Concurrency, objects and visualisation

Chris Exton
School of Network Computing
Monash University
exton@monash.edu.au

Michael Kölling
School of Network Computing
Monash University
mik@monash.edu.au

Abstract

Object-oriented programming and concurrency are increasingly popular in computing education. Both are difficult topics in themselves, and the combination of both introduces subtle interactions that are not easily understood. We propose the development of a visualisation tool to illustrate both object-orientation as well as concurrency issues.

Designing such a tool is a challenging task. It has been shown that visualisation tools are not always as effective as their authors had hoped, and the issues to be illustrated by our potential tools are not yet well defined.

In this paper, we investigate both the visualisation aspect and the functionality that such a tool may have and we develop some guidelines for the design of a concurrent object visualisation tool.

1 Introduction

Concurrent programming has become an important area in the development of software for application systems. For many computer science degrees concurrent programming is becoming an essential part of the undergraduate curriculum [1]. This movement was supported by the Association for Computing Machinery (ACM) as part of its Curriculum 91 recommendations in which it advocated introducing distributed and parallel programming constructs into the undergraduate study curriculum.

Learning and teaching concurrent programming is difficult. Not only is the subject matter complex, but it often is difficult for students to assess when they have actually correctly solved a problem. Errors in concurrent programs are often hard to detect – getting the correct output in a test execution does not prove anything. Choi and Lewis [2] have found in a detailed study that 56 out of 180 student submissions contained errors, although virtually all of them produced correct output.

The comprehension and analysis steps of cognitive learning, as described by Bloom [3], are difficult for students in this area, since many of the mechanisms students have established for non-concurrent programming techniques do not migrate well to the concurrent equivalent.

The integration of concurrency and object-orientation would seem inevitable given the current popularity of object-orientation and the increasing necessity for concurrent computation. To many, this combination seems quite natural. However, the construction of such a system is not the simple matter that one would initially believe. The claimed benefits of object-orientation are many and include reuse, quality, emphasis on modelling the real world. These advantages are provided by the use of inheritance, encapsulation, abstraction and polymorphism. By incorporating concurrency with object-orientation, one would hope to maintain the benefits of object-orientation while gaining the performance and increased ability that is associated with concurrent and parallel software.

However, there exist subtle and yet important semantic conflicts between concurrency and object-orientation. While it is not difficult to produce a language such as Java that supports the constructs associated with concurrency, and the functionality associated with object-orientation, the designer of a concurrent object-oriented program must also consider the ramifications of their design. It is possible that a poorly designed concurrent object-oriented program would cancel the very benefits that the designer had hoped to attain.

Dijkstra, in 1968 in his famous letter to the ACM, stated that "our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed" [4]. One possible conclusion to draw from this is that software tools that aid understanding of concurrent processes may be beneficial to teaching and learning about concurrency. It is, however, not at all clear what such a software tool should look like.

This is especially true for object-oriented concurrent systems. Concurrency visualisation systems have, in the past, focused on non-object-oriented systems. Concurrency together with objects introduces new behavioural aspects that should be included into a visualisation tool.

In this paper, we will investigate requirements and propose solutions for the design of an educational software tool that integrates concurrency and object-oriented programming.

This paper is organised as follows: In section 2, we investigate some issues specific to concurrency in Java. Section 3 comments on existing software tools currently

available. In section 4, we investigate problem issues distinct to concurrent programming. Following that, we discuss a possible tool design to support understanding of these issues.

2 Concurrency and Java

Java has, over the past few years, been adopted by many universities as a teaching language. The acceptance of Java in universities and industry has been so rapid that within a very short time it has become one of the most used languages for teaching of programming practices. This creates some challenges and opportunities.

Java is concurrent. This is true in two respects. Firstly, Java supports threads and synchronisation constructs as part of the standard language. Secondly, the Java platform (the virtual machine) executes as a collection of concurrent threads.

