Concurrent System Design:
Applied Mathematics & Modeling in Software Engineering Education

Michael J. Lutz, James R. Vallino
Rochester Institute of Technology

Introduction

A hallmark of engineering design is the use of models to explore the consequences of design decisions. Sometimes these models are physical prototypes or informal drawings, but the *sine qua non* of contemporary practice is the use of formal, mathematical models of system structure and behavior. Whether circuit models in electrical engineering, heat-transfer models in mechanical engineering, or queuing theory models in industrial engineering, mathematics makes it possible to perform rigorous analysis that is the cornerstone of modern engineering.

Until recently, such modeling was impractical for software systems. Informal models abounded, such as those created in UML[1], but rigorous models from which one could derive significant properties were either so rudimentary or so tedious to use that it was difficult to justify the incremental benefit in other than the most critical of systems. In part this is a reflection of the relative immaturity of software engineering, but it also reflects a key distinction between software and traditional engineering: whereas the latter builds on numerical computation and continuous functions, software is more appropriately modeled using logic, set theory, and other aspects of discrete mathematics. Most of the models stress relationships between software components, and numerical computation is the exception rather than the norm.

Recent advances in both theory and application have made it possible to model significant aspects of software behavior precisely, and to use tools to help analyze the resulting properties[2,3,4]. In this paper, we focus on a course developed by James Vallino and since taught and modified by Michael Lutz, to present formal modeling to our software engineering students at RIT. Our overall goals were three-fold: To acquaint our students with modern modeling tools, to connect the courses they take in discrete mathematics to real applications, and to persuade them that mathematics has much to offer to the engineering of quality software.

Concurrency and FSP: Models to Implementations

Concurrency is a ripe area for formal modeling, in part because of the types of systems that embody concurrency. Real-time, safety-critical systems, for instance, often are structured using concurrent tasks. Failures in a safety-critical system are serious enough that the design may warrant more careful formal modeling than would be the case when developing standard desktop applications.

The first-year programming sequence that our students take introduces basic thread concepts including creation and synchronization. An ability to analyze, design and implement multi-threaded applications is an important skill area for current software engineering practice. The brief introduction our students receive in their first year describes the mechanisms that a

Copyright © 2005, American Society for Engineering Education

language and operating system provides for threading and synchronization within an application. The student may be able to recite the necessary and sufficient conditions for deadlock to exist but not see how those conditions might arise in an application being designed. At this basic level, students do not know how to engineer concurrent systems that are free of race conditions and deadlocks. The RIT Software Engineering program provides this deeper coverage in the upper-level technical elective course Principles of Concurrent Software Systems.

For a modeling methodology to be useful for the design of concurrent systems it should meet two criteria. First, the formalisms should be at a level that reduces the scale and complexity of the system sufficiently to allow the software engineer to analyze its important concurrent properties such as deadlock and progress checks. Second, there should be tool support available so that the analysis is done mechanically rather than by hand. The Finite State Process (FSP) modeling technique described by Magee and Kramer satisfies both of these criteria. Based on finite state machines, the basics of FSP modeling has been seen by the students before and is easy to grasp. The system uses a text notation for defining the FSP models. Individual sequential models use standard finite state machine semantics (mutually exclusive states, instantaneous execution of actions causing transitions). Students do not have difficulty modeling non-concurrent FSP's. The example below defines two FSP models. The first is for an electrical switch and the second defines the operation of a lamp controlled by two three-way switches. Tokens in `CAPS` represent processes in the methodology. The `LAMP` process is defined using two local processes, **ONEON** and **ON**. Tokens in lower case are actions that cause the FSP to take a transition to another state when the action is executed.

```
SWITCH = (on -> off -> SWITCH).

LAMP = (on -> ONEON),
ONEON = (off -> LAMP | on -> ON),
ON = (off -> ONEON).
```

Figure 1 – Finite State Process (FSP) Examples

FSP models can also be shown graphically. The figures below are graphical representations of the two FSP models defined in Figure 1 above. Standard finite state diagramming is used. The red state node represents the current state of the FSP. All models start in state 0.

