberg et al. [17], this is seeing the programming task in the framework of the university's requirements ("Assignment"). This is reflected in their testing style and in terms of the testing approaches described in the same paper, this means an additional category must be added below "Unplanned": "Testing not required". It is likely that lack of experience with concurrent programming accounts for this behaviour.

Even the students who did test their solution did not verify that their corrected program correctly handled the type of interleaving that caused the buggy program to fail. They could have done this either by running the program and examining the interleavings to determine whether they would have triggered the bug (which could easily be done with Atropos) or controlling the execution order of the program (which, in this case, could be done using breakpoints in a debugger). However, Brian did run the fixed program several times to confirm it did not deadlock where the incorrect program did.

Another possible contributing factor is that the students are used to relying on external support such as automated assessment systems for their testing. If there is no consequence to not testing, students do not bother to test. Students can be encouraged to test their programs by making the thoroughness of their testing an evaluation criterion. [5]

In a debugging task, they will, as demonstrated in Subsection 4.3, attempt to work around or fix a bug without confirming their hypothesis of what the bug is through examination of program execution. This also renders Atropos (and, indeed, most debugging and visualisation tools) somewhat irrelevant to their needs. Working around a bug can be prevented by specifically requiring that the corrected program has similar time and space requirements, which would preclude adding additional copies of data or making the concurrent execution sequential. It would also be useful to avoid confusion as to how the defective program is supposed to work by clearly specifying the expected behaviour.

It is hardly surprising that students have simplistic ways to **Determine Correctness (7)** in these tasks compared to, for example, those described in [17], which involved programming assignments in which students were told to explain how they had ensured the correctness of their solution, e.g. by testing. Another notable difference in the setting is that the tasks our students were working on were intended to be roughly two hours of work in total for them, while each programming assignment in [17] was supposed to be 20 hours. Also, the availability of almost instant feedback from teaching assistants and lack of consequences of submitting an incorrect solution are likely to have contributed to the general lack of interest in testing. One response to RQ 2 is thus: in order to encourage students to make use of testing and debugging tools, the effort needed to get started must (seem to) be less than other options, such as asking an assistant to look at the program.

5.3 Debugging Process

The students' debugging processes, which are relevant to answering RQ 5, are shown in Figure 1. The large-scale structure of their processes appears to be sound. We would expect, that after initially having to **Understand Atro-**

pos (9), **Determine Goals (8)** and **Prepare (11)**, that the students would make sure they **Understand Program Code (1)** and then **Run the Program (5)** in order to **Determine Correctness (7)**. Once the program has been found to fail, the resulting execution would then be used to track down the bug; a process in which the students would **Formulate Hypotheses (4)** based on what they learn when they **Observe Program Behaviour (6)** and **Determine Correctness (7)**. Once they are satisfied they have confirmed their hypothesis of what the defect is, they would **Fix Bugs (3A)** and **Run the Program (5)** and **Observe Program Behaviour (6)** in order to **Determine Correctness (7)** of their corrected program. What the students have done mostly fits this pattern, which would suggest that Atropos (or any debugging or visualisation tool that could be used in a similar fashion) would fit well into their large-scale approach to debugging. In line with the results of Ahmadzadeh et al. [1], Brian and Dennis did not take the time to **Understand Program Code (1)** first (see Figure 1) and hence had difficulties debugging.

5.4 Successful and Unsuccessful Atropos Use

As a basis for comparison, we will first present how we expected students to apply Atropos in the debugging task. The first step is to determine that the program has deadlocked by having all threads waiting for another thread to insert a tuple in the space. This can easily be deduced by checking where the threads' execution blocked. The second step is to identify which tuples are being waited for and in which place in the main program. This can be done by determining from where the blocked operations were called. It should be noted that examining the call stack at the time the program deadlocked is also sufficient to collect the information required so far; this can be done using e.g. Eclipse's debugger like Brian did. The third step is to identify that the reason why these tuples cannot be found is that they were removed from the space earlier and where this happens. Finally, the problem can be fixed by returning the missing tuples to the space before getting others. This fix can then be verified by running the program and confirming that it still runs correctly even when threads are interleaved in ways that caused the bug to manifest in the original version of the program.

In order to answer RQs 3, 4 and 5, and hence RQs 1 and 2, we must look at how students used and failed to use Atropos. Students were able to use Atropos to extract information to help clear up some misunderstandings of, for example, what threads exist in a running program, as shown in Subsection 4.6. However, this information is derived from the list of how threads ended, not from the dependence graph itself. This happened even though the students 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. [6]). Peter and John spent 11 minutes (Charles and Ada 5 minutes) trying to **Understand Program Code (1)**, but both pairs only spent 3 minutes trying to **Observe Program Behaviour (6)**.