The second point almost forces the teaching of concurrency into a curriculum taught with Java. Every Java application is concurrent. The garbage collector, for example, and screen painting for GUI applications are executed as threads running independently from the user's main thread. Lack of understanding of the details of this can lead to problems even in introductory courses. The garbage collector, for example, can influence experiments with runtime performance of algorithms and data structures. Not understanding the screen painter thread can lead to surprising bugs in the display of animated graphics or user interfaces.

The inclusion of concurrency constructs in the language, however, creates new opportunities. Currently, concurrency courses are taught in a wide variety of different languages and systems. Most programming languages used in universities do not include concurrency constructs in the language itself. As a result, a wide variety of concurrency libraries is used, some standard (such as pthreads), some home-made. This large variety of systems being used fragments the user base and makes it hard to provide good tools.

Having concurrency mechanisms in a widely accepted teaching language may lead to a much larger number of courses using the same tools, and thus may make it more attractive for tool developers to invest time in effort in developing good tools.

3 Software tools for education

Numerous software tools are available for use in education. A common characteristic of most of them, especially those dealing with concurrency, is the use of visualisation techniques.

Visualisation can come in many forms. Very different technologies can be presented under the heading of "visualisation". There is a basic idea, though, that is common to all of them: the assumption that making the inner workings of an abstract problem visible in some form can provide additional insight that aids understanding.

Visualisation tools can be separated into three distinct categories: algorithm visualisation, visualisation with interaction and execution environments with visualisation. These three have quite different characteristics, and we will briefly discuss each of them.

3.1 Algorithm visualisation

Algorithm visualisation tools, such as Balsa [5], XTANGO [6] or ANIMAL [7], typically provide a visual representation of the execution of a given algorithm. They are often hard coded (the visualised algorithm cannot be easily changed by the end user) and provide no or little user interaction.

Numerous studies exist trying to show the effects of using algorithm animation on student performance. The hope generally was that using algorithm animation systems would increase students' understanding of the subject matter. The results, however, are mixed, with the majority concluding that algorithm visualisation without interaction is not very effective. Stasko *et al*, for instance, demonstrate that a student group using an algorithm visualiser an addition to text based instruction did not perform much better than a control group studying with the text alone [8]. They found no statistically significant difference between the two groups. Jarc *et al* even report a negative effect. Students using an animation system spent more time on studying the material but performed worse in post-tests than a control group [9].

For our work, we can conclude from this that a passive visualisation system alone is not desirable as an aid to support understanding of concurrency.

3.2 Visualisation with interaction

Algorithm visualisation with interaction is an extension of the idea described above. In these systems, users are asked to make predictions or solve questions as part of using the visualisation. Stasko *et al*, in the same study referred to above [8], also conclude that animation systems should have a rewind/replay facility, since such a facility has the potential to increase the learning effect achieved through animations.

Some more recent algorithm animation systems, such as JHAVÉ [10], include functionality to engage students more actively in the visualisation. These interactions include users constructing their own input data, stopping the animation at various points and letting the user make predictions and quizzes and tests.

Some studies seem to show that this can increase the level of comprehension achieved by using the visualisation system [10, 11], while others indicate no beneficial effect of algorithm visualisation even with interaction facilities [9].

3.3 Execution with visualisation

Execution systems with visualisation follow a different approach to algorithm visualisers. They do not have an algorithm (or a set of algorithms) built in, but instead let users write algorithms in a normal fashion. They typically extended program execution environments with visualisation tools.

Systems of this kind are much more flexible than the tools described above. They allow users to visualise any code that can be written in a given system. An example of such a system for Java is Jeliot 2000 [12], a Java

interpreter that provides continuous visualisation of statement execution.

Execution visualisers again fall into different categories: those that require source code to be annotated to aid visualisation, and those that work on standard, unaltered program code.

Systems that can visualise original code are clearly the more flexible: they can be used to visualise programs taken from various sources, without the need for modification for the purpose of visualisation. Visualisers with annotations, on the other hand, can be easier to implement and may provide functionality that is difficult to achieve on some systems without annotation.