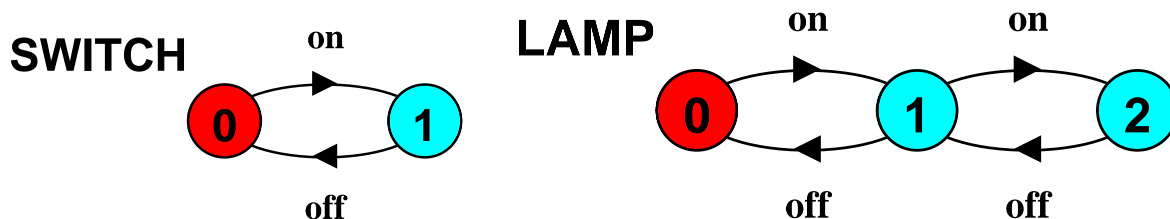

Figure 2 – Graphical Representation of Finite State Processes

Copyright © 2005, American Society for Engineering Education

Modeling of concurrent systems is accomplished by composing multiple sequence FSP's into a single parallel composition. The notation allows labels to be applied to models so that multiple instances of one FSP can be added to the parallel composition. The model defines two **`SWITCH`** FSP's with labels **`a`** and **`b`**. The labels are prepended to all actions in the FSP. These two switches are composed with a lamp to form a three-way lighting system, Figure 3. Concurrently executing FSP's are synchronized by matching action names. If an action is shared by multiple FSP's, all FSP's sharing that action must execute it at the same time. The action name matching is done considering all labels applied to the actions. In the **`THREEWAYLIGHT`** composite the **`LAMP`** FSP shares its actions with switch **`a`** and **`b`**.

```
||THREEWAYLIGHT = (a:SWITCH || b:SWITCH || {a,b}::LAMP).
```

Figure 3 – Composite Finite State Process

Figure 4 shows the individual FSP's that make up the **`THREEWAYLIGHT`** composite. The composite FSP can also be represented in a graphical form. For the example system the composite diagram is shown in Figure 5. Each state in the composite diagram represents a combination of states for all of the individual FSP's. Without synchronization the state space of the composite will be the multiplication of the number of states in each component FSP. With shared action synchronization the composite model often obtains a state space reduction. The synchronization eliminates some state combinations. For example, the composite state of **`a:SWITCH.0`** – **`b:SWITCH.0`** – **`LAMP.2`** is not possible. This would represent both switches being off and the lamp being turned on.