In Subsection 4.6, we see that several students were effectively prevented from finding the relevant data in Atropos by difficulties caused by the visualisation displaying implementation details of the tuple space as a consequence of following data and control dependencies of the `wait` operations in which the program deadlocked. Dennis managed to group the operations in the tuple space method executions together, but the sheer amount of data dependencies of a method execution put a stop to his progress. 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. Ada's suggestion in Subsection 4.6 suggests that a more intuitive operation would be to show the operation that invoked the method execution to which a selected operation belongs; this has since been added to Atropos. In the latter case, grouping together multiple reads of the same values in the context menu could have made it far easier to navigate. In both cases, one can argue that the problem is that the students are being shown what is happening in the tuple space even though they are interested in how it is used.

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. To help users get started, Atropos could explicitly identify symptoms, such as deadlocks or incorrect behaviour, and recommend these as starting points for backward debugging. The backward debugging process could be supported by providing the option to mark operations as correct or incorrect behaviour, making it easier to see what possible causes of an incorrect operation's behaviour have been explored.

Similarly, to help users examine executions at an appropriate level of abstraction, it would be useful if users could select, before starting to explore a trace, which classes' execution the user wants to examine. A third option would be to start exploring a thread from not just the last operation, but all the operations that would be shown in a stack trace.

Another possible approach is to complement the DDG with overviews of the operations performed by each thread (for details, see [15]). This would enable users to skip to interesting parts of the program execution rather than find a route back along the DDG from a failure. This would also provide an overview of program execution that some students tried to achieve by repeatedly requesting the previous line until they had a list of what a thread had done.

5.5 Summary

It is unclear to students how Atropos fits in the debugging process and how to effectively apply it when debugging. In part, this is due to unfamiliarity with a new tool, which can be mitigated by giving students explicit guidance on debugging strategies using this tool as part of the lectures.

Some students also avoided checking their hypotheses or code and instead chose to ask the assistant to evaluate it. This is probably because assistants were available to the students during the sessions. This makes software to help in evaluating these hypotheses, such as Atropos, irrelevant.

Navigating a DDG using Atropos turned out to be the most problematic aspect of using it. The students failed to reach a useful level of abstraction; instead, they spent much of their time examining the tuple space implementation's behaviour. It is clear that the mechanisms currently available to students to deal with complexity of the DDG do not fit their expectations of how it should be done. In order for students to be able to deal with programs of this level of complexity in Atropos (or any other tool) effectively, it must provide an obvious and easy way to elide details of the execution of methods that are assumed to be correct or a way for students to get an overview of an execution.

6. CONCLUSIONS

Our results support our theory [17] that students primarily see programming assignments as a task to be completed to get a grade and that they will avoid any tasks they see as extraneous such as testing it or examining its behaviour in detail. Hence, if a teacher wants students to test their programs or examine the behaviour of a program in detail, this must be made an explicit part of the task with clearly defined requirements. It is worrisome that students even at this advanced level do not take testing seriously, but this may be a consequence of having feedback from teaching assistants immediately available.

As Isohanni and Knobelsdorf [9] noted, students seem to avoid using visualisation even when instructed to do so, preferring to try to reason about the program statically. We believe one can encourage students to use visualisation by integrating examples of the effective use of the visualisation into the teaching.

While it seems that the visual representation used by Atropos can be interpreted by students without undue effort, navigating through a program is still difficult. While the dependence graph visualisation would seem to be useful for understanding short causal chains at the level of proficiency with the visualisation our students showed, it is necessary to develop ways to support students in navigating the graph before they can make full use of Atropos.

7. REFERENCES

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press.

[2] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education, second edition, 2006.

[3] Y. Ben-David Kolikant. Students' alternative standards for correctness. In *The Proceedings of the First International Computing Education Research Workshop*, pages 37–46, 2005.

[4] Y. Ben-David Kolikant and M. Ben-Ari. Fertile zones of cultural encounter in computer science education. *Journal of the Learning Sciences*, 17(1):1–32, Jan. 2008.

[5] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3):1–24, 2003.

[6] 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.

[7] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[8] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.

[9] E. Isohanni and M. Knobelsdorf. Behind the curtain: Students' use of VIP after class. In *ICER '10: Proceedings of the Sixth International Workshop on Computing Education Research*, pages 87–95, Aarhus, Denmark, Aug. 2010. ACM.

[10] R. Johnson and A. J. Onwuegbuzie. Mixed methods research: A research paradigm whose time has come. *Educational Researcher*, 33(7):14–26, 2004.

[11] U. Kiesmüller. Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education. *Trans. Comput. Educ.*, 9(3):1–26, 2009.

[12] 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.

[13] 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.

[14] 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*, pages 15–25, Toronto, Canada, July 2011. ACM.

[15] J. Lönnberg, M. Ben-Ari, and L. Malmi. Visualising concurrent programs with dynamic dependence graphs. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*, To appear.

[16] 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.

[17] 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.

[18] L. Murphy, S. Fitzgerald, B. Hanks, and R. McCauley. Pair debugging: A transactive discourse analysis. In *ICER'10: Proceedings of the International Computing Education Research Workshop*, pages 51–58, Aarhus, Denmark, Aug. 2010. ACM.

[19] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[20] C. Yehezkel, M. Ben-Ari, and T. Dreyfus. The contribution of visualization to learning computer architecture. *Computer Science Education*, 17(2):117 – 127, June 2007.