View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at University of West Bohemia

VISUALISING THE EXECUTION OF CONCURRENT OBJECT-ORIENTED PROGRAMS DYNAMICALLY USING UML

Hugo Leroux and Chris Exton

School of Network Computing
Monash University, Mc Mahons Road,
Frankston 3199
Australia

Email: hugo.leroux@monash.edu.au , exton@monash.edu.au

ABSTRACT

Understanding the intricacies behind concurrency within object-oriented programming languages has always been a challenge for undergraduate students. This is particularly true since both are complex issues in their own rights. Visualisation, when used adequately, can be of tremendous assistance in expediting comprehension of such complex issues. The aim of this paper is to discuss the potential of UML, as a medium within visualisation, to assist the comprehension of the execution of a concurrent object-oriented program. We thus investigate the qualities of UML as a language; discuss some of the issues associated with concurrency and Java and finally discuss the design of our visualisation tool.

1. INTRODUCTION

Concurrent object-oriented programming is hard. Object-oriented programming and design introduces non-trivial concepts such as inheritance, encapsulation, abstraction and polymorphism. Concurrency, on the other hand, brings forth new concepts such as deadlocks, safety and liveness issues and interleaving concepts during different executions of a program. Students often have difficulty in grasping the different concepts when concurrency and object-orientation are combined. This is particularly so because many of the basic mechanisms that are present in non-concurrent programming do not migrate well in a concurrent programming setting. Stepping through a source code, for example, is insufficient a means to understand the execution of a program, since the actual sequence of events is highly dependent upon the run-time scheduler.

When first exposed to a concurrent program, the majority of students seem to have problems understanding the interleaving of threads during execution. Synchronisation is another problematic issue. They very often find it challenging to determine which data item should be synchronised and which shouldn't, with the result that they either synchronise too much or too little. As described in a detailed study of student's submission of multi-threaded programs, Choi and Lewis [Choi00a] concluded that detecting an error in a concurrent program is hard. In their study, they found that 56 out of 180 submissions contained errors although most of them produced correct outputs. And, as they stated, many of these errors are often undetected during the marking process due to lack of time and the rarity of occurrence of the errors. One implication of the latter is that due to lack of feedback on the error, the student believes that his code is correct. This can have a exponential effect in later, more complex programs, where bugs can be much harder to uncover.

When Dijkstra stated in 1968, that *our intellectual powers are rather geared to master static relations and our powers to visualise processes evolving in time are relatively poorly developed* [Dijks68a], he was indirectly advocating for the integration of visualisation tools within programming environment, to aid program comprehension.

The main objective of this paper is to select a suitable technique for representing an executing concurrent program in an intuitive fashion that

highlights the concurrent and dynamic aspects of such program whilst retaining an appropriate level of abstraction. The Unified Modelling Language (UML) holds great promise in catering for our objective. It is a simple, well-established and widely available language. In this paper, we will investigate the effectiveness of UML as a medium for our visualisation tool and propose a design based upon it.

The paper is organised as follows. Section 2 looks at justifications of UML as a suitable medium for a visualisation tool. Section 3 discusses some of the issues relating to concurrency and Java. And finally, section 4 outlines the design of the visualisation tool.

2. JUSTIFICATION OF UML

Our first motivation in the choice of UML as a concept for dynamic visualisation of executing concurrent programs is popularity. UML is widely available and accepted as a mature language both by the software industry and educators. Furthermore, most of the students undertaking a concurrency subject should already be familiar with the basic concepts in UML, such as *Use Case diagrams*, *Sequence diagrams* and *Class diagrams*. Should it happen that the students are not familiar with it, we believe that this subset of UML is fairly easy to learn with a gentle learning curve.

Our second motivation is ease-of-use. Typically students have to learn and understand concurrency issues in a short period of time. As such, they will not be naturally inclined into learning any new concepts with regard to using a visualisation tool. Furthermore, they will most probably use UML during their analysis and design stages, so it makes sense to maintain the same interface for visual execution representation. This should, we believe, improve acceptance of the tool as a visual aid to program execution.

