1
Introduction

Between June 1985 and January 1987, a computerized radiation therapy machine called Therac-25 caused six known accidents involving massive overdoses with resultant deaths and serious injuries. Although most accidents are systemic involving complex interactions between various components and activities, and Therac-25 is not an exception in this respect, concurrent programming errors played an important part in these six accidents. Race conditions between different concurrent activities in the control program resulted in occasional erroneous control outputs. Furthermore, the sporadic nature of the errors caused by faulty concurrent programs contributed to the delay in recognizing that there was a problem. The designers of the Therac-25 software seemed largely unaware of the principles and practice of concurrent programming.

The wide acceptance of Java with its in-built concurrency constructs means that concurrent programming is no longer restricted to the minority of programmers involved in operating systems and embedded real-time applications. Concurrency is useful in a wide range of applications where responsiveness and throughput are issues. While most programmers are not engaged in the implementation of safety critical systems such as Therac-25, increasing numbers are using concurrent programming constructs in less esoteric applications. Errors in these applications and systems may not be directly life-threatening but they adversely affect our quality of life and may have severe financial implications. An understanding of the principles of concurrent programming and an appreciation of how it is practiced are an essential part of the education of computing science undergraduates and of the background of software engineering professionals. The pervasive nature of computing and the Internet makes it also an important topic for those whose primary activity may not be computing but who write programs none the less.

1.1 Concurrent Programs

Most complex systems and tasks that occur in the physical world can be broken down into a set of simpler activities. For example, the activities involved in building a house include bricklaying, carpentry, plumbing, electrical installation and roofing. These activities do not always occur strictly sequentially, one after the other, but can overlap and take place concurrently. For example, the plumbing and wiring in a new house can be installed at the same time. The activity described by a computer program can also be subdivided into simpler activities, each described by a subprogram. In traditional sequential programs, these subprograms or procedures are executed one after the other in a fixed order determined by the program and its input. The execution of one procedure does not overlap in time with another. In concurrent programs, computational activities are permitted to overlap in time and the subprogram executions describing these activities proceed concurrently.

The execution of a program (or subprogram) is termed a *process* and the execution of a concurrent program thus consists of multiple processes. As we see later, concurrent execution does not require multiple processors. Interleaving the instructions from multiple processes on a single processor can be used to simulate concurrency, giving the illusion of parallel execution. Of course, if a computer has multiple processors then the instructions of a concurrent program can actually be executed in parallel rather than being interleaved.

Structuring a program as a set of concurrent activities or processes has many advantages. For programs that interact with the environment to control some physical system, the parallelism and concurrency in that system can be reflected in the control program structure. Concurrency can be used to speed up response to user interaction by offloading time-consuming tasks to separate processes. Throughput can be improved by using multiple processes to manage communication and device latencies. These advantages are illustrated in detail in subsequent chapters. However, the advantages of concurrency may be offset by the increased complexity of concurrent programs. Managing this complexity and the principles and techniques necessary for the construction of well-behaved concurrent programs is the main subject matter of this book.

In order to illustrate the need for a rigorous approach to concurrent program design and implementation, let us consider an example.

Consider an automobile cruise control system that has the following requirements. It is controlled by three buttons: *resume, on* and *off* (Figure 1.1). When the engine is running and *on* is pressed, the cruise control system records the current speed and maintains the car at this speed. When the accelerator, brake or *off* is pressed, the cruise control system disengages but retains the speed setting. If *resume* is pressed, the system accelerates or de-accelerates the car back to the previously recorded speed.

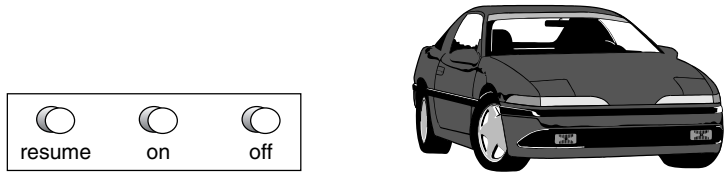

Figure 1.1 Cruise control system.

Our task is to provide a Java program that satisfies the specified requirements and behaves in a safe manner. How should we design such a program? What software processes should we construct and how should we structure them to form a program? How can we ensure that our program provides the behavior that we require while avoiding unsafe or undesirable behavior?