Another distinction is time of analysis: *live* systems display results while the application is executing, while *post-mortem* systems display execution information after the program has ended, typically from a log file written during execution.

3.4 A visualisation tool for teaching concurrency with objects

Algorithm animations are appropriate for algorithm and data structures courses, but too constrained for what we want to achieve in visualising concurrent objects. We want to allow students to investigate their own programs developed for arbitrary assignments. Thus, we will only consider execution visualisers from here on.

An execution visualiser for concurrent objects must provide information about both interesting object events and interesting concurrency events. These may include object creation, object destruction, method entry and object relationships on the object side and thread creation, thread status, synchronisation events and What exactly these "interesting events" are will have to be discussed in much more detail, and we will do this in the next section on concurrency issues.

It is, however, already apparent that the amount of data to be collected and visualised is potentially huge, and that some details may be "interesting" in some situations but not in others.

The issues involved are abstraction, emphasis, representation and navigation.

Abstraction can give us the ability to avoid the limitations of implementation languages and facilitate discussion on the pros and cons of various design tradeoffs. However the use of abstraction, as a tool of comprehension, is a dual edged sword. There is a spectrum here with no discrete divisions. For example, excluding too much detail may deprive the students of the ability to correlate what they are observing with actual events, while including too much detail may blind the students to the patterns of behaviour we wish them to observe.

Adjustable filters do provide a means to remove unnecessary detail. However, the student must first have gained the required level of knowledge and ability to distinguish between what information should be included and what can be excluded.

We may also want to present the same information with the same level of abstraction but with a different *emphasis*. For example we may wish to observe the effects of the inheritance anomaly [13], deadlock or dormancy. Each of these necessitates a different emphasis on the same data. Thus it should be possible to highlight the particular run-time phenomenon that we wish to consider by changing the emphasis.

The means by which we represent information is thus partially determined by the level of abstraction and emphasis. However, to allow for this, considerable thought must be given to *representation*. The use of traditional methods such as graphs and charts may be of some use whilst more artistic or attractive means of representation, such as animation or even virtual reality, may be used to increase understanding, enthusiasm and serve as a motivational factor.

Human-computer interaction issues play a pivotal role. The ability to navigate in a potentially very large and complicated visual environment is often one of the hardest problems to solve.

Navigation should allow the user to explore and interact with the visual environment and provide a means of reducing or increasing visual complexity by varying abstraction, representation and emphasis.

Before we go on to describe the design for a visualisation tool in more detail, we will discuss a set of common problem areas in working with concurrent objects. These problem areas give us the goals for our tool: we would hope that a well designed visualiser will allow to illustrate and discuss any of these issues.

4 Issues with object-oriented concurrency

Migrating themes, procedures and understandings from the sequential world to the parallel and distributed worlds of software construction has often been problematic. Concurrent programming introduces new types of errors and complications when compared to traditional sequential program development. Ben-Ari [14] states that "when teaching concurrent and distributed programming, it is extremely important that you demonstrate to students the strange behaviour that such programs can show". We have placed the interesting concurrency errors and behaviours that we feel a student must become cognisant of into three broad categories. These are liveness, safety, and a third which is directly related to the object-oriented paradigm.

4.1 Safety

The issue of safety is often difficult for many students to appreciate. In many cases a program may produce correct output every time it is tested as the order of interleaving produced by a thread scheduler on a given machine and environment is sufficiently similar to produce a correct result for each test. This fact alone, however, does not assure that the program is safe. Students often have the unpleasant experience of watching their tested program fail on its first execution on their tutors machine. This is solely because the operating system and environment used have resulted in a different interleaving of threads and uncovered a previously unknown safety error. Thread interleaving is a difficult concept for many students to grasp as it depends on factors outside their control. In a controlled execution environment it is possible to impede to execution of

selected threads and to highlight safety errors in the students code.