Our third motivation stems from the mixed results that various algorithm animation and visualisation tools in the literature have received. Hendrix *et al.* [Hendr00a] concluded that effective software visualisations can provide measurable benefits in program comprehension. Indeed their study showed that Control Structure Diagrams have a positive effect on program comprehensibility. On the other hand, Stasko *et al.* [Stask93a] and Byrne *et al.* [Byrne96a] do not share the same view. Stasko *et al.* believe that algorithm animation is not effective unless accompanied by comprehensive motivational instructions. They also advocate for an *active learning process* where the student actually builds an algorithm animation, as opposed to a *passive learning* approach of watching the animation.

Fourthly, we aim at providing a tool that is effective for use by students. In this respect, one of the authors of this paper defines four overlapping guidelines that need to be addressed: *abstraction, representation, emphasis* and *navigation* [Exton00a].

Abstraction refers to the inclusion or exclusion of details from the underlying program to expedite the student's comprehension of it. By changing the emphasis on the <u>same</u> data using the <u>same</u> level of abstraction, we are able to highlight some particular runtime scenarios in the execution. How we present the data, thus representation, directly impacts comprehension of the underlying program and interest in the tool. In essence, navigation is an amalgamation of the above three guidelines, where there is a trade-off between abstraction, emphasis and representation to form the equation.

We have identified two components of UML to serve the purpose of our visualisation tool. These are the *sequence* and *class state* diagrams. Before we commit to explaining how they both fit into our model, we should elaborate further on some problems relating to concurrency and Java, in particular. This serves two purposes; firstly, it allows us to identify what are the requirements of a visualisation tool and the issues to be addressed; secondly, it acts as a *blueprint* to evaluate the correctness and effectiveness of our tool.

3. ISSUES WITH CONCURRENCY AND JAVA

The migration from a sequential environment to a concurrent one is not without its fair share of problems. Choi and Lewis [Choi00a] have identified many synchronisation problems in their study. We analysed them and together with our own understanding of concurrency issues, we have established three broad categories of concurrent errors and behaviours. These are safety, liveness and concurrent object-oriented anomalies as discussed in [Exton00b].

3.1 Safety

A safety property asserts that nothing bad ever happens during execution. The simplest way to achieve safety is to avoid changing the state of the object during execution. However this seriously limits the functionality of a program. The objective is to have controlled state changes.

Safety is an issue that is not fully comprehended by students. Choi and Lewis [Choi00a] found that 23 out of 56 errors were data race errors. These errors occur as a result of the

absence or improper use of locking mechanisms on shared data. They uncovered eight different types of data race errors. Data race errors may go undetected on the student's computer but may appear at the first execution on a different platform. This is due to the run-time scheduler on the student's machine generating a different sequence of interleaving of threads as on the new platform.

3.2 Liveness

A liveness property states that something good eventually happens. Liveness problems are often harder to detect than safety problems. Lea [Lea97a] identifies four interrelated senses in which one or more threads can fail to be live: *contention, dormancy, deadlock* and *premature termination*.

Contention or starvation occur when a thread fails to run because other threads are monopolising the processor. Dormancy occurs when a non-runnable thread fails to become runnable. In Java, dormancy is often the result of a *wait*() not being followed by a *notify*(), thus blocking the running of the thread during the entire program execution. It has been noted [Exton00b] that the unsuspecting student bypasses the error by forking a new thread object each time that the operation is required, thus achieving the correct output in a very inefficient manner.

Deadlock occurs when two or more threads block each other while trying to acquire a resource that is locked by the opposing thread. Choi and Lewis [Choi00a] found 11 occurrences of deadlocks among the 56 incorrect submissions. Premature termination is often undetected as other threads continue executing, thus helping to hide the error.

3.3 Concurrency and Java

Inheritance is one of the most powerful constructs in Java. As stressed in [Briot98a], it is natural to use inheritance to specialise synchronisation specifications associated with a class of objects. However, the use of inheritance with synchronisation can lead to a redefinition of non-trivial classes. This phenomenon, named the *inheritance anomaly*, was introduced by Matsuoka and Yonezawa [Matsu93a].