Given no guidance, we may be tempted simply to use previous design experience and construct the program as best as we can, using the appropriate Java concurrency constructs. To test the cruise control software, we could construct a simulation environment such as that illustrated in Figure 1.2. The website that accompanies this book contains this environment as an interactive Java applet for

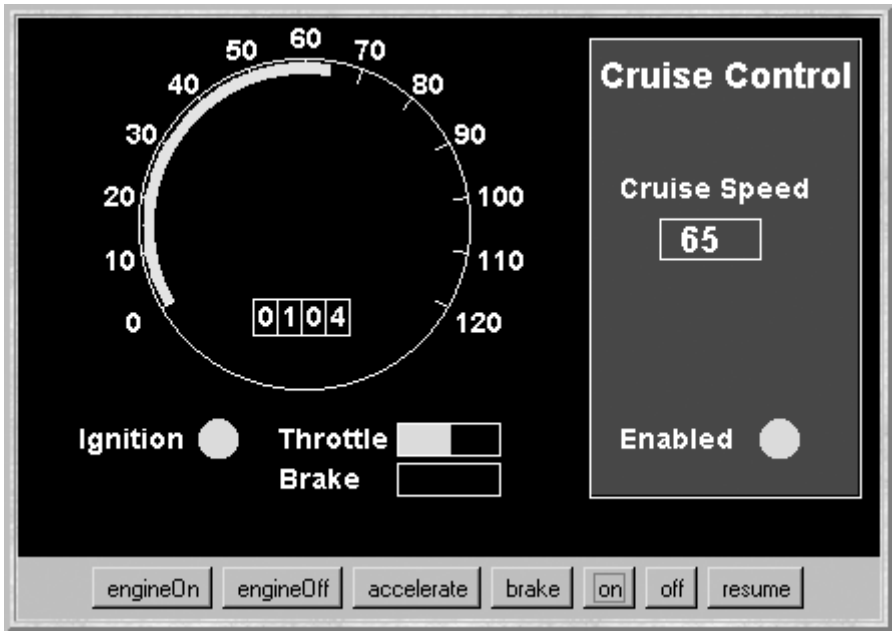

Figure 1.2 Simulation of the cruise control system.

use and experimentation (http://www.wileyeurope.com/college/magee). The buttons at the bottom of the display can be used to control the simulation: to switch the engine on or off; to *resume* or turn the cruise control system *on* or *off*; and to press the accelerator or brake (simulated by repeatedly pressing the relevant button).

The behavior of the system can be checked using particular scenarios such as the following:

- Is the cruise control system enabled after the engine is switched on and the *on* button is pressed?
- Is the cruise control system disabled when the brake is pressed?
- Is the cruise control system enabled when *resume* is then pressed?

However, testing such software is difficult, as there are many possible scenarios. How do we know when we have conducted a sufficient number of test scenarios?

For instance, what happens in the unlikely event that the engine is switched off while the cruise control system is still enabled? The system behaves as follows. It *retains the cruise control setting*, and, when the ignition is again switched on, the car accelerates so as to resume the previous speed setting!

Would testing have discovered that this dangerous behavior is present in the system? Perhaps, but in general testing is extremely difficult for concurrent programs as it relies on executing the particular sequence of events and actions that cause a problem. Since concurrent events may occur in any order, the problem sequences may never occur in the test environment, but may only show up in the deployed system, as with the Therac-25 machine.

There must be a better way to design, check and construct concurrent programs!

1.2 The Modeling Approach

A model is a simplified representation of the real world and, as such, includes only those aspects of the real-world system relevant to the problem at hand. For example, a model airplane, used in wind tunnel tests, models only the external shape of the airplane. The power of the airplane engines, the number of seats and its cargo capacity do not affect the plane's aerodynamic properties. Models are widely used in engineering since they can be used to focus on a particular aspect of a real-world system such as the aerodynamic properties of an airplane or the strength of a bridge. The reduction in scale and complexity achieved by modeling allows engineers to analyze properties such as the stress and strain on the structural components of a bridge. The earliest models used in engineering, such as airplane models for wind tunnels and ship models for drag tanks, were

physical. Modern models tend to be mathematical in nature and as such can be analyzed using computers.

This book takes a modeling approach to the design of concurrent programs. Our models represent the behavior of real concurrent programs written in Java. The models abstract much of the detail of real programs concerned with data representation, resource allocation and user interaction. They let us focus on concurrency. We can animate these models to investigate the concurrent behavior of the intended program. More importantly, we can *mechanically* verify that a model satisfies particular safety and progress properties, which are required of the program when it is implemented. This mechanical or algorithmic verification is made possible by a model-checking tool *LTSA* (Labeled Transition System Analyzer). Exhaustive model checking using *LTSA* allows us to check for both desirable and undesirable properties for all possible sequences of events and actions. *LTSA* is available from the World Wide Web (http://www.wileyeurope.com/college/magee). As it has been implemented in Java, it runs on a wide range of platforms, either as an applet or as an application program.