4.2 Liveness

Liveness failures are often much harder to identify than safety failure and lead to some mystifying observations for students. For example two components that are each live when used in other contexts may fail to be live when used together. Thus unit testing often fails to reveal such failures. Lee [15] lists four interrelated senses in which one or more threads can fail contention, dormancy, deadlock and premature termination. Contention or starvation occurs when runnable threads are not executed because other running threads are monopolising the processor. Dormancy is perhaps the most common and arises when a non-runnable thread fails to become runnable. In Java this most commonly occurs when a *wait()* was not balanced by a *notify()* thus the thread in effect remains blocked until the program terminates. This common error is often addressed by the unsuspecting student by instantiating a new thread object every time the same operation is required. This incorrect solution is often unobservable in terms of expected output as the program often produces the correct results but in an extremely inefficient and precarious manner. Deadlock is the result of two or more threads trying to acquire a resource that is already held by the opposing thread. Deadlock failures are notoriously difficult to locate as they are often moving targets. Thus a student may place a series of display statements to narrow down the problem area but the deadlock may occur at another location on the next execution of the program. Premature termination in a multi threaded program again often goes undetected as the other threads will often continue executing which serves to mask the error.

4.3 Concurrency and Object Orientation

On first glance, the combination of concurrency and inheritance seems quite natural and unproblematic. There exist, however, subtle and yet important semantic conflicts between the two. The inheritance of concurrency constructs can result in the need for non-trivial class redefinition [13]. This means that the programmer will quite often need to investigate the implementation details of various superclasses in order to incorporate the necessary redefinitions into a subclass.

Further problems can be caused by interactions of concurrency with exception handling. In a sequential program, the semantics of exception propagation and handling are clearly defined and simple: an exception will propagate by winding back the stack until a handler is found to catch the exception; if no handler is found, then the process is terminated. Although the implementation of exceptions may be well understood in a non-concurrent language environment, the understanding of exception handling behaviour is often not considered when dealing with a multi-threaded scenario.

In many parallel environments, when an exception is raised during the execution of a thread and the thread fails to handle the exception, the thread is abandoned without further effect. The process continues execution and any notification of the lost exception is left up to the programmer to implement explicitly, as is the case with C++, Ada 95 and Java.

The problems associated with the combination of concurrency and object orientation such as the inheritance anomaly and exception handling are profound and need to be explicitly addressed in addition to the more typical concerns of Safety and Liveness that are apparent in a non-object-oriented environment.

5 Tool design

In this part of the paper we want to discuss some design decisions for a software tool to aid teaching and learning of concurrent programming.

When designing a software system, two aspects are important to keep in mind at all times: the *purpose* of the system and the targeted *user group*. Many bad design decisions result from not having a clear picture of these two aspects.

Our purpose is educational, our target group are students. These two decisions will fundamentally influence all decisions to be made. The tools needed for experts and students differ profoundly: while experts need flexibility and detail, even if it means spending a long time learning how to use a particular tool, students rely more heavily on simplicity, allowing them to use a tool effectively within a comparatively short period of time.

In restricting the user group fairly narrowly (by not attempting to serve both students and experts with a single tool) we believe that we can serve that one group better (at the expense, possibly, of not providing adequate tools for other users). Our aim is to design a good tool for a specific group rather than a mediocre tool for everyone.

Visualisation in itself does not necessarily serve the purpose of clarifying abstract information. The assumption that information becomes clearer just by being presented graphically would be misguided. The key lies in the *interpretation* of the visible information. Only if the information encoded in the graph is easily and correctly interpreted does it serve a purpose. The commonly quoted saying of the picture saying a thousand words is too often taken as meaning "graphics are good" without much further qualification. But we have to ask ourselves: which thousand words is our picture really saying? And does it say the same thousand words to everyone?

Reading of graphical information must be learned just as well as reading text. The success of our system will depend in part on the question of how easily students can learn to read the visual output.