Furthermore, as described in [Exton00b], the result of interacting concurrency with exception handling mechanisms can lead to complex problems for the programmer. While the semantics of exception handling are well defined and clearly understood for sequential programs, such is not the case in a concurrent environment. If an exception is raised during the execution of a thread in Java and the thread fails to handle the exception, then the thread is abandoned without further notice. And since it is a concurrent program, execution continues, helping to mask the error. Unless the student is very thorough in his/her design, this type of error can go undetected.

As we have outlined above, in addition to traditional safety and liveness issues, concurrency in Java brings forth new challenging concepts to the student. It is our belief that ignoring these issues in the design of a visualisation system is a serious lacking.

4. DESIGN OF OUR VISUALISATION TOOL

Exton and Kolling [Exton00b] mention two aspects that need to be addressed when designing a software system: the *purpose* of the system and its intended *user group*. Similar to them, our purpose is educational and our target group are students.

As mentioned in section 2, our main aims for the visualisation tools are simplicity, ease-of-use and effectiveness due to the fact that students have limited time to learn new concepts and will most probably lose faith in our tool, if we impose a steep learning curve on them to get familiar with the tool.

UML addresses the first two concepts, and it is our belief that it can also address the third one. We begin by observing the properties of sequence diagrams and we illustrate our discussion with an example.

4.1 Sequence Diagram

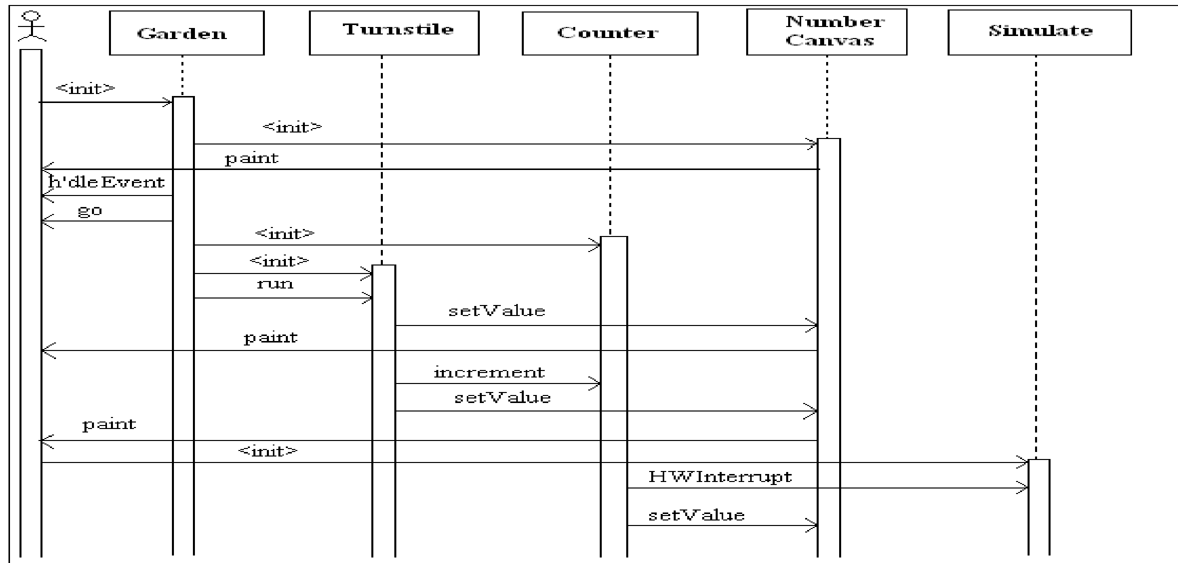
Sequence diagrams show the explicit sequence of interaction between objects. They also show the sequence of messages by means of which the objects communicate. In particular the time dimension is emphasized. However, they do not show the associations among the objects.

A sequence diagram has two dimensions:

1. the vertical dimension represents time, and,
2. the horizontal dimension represents the different objects.

Time proceeds from top to bottom and if desired, the axes can be interchanged.

The description that follows has been adapted from [OMG99a, Emme00a]. Objects in the sequence diagram are shown as rectangles. The annotation in the rectangle consists of an optional object name and a type identifier, separated by a colon. To differentiate the objects from the classes, which are also represented as rectangles in UML, the annotation of objects is underlined. Objects are shown as a vertical dashed line called the "lifeline"



Sequence diagram of Ornamental Garden
Figure 1

that indicates their lifetime. If the object ceases to exist, then the destruction point is shown by an "X" at the bottom of the lifeline.