The models introduced in the book are based on finite state machines. Finite state machines are familiar to many programmers and engineers. They are used to specify the dynamic behavior of objects in well-known object-oriented design methods such as Booch (1986), OMT (Object Modeling Technique) (Rumbaugh, Blaha, Premerlani, *et al*., 1991) and, more recently, the all-encompassing UML (Unified Modeling Language) (Booch, Rumbaugh and Jacobson, 1998). They are also extensively used in the design of digital circuits – the original engineering use. For those not yet familiar with state machines, they have an intuitive, easily grasped semantics and a simple graphical representation. The state machines used in this book (technically, Labeled Transition Systems) have well-defined mathematical properties, which facilitate formal analysis and mechanical checking, thus avoiding the tedium (and error introduction) inherent in manual formal methods.

For instance, for the cruise control system described in section 1.1, we can model the various processes of the system as state machines. A state machine for the process responsible for obtaining the current speed is given in Figure 1.3. Starting from *state*(0), it indicates that once the engine is switched on, it transits to *state*(1) and can then repeatedly obtain a speed reading until the engine is switched off, when it returns to *state*(0). Other processes can be modeled similarly. We can compose the system from the constituent processes according to the proposed design structure, indicating the interactions between the processes. The advantage is that such models can be used to animate and check the behavior of the overall system *before* it is implemented. Figure 1.4 shows an animation of the model for the cruise control system. It clearly shows the problem encountered in our simulation: if the engine is switched off and on again when cruise control is enabled, the previous speed setting is resumed. Exhaustive analysis can be used to identify

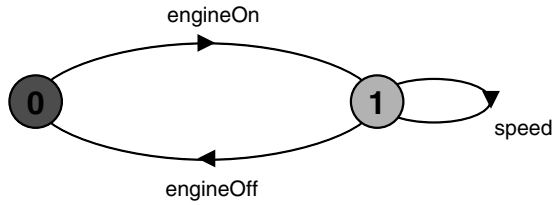

Figure 1.3 Speed input process.

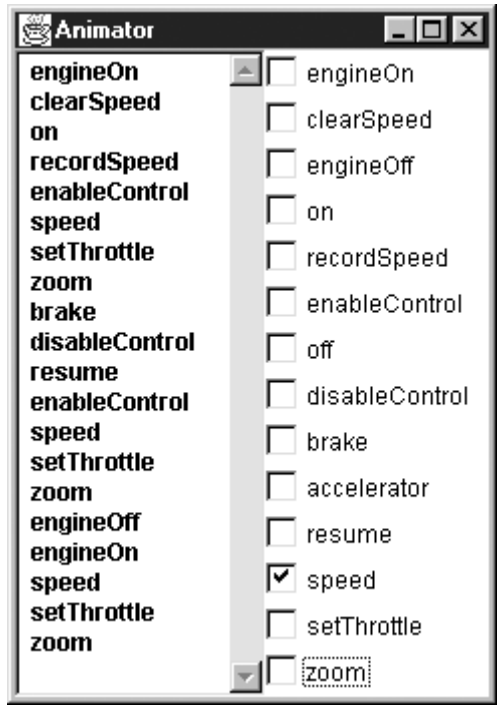

Figure 1.4 Animation of the cruise control system.

the problem under all possible situations. Furthermore, to help understand and correct the problem, the model checker produces the particular sequence of actions that led to it!

Later chapters describe and illustrate how to provide and use such models to gain confidence in the correctness and validity of a proposed design. We illustrate how premature and erroneous implementations can be avoided by careful modeling and analysis. Further, we indicate how such models can be

systematically transformed into Java programs. The cruise control system is fully described, modeled and implemented in Chapter 8.

Note that representing state machines graphically severely limits the complexity of problems that can be addressed. Consequently, we use a textual notation (Finite State Processes, *FSP*) to describe our models. The *LTSA* tool associated with the book translates *FSP* descriptions to the equivalent graphical description. The book itself presents the initial models in both textual and graphical forms to enable the reader to become familiar with the meaning of *FSP* descriptions. Technically, *FSP* is a process calculus – one of a family of notations pioneered by Milner (1989), Calculus of Communicating Systems (CCS), and Hoare (1985), Communicating Sequential Processes (CSP), for concisely describing and reasoning about concurrent programs. The difference from these notations is largely syntactic: *FSP* is designed to be easily machine readable. Like CCS and CSP, *FSP* has algebraic properties; however, it is used in this book primarily as a concise way of describing Labeled Transition Systems.

1.3 Practice

Previous authors of concurrent programming textbooks have been hampered by the lack of a widely available and generally accepted programming language with concurrency features. Java meets the criteria of availability and acceptance and has the advantage of being a general-purpose language with concurrency rather than a special-purpose language of restricted applicability. Consequently, we use Java exclusively as the language for programming examples. The simplicity of the concurrency features in Java is an advantage since more complex concurrency abstractions can be constructed and packaged as classes.