5.1 Functionality

From the issues discussed in previous sections of this paper we can now give an overview of desirable functionality of an educational concurrency visualisation system. Some of the functions are useful for analysing concurrent applications in general, others are related specifically to concurrency in object-oriented systems.

The system should:

- provide information about existing threads and their current status;

- provide a history of each thread;
- provide information about existing synchronisation objects (such as locks, semaphores or monitors) and their status;
- provide information about object events, such as object creation, destruction and method entry. The object events must be linked to the thread responsible for causing this event;
- have the ability to link events to source code;
- have the ability to rewind and replay an execution;
- provide filters to provide different levels of abstraction or emphasis;
- be able to illustrate the same data set using different filters;
- allow the application of filters at the time of inspection (retrospectively), rather than before an execution.

In addition to these, it will be crucial to illustrate and clarify the duality of threads and objects. Since threads are often represented as objects on an object-oriented library (but a fundamental difference between threads and objects remains) there is great potential for confusion.

Automatic problem detection tools, such as automatic deadlock or race detection may be a useful addition.

5.2 Platform

One of the problems with existing multi-threading visualisation or teaching systems is the lack of a widely accepted platform. Ideally, we want a widely used standard language with a standard multi-threading library. The drawbacks of using specific languages, systems or libraries have been mentioned above.

Java gives us a platform that provides a commonly accepted language together with standard multi-threading constructs. For that reason, we will choose Java as the user platform for our tool.

5.3 Architecture

One of the most fundamental design decisions for the tool architecture is the time of analysis: should it work on a live execution, or should the analysis be done post mortem?

From the desired functionality discussed above, it is obvious that we would need post-mortem analysis. This is needed to apply filters retrospectively, change levels of abstraction and emphasis and to provide rewind/replay capabilities.

Bedy *et al* [16], describing their own system, argue for live analysis. Their arguments against post-mortem analysis are that the data set collected during execution may be incomplete or corrupted if an execution does not end normally, and that a program must be explicitly instrumented for post-mortem analysis. Explicit instrumentation adds an extra level of complexity and a potential source of additional errors.

The first argument can be countered if we are successful in designing a system that accepts incomplete data sets. That does not seem to pose a real problem. The second argument does not hold true for Java. The fact that Java is executed on a virtual machine enables us to provide post-mortem analysis without instrumentation of the code. Instead of adapting the user code for the purpose of gathering data, we can adapt the virtual machine.

One commonly used technique to gather data for either live or post-mortem analysis is the use of custom-made synchronisation libraries. These libraries, in addition to providing the synchronisation functionality, perform the data collection or communication. Custom-made libraries are used either by writing source code specific to these libraries, or by automatic binary instrumentation. Both have their drawbacks. Requiring non-standard calls in the source prevents applications to be analysed that were not originally developed for this analysis tool. Binary instrumentation is usually platform dependent.

The solution to gather the information from the Java virtual machine overcomes these problems. We will be able to gather object and synchronisation data with standard user code using standard constructs and libraries. A user should be able to analyse any application taken from anywhere without the need to modify the application in any way.

5.4 Implementation

Above, we have argued to gather information for visualisation purposes from the Java virtual machine. Customising the machine for our tool would pose some problems: since the virtual machine implementation itself is dependent on the hardware platform, we would lose portability. Luckily, as of version 1.3, the Java Software Development Kit (SDK) includes debugging libraries (named JPDA [17]) that are powerful enough for our purposes. Using those libraries we should be able to gather all data required without using non-standard tools.

5.5 Interface

Designing the interface for the tool will be one of the major challenges of the project. The design of both the input (for navigation through a large data space) and output (visualising complex dependencies and behaviour) poses many open questions that we cannot answer at this stage.

One thing is clear, though: Since our target group are students, we cannot expect users to spend several months learning to drive and interpret the visualisation tool. For professional software engineers, it can be acceptable to spend a long getting accustomed to a tool, if that investment pays off over many years to come. For students, the challenge will be to provide an interface straight forward enough that it can be mastered within a relatively short period of time.