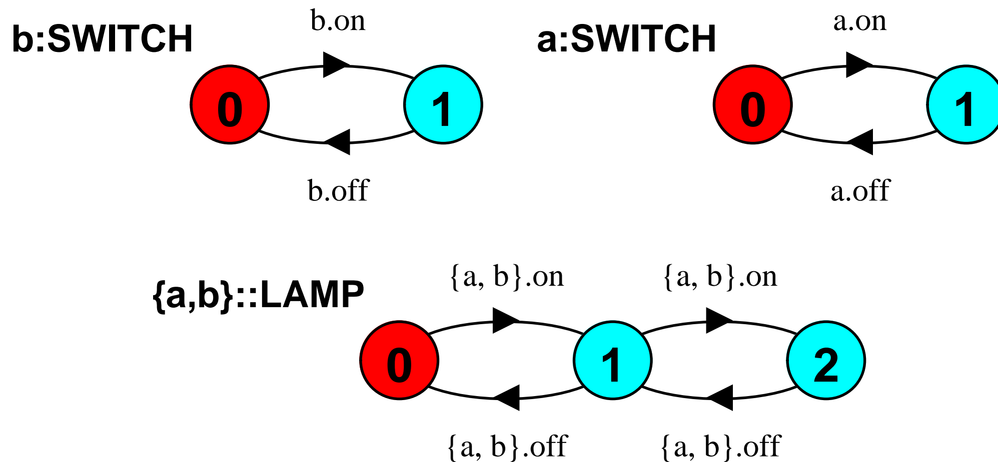

Figure 4 – Labeled Finite State Process Components of THREEWAYLIGHT

Copyright © 2005, American Society for Engineering Education

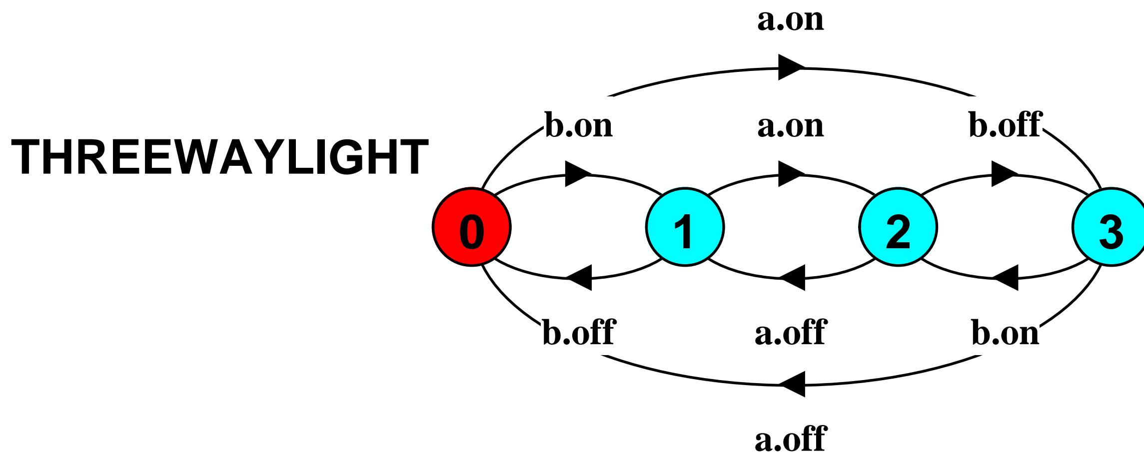

Figure 5 – Composite Finite State Process

The tool that is used for defining and analyzing the FSP's is the Labeled Transition System Analyzer. Figure 6 shows the editing pane for this application. Unlike some tools used in industrial practice, such as Rational's Rose Real-Time and ILogix's Rhapsody, LTSA can be quickly learned by students; with just a few hours of lab time, students know how to work within the LTSA environment.

Features provided by LTSA allow the student to experiment with their models. A trace tool lets the student manually execute the model by triggering actions and watching how the system reacts. The actions available at each step are of particular interest to the student. By running a simulation and studying the actions available at each point the student can determine if the synchronization within the model is working correctly.

The model checking features within the LTSA allow for analysis of other concurrent properties of the model. The tool checks for deadlock conditions, safety violations and progress failures. For safety issues, the student defines a correct sequence for actions to be executed by the composite process. The model checker determines if there is any trace of actions within the composite FSP that would violate this sequence. Progress checks determine if there are regions of the composite FSP in which actions that must be executed can never be triggered.