Messages sent from one object to the next are shown as horizontal arrows. There are various flavours of arrows depending on the type of communication. An arrow with a filled solid arrowhead shows a local message implemented as procedure call. An arrow with a stich arrowhead shows a synchronous message. An arrow with a half stick arrowhead indicates an asynchronous communication between the two objects. A dashed arrow with a stick arrowhead indicates a return from a procedure call. Messages are ordered in time. In a concurrent system, a full arrowhead shows the yielding of a thread of control and a half arrowhead shows the sending of a message without yielding control.

A rectangular stripe along the lifeline represents the activation of an object. Activation corresponds to a period when the object performs an action, either directly or by sending a message to another object. The top of the stripe indicates the beginning of activation, while the bottom shows when the object is deactivated.

Screen caption of Ornamental Garden
Figure 2

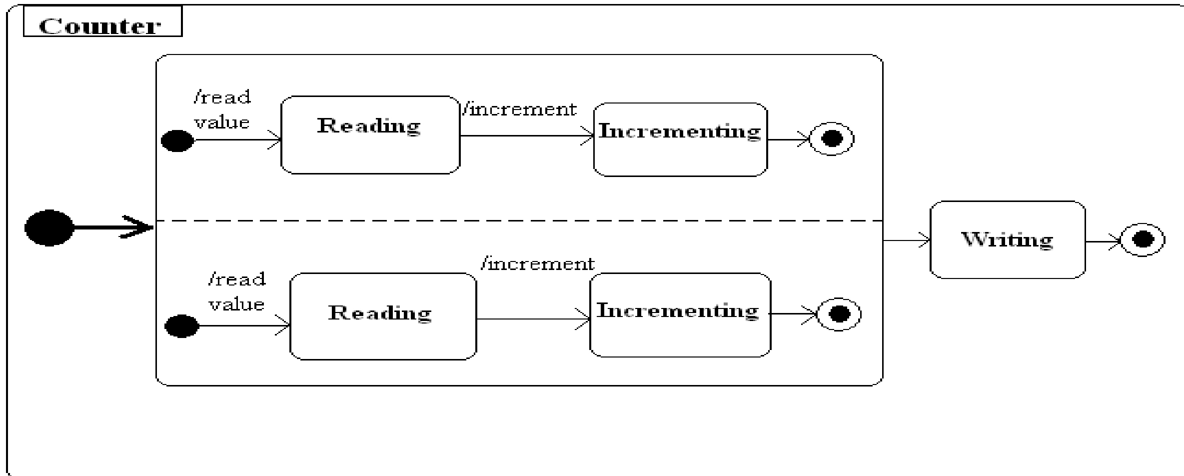
Fig. 1 depicts the Sequence diagram for the Ornamental Garden program [Magee99a] (see Fig. 2). This program is useful in demonstrating the interleaving of events and modelling mutual exclusion. As described above, the rectangles show the objects executing in the program. We have omitted the calls to the AWT library, so as not to blind the reader with too much unnecessary detail.

A sequence diagram, as described above, can be used during program execution to show the interleaving of events in the program. Deadlocks and livelocks can be identified by analysing the time sequencing of messages between objects and following the responses.

Premature termination of a thread together with its cause can be detected by keeping track of the messages that lead up to the scene of the crime. It should also prove useful in the preliminary detection of data race condition by following the sequence of actions that are being performed on an object. However, it is our belief that data race conditions are much better handled by *state diagrams*.

4.2 State Diagrams

State diagrams model the dynamic behaviour of objects. They describe the possible sequence of states and actions through which the object can proceed during its lifetime as a result of reacting to discrete events. A state is defined as a condition during the life of an object during which it satisfies some condition, performs some action or waits for some event.



State diagram for the Ornamental Garden program.
Figure 3

State diagrams are represented in UML as rounded rectangles (to represent the states) and directed edges (to represent the transitions between states). The annotation for the transition has two optional parts separated by a slash, as shown below.

Action-label '/' action-expression

The action-label specifies a condition that must be met in order for the transition to be performed. The action-expression defines the operation to be executed during the transition.