The full source of the set of example programs presented in the book is provided with the book and is available on the Web. In addition, all of the example programs may be run as applets in Web browsers. We believe that the ability to execute the programs is a significant aid to understanding the issues involved. The controls embedded in most of the example programs enable different execution scenarios to be set up, facilitating ''what if'' questions to be asked of the programs. The satisfaction of seeing (and experiencing) rather than merely believing is important in sustaining both interest and comprehension. This is as true for self-study as it is for formally taught courses.

The availability of Java on a wide range of platforms means that most readers will be able to treat both the modeling and programming problems included in the book as implementation rather than purely pen-and-paper exercises. In many of the problems a graphical interface is already provided so that the reader can concentrate on the concurrent programming component of the problem.

We make no apologies for including in the set of examples and exercises those that are sometimes disparagingly referred to as ''toy problems''. Typical of this class of example is the Dining Philosophers problem. The authors regard these examples as being valuable in condensing and crystallizing particular concurrent programming problems. They let the reader concentrate on the concurrency issues without the burden of understanding the application context. These examples are widely used in the literature on concurrent programming as a means of comparing different concurrent programming languages and constructs.

1.4 Content Overview

The concepts of concurrency are presented in a careful, systematic manner. Each concept is introduced and explained, indicating how it is modeled and implemented. In this way, state models and Java programs are presented hand-in-hand throughout the book. Furthermore, every chapter uses examples to illustrate the concepts, models and programs.

The next two chapters introduce the basic concepts of concurrent programming. Chapter 2 introduces the concept of a process, for modeling a sequence of actions, and a thread, for implementing such a sequence in Java. Chapter 3 introduces concurrency, both in the form of models of concurrent processes and in the form of multi-threaded programs.

The following two chapters deal with some of the basic problems associated with concurrency and the means for dealing with them. Chapter 4 discusses shared objects and the associated problem of interference if concurrent activities are allowed free access to such objects. This leads to the need for mutually exclusive access to shared objects. Further requirements for synchronization and coordination are introduced in Chapter 5, manifested as guarded actions in the models and monitors in Java.

Concurrent programs must be checked to ensure that they satisfy the required properties. One of the general properties is the absence of deadlock, where the program stops and makes no further progress. Deadlock is discussed in Chapter 6. Properties are generally described as either safety properties, concerned with a program not reaching a bad state, or liveness properties, concerned with a program eventually reaching a good state. These are usually specific to the particular application required. The modeling and checking of safety and liveness properties for Java programs are discussed in Chapter 7.

Chapter 8 reiterates the design approach used implicitly in the previous chapters, that of model-based design of programs. The cruise control system, discussed above, is used as the example.

The last six chapters of the book deal with a number of more advanced topics of interest. Chapter 9 deals with dynamic systems of processes and threads.

Chapter 10 deals with systems that interact using message passing. Chapter 11 discusses various concurrent software architectures, modeling and implementing common structures and patterns of interaction. Chapter 12 discusses timed systems, indicating how time can be modeled and included in implementations of concurrent programs. Chapter 13 addresses the problem of verifying implementations by modeling the relevant program language constructs and analyzing the resultant models. Finally Chapter 14 introduces fluents as a means of specifying properties in a state-based manner and of checking properties specified using a temporal logic.

Summary

This chapter has introduced the area of concurrent programs and justified the need for a model-based approach to design and construction. In particular:

- Finite state models are used to represent concurrent behavior. These can be animated and analyzed to gain confidence in the correctness and validity of a proposed design.
- The Java programming language is used for constructing concurrent programs. Java is general-purpose and has concurrency features.
- Examples and exercises are used throughout the book to illustrate the concepts and provide the opportunity for experimentation and learning by experience.

Notes and Further Reading

A comprehensive description of the Therac-25 incident and investigation can be obtained from the paper, *An investigation of the Therac-25 accidents*, by Nancy Leveson and Clark Turner (1993).

The automobile cruise control system is a simplified version of a real system. The example is fully discussed in Chapter 8.

There are a number of existing books on concurrency and concurrent programming. A collection of original papers on the invention and origins of concurrent programming from the mid 1960s to the late 1970s is presented in the book by Per Brinch Hansen (2002), *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Ben-Ari (1990) provides a simple introduction to the area in his book, *Principles of Concurrent and Distributed Programming*. A comprehensive coverage of the area, with logical reasoning and many examples, is provided by Greg Andrews (1991) in his book, *Concurrent Programming: Principles and Practice*. A further readable text in the area is that by Burns and Davies (1993), *Concurrent Programming*. A formal, logic-based approach is provided by Fred Schneider

(1997) in his book, *On Concurrent Programming*. For more on the pragmatics of object-orientation and concurrent programming in Java, readers may consult the book by Doug Lea (1999), *Concurrent Programming in Java™: Design Principles and Patterns*. A recent comprehensive text is that by Vijay Garg (2004), *Concurrent and Distributed Computing in Java*.