Copyright © 2005, American Society for Engineering Education

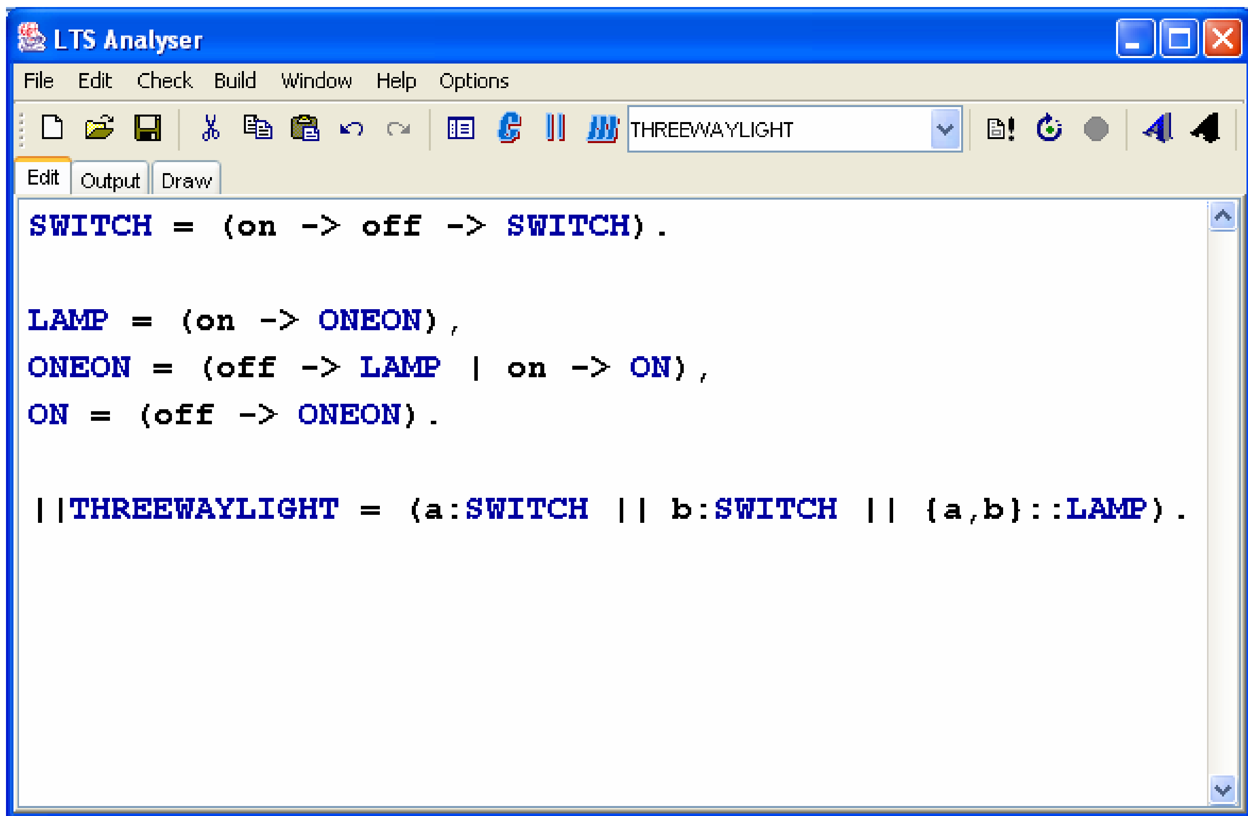

Figure 6 – Labeled Transition System Analyzer (LTSA) Tool

Analyzing the FSP model that captures the essential elements of concurrency and component interaction provides the student with a better understanding of the requirements for the system he or she is designing. Developing the system model forces the student to study the requirements of the system, focused on the concurrency aspects and the synchronization that is necessary. The model checking analysis should indicate that the model does not violate any safety or progress properties that the designed described.

To complete the "models to implementation" process, component FSP's are categorized as active or passive elements in the system. Active elements drive the system and are associated with threads of execution. Passive elements provide control and are associated with synchronization primitives. In the class we describe and provide examples of a straightforward mapping from model primitive to code implementation primitives. Having mathematically proven that the model does not contain any violations, the intention is to keep the implementation as closely tied to the model as possible. The optimal situation would be automatic code generation from the model. The LTSA does not provide that feature and students will manually do the implementation following guidelines for translating from model element to implementation. This provides a somewhat mechanical conversion to generate the code for the concurrency framework captured in the model.