UML specifies two types of states: *simple* state and *composite* state. A composite state comprises of two or more sub states, which may be *sequential* or *concurrent*. There are no restrictions with regard to the number of nested component states within the state diagram.

Transition to and from concurrent states may have multiple source and target states. The default state of a composite state is depicted as an unlabelled transition originating from a filled circle.

Figure 3 shows the state diagram for the *Counter* state. The Counter object can transit from one of three states: *reading, incrementing* and *writing*. The two threads executing inside the counter object could either be reading or incrementing the value. Eventually, they will write the value on screen, although in the unsynchronised version, the exact sequence of these events is unknown, resulting in the scenario in Fig. 2.

The state diagram provides an alternative view to the sequence diagram, while maintaining the same level of abstraction. By analysing the transitions that the states of the respective objects go through, one can appreciate the nature of the problem, should a deadlock or live lock occur inside the program. We don't believe that state diagrams are very efficient at describing the interleaving of events since there is no notion of time sequencing mentioned. However, we believe that data race conditions can be detected by analysing the state of the object and its attributes prior to a transition being executed on the object.

5 CONCLUSION

The introduction of concurrency and object-orientation into the undergraduate curriculum has been regarded, by many researchers, as challenging. Students face the overwhelming task of learning and understanding the complex issues associated with both concepts in a very short period of time.

Research in visualisation aims at reducing their burden by providing effective and efficient tools to assist and interact with them to expedite their understanding of these concepts. However, there has been mixed success from existing tools in achieving this goal.

In this paper, we have presented research in the design of a visualisation tool based on UML to assist students. We have also outlined some issues related to the learning and teaching of concurrent object-oriented languages, such as Java. Work is currently under way to implement our visualisation tool. The main objective of this tool is to provide a simple, yet powerful tool to dynamically represent the execution of concurrent object-oriented programs.

REFERENCES

[Briot98a] J-P. Briot, R. Guerraoui, K-P. Lohr, *Concurrency and Distribution in Object-Oriented*

Programming in ACM Computing Surveys, Vol. 30, No. 3, September 1998.

[Byrne96a] 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.

[Choi00a] S.-E. Choi and E.C. Lewis, *A study of Common Pitfalls in Simple Multi-Threaded Programs*, in SIGCSE 2000 Proceedings, ACM, Austin, Texas, pp. 325-329, March 2000.

[Dijks68a] E. W. Dijkstra, *GO TO statement considered harmful*, Communications of the ACM, Vol. 11 No. 3, pp. 147-148, March 1968.

[Emme00a] W. Emmerich, *Engineering Distributed Objects,* John Wiley & Sons, pp. 35-42, 2000.

[Exton00a] C. Exton, *Dynamic Visualisation of Concurrent Object-Oriented systems*, in Proceedings of the IEEE International Workshop on Advanced Learning Technologies (IWALT2000), New Zealand, December 2000, pp 294-295.

[Exton00b] C. Exton and M. Kolling, *Concurrency, objects and visualisation*, in Proceedings of ACM SIGCSE Fourth Australian Computing Education Conference (ACE2000), Melbourne, December 2000, pp 109-115.

[Hendr00a] T. D Hendrix, J. H. Cross, S. Maghsoodloo and M. L. McKinney, *Do Visualizations Improve Program Comprehensibility? Experiments With Control Structure Diagrams for Java,* in SIGCSE 2000 Proceedings, ACM, Austin Texas, pp. 382-386, March 2000.

[Lea97a] D. Lea, *Concurrent Programming in Java: Design principles and patterns,* 2nd Edition, Addison-Wesley, 1999.

[Magee99a] J. Magee and J. Kramer, *Concurrency: State Models & Java Programs,* John Wiley & Sons, pp. 64-76, 1999.

[Matsu93a] S. Matsuoka and A. Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, pp. 107-150, 1993.

[OMG99a] Object Management Group, *UML Notation Guide Version 1.3.*

[Stask93a] J. Stasko, A. Badre and C. Lewis, *Do Algorithm Animations Assist Learning? An Empirical Study and Analysis,* in Proceedings of INTERCHI'93 Conference on Human Factors in Computer Systems, Amsterdam, pp. 61-66, April 1993.