6 Conclusion

The object-oriented paradigm is well suited to concurrency, with its dual focus on modularity and encapsulation. The combination of concurrency and object-orientation has many potential benefits. The challenge for students, however, has been increased. To get students to truly realise these benefits is not a trivial task as aside from understanding the non-concurrent aspects of object-oriented programming the complex

nature of inter-object relationships during concurrent execution must also be understood.

Various software tools exist for concurrent program visualisation, each with a slightly different approach. However, very few of these tools have been designed with the intention of addressing the new issues associated with the integration of concurrency and object-orientation that aid student comprehension of how parallel execution relates to classes, objects, methods, and the semantics of normal non-parallel execution.

It is hoped that, with dutiful consideration of the issues presented in this paper, that we will be able to design a software visualisation tool that specifically addresses these issues. Work is already underway with an implementation of a visualisation tool called Elucidate. The major goal of the Elucidate project is to provide an expressive and flexible tool for teaching the structures and patterns of concurrent object-oriented behaviour. Although the first version is complete there is still much work to be done to dynamically represent all aspects of an executing multi-threaded program.

References

[1] M. B. Feldman and B. D. Bachus, *Concurrent Programming CAN be introduced into the Lower-Level Undergraduate Curriculum,* SIGCSE Bulletin, Vol. 29 No. 3, September 1997.

[2] S.-E. Choi and E. C. Lewis, *A Study of Common Pitfalls in Simple Multi-Threaded Programs*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 325-329, March 2000.

[3] B. S. E. Bloom, *Taxonomy of educational objectives : The classification of educational goals, Handbook 1: Cognitive domain,* David McKay Company, New York, 1959.

[4] E. W. Dijkstra, *GO TO statement considered harmful,* Communications of the ACM, Vol. 11 No. 3, 147-148, March 1968.

[5] M. H. Brown, *Exploring algorithms using Balsa II,* Computer, Vol. 21 No. 5, 14-36, May 1988.

[6] J. T. Stasko, *Animating algorithms with XTANGO,* SIGACT News, Vol. 23 No. 2, 67-71, Spring 1992.

[7] G. Rößling and B. Freisleben, *Experiences in Using Animations in Introductory Computer Science Lectures*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 134-138, March 2000.

[8] J. Stasko, A. Badre and C. Lewis, *Do Algorithm Animations Assist Learning? An Empirical Study and Analysis*, in Proceedings of the INTERCHI '93 Conference on Human Factors in Computer Systems, Amsterdam, 61-66, April 1993.

[9] D. J. Jarc, M. B. Feldman and R. S. Heller, *Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 377-381, March 2000.

[10] T. L. Naps, J. R. Eagan and L. L. Norton, *JHAVÉ -- An Environment to Actively Engage Students in Web-based Algorithm Visualizations*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 109-113, March 2000.

[11] M. D. Byrne, R. Catrambone and J. T. Stasko, *Do Algorithm Animations Aid Learning?,* Georgia Institute of Technology, Technical Report GIT-GVU-96-18, August 1996.

[12] J. Haajanen, *et al.*, *Animation of user algorithms on the Web*, in IEEE Symposium on Visual Languages, IEEE, 360-367, 1997.

[13] S. Matsuoka and A. Yonezawa, *Analysis of Inheritance Anomaly,* in *Object-Oriented Concurrent Programming Languages in Research Directions in Object-Based Concurrency*, MIT Press, 1993.

[14] M. Ben-Ari, *Distributed algorithms in Java*, in 2nd SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, Uppsala, Sweden, 62-64, 1997.

[15] D. Lee, *Concurrent Programming in Java: Design principles and patterns,* 2nd Edition, Addison-Wesley, 1999.

[16] M. Bedy, S. Carr, X. Huang and C.-K. Shene, *A Visualization System for Multithreaded Programming*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, 1-5, March 2000.

[17] Sun Microsystems, Inc, *Java™ Platform Debugger Architecture,* http://java.sun.com/products/jpda.