Copyright © 2005, American Society for Engineering Education

In this course, we assign projects such as small transit system simulations and computer games. Each of these projects requires the team to develop an FSP model and use that to drive the implementation. One problem with the finite state process definition used by LTSA is state-space explosion. The composite FSP has exponential growth for the number of states in the system. The larger projects that we assign in this course will commonly have millions of states in the composite. While LTSA can handle systems of this size, a naïve approach to modeling will exceed the capacity of the tool. This actually aids in the student learning, in that it forces them model the system at a level of abstraction that captures all the essential concurrency issues while at the same time fits within the capacity of the LTSA.

Successes and Failures

The first time we taught the Principles of Concurrent Software Systems course we did not explicitly cover the formal Finite State Process semantics. The software engineering students will have taken two quarters of discrete mathematics as a prerequisite, so the class discussion was informal – we assumed that the students would recognize the application of the discrete math principles in the definition of FSP semantics.

We were quite surprised, then, when over 75% of the students answered "Not applicable" to the question "How much did this course require you to demonstrate an ability to apply mathematics, especially discrete mathematics, to the modeling and analysis of software systems?" We have since changed the syllabus for the course to explicitly discuss the formal semantics for each FSP feature when it is presented. Students now appropriately recognize that while they may not be "doing discrete math" they are applying it in the design and analysis of these concurrent systems.

We emphasize to the students the process of moving from model to implementation by having the model drive the structure of the implementation. Our students have not completely embraced this methodology. In grappling with the design and implementation of concurrent systems, our students are at a point similar to where they were when first learning object-oriented design and programming. Initially, most students did not trust their FSP designs enough to use them to drive their implementation. Instead they often created the model simply to satisfy a project requirement. The implementation then progressed in a chaotic fashion, and only when the student saw the working system was he or she confident of the design as embodied in the code. Students do not have the confidence that the FSP model of a concurrent system does represent a correctly functioning system. They do not trust their ability to use the model to create a working implementation.

We have observed, however, that the emphasis on modeling in the course benefits students even if they abandon the model during implementation. Modeling helps students gain a better understanding of the requirements for the system and the thread synchronization points.

We are happy to report that most students see the benefit of analyses that can be done on the concurrent model. Even though the LTSA tool is not "industrial strength", some students have reported using it to solve industrial problems while on co-op. One student in particular, who had taken an early offering of Principles of Concurrent Software Systems, reported used LTSA while on co-op assignment with a manufacturer of digital signal processing (DSP) equipment. With

Copyright © 2005, American Society for Engineering Education

each new release of the hardware board the application programmers were given an updated version of the protocol specification for interfacing with the DSP board. With one release, the co-op student sensed that there was a problem in the protocol but could not pinpoint it. At this point the standard approach might be to build a skeleton implementation of the application program and see how it operates. This typically would result in a large number of hours of testing and debugging to uncover what may be a subtle problem in the protocol.

The student, however, remembered the features provided by LTSA; with an afternoon of effort he modeled the protocol, executed traces, and ran safety checks. The analysis uncovered a progress property failure that prevented the protocol from continuing to completion under certain unusual circumstances. The student discussed this problem with the hardware designers and they acknowledged that he had uncovered a problem with the protocol. The model highlighted the exact problem that was latent in the system and eliminated typical finger pointing between the hardware and software engineers.

Conclusion

Java with its concurrency primitives defined in the language specification has accelerated the use of concurrent programming for applications. It is now standard practice in areas beyond the embedded and real-time systems domain. We believe that an ability to deal with concurrent systems is fundamental to current software engineering practice. As a result, we are making a curriculum revision to require Principles of Concurrent Software Systems as part of each student's program of study.

Requiring this course will serve two purposes for our program. First, it ensures that all students completing our program will have the deeper exposure to concurrent system analysis, design and implementation. Second, the discussion of the discrete mathematics concepts at the core of finite state processes provides an opportunity for all students to demonstrate their achievement of one of our program outcomes: Apply mathematics, especially discrete mathematics, to the modeling and analysis of software systems.

Bibliography

1. Blaha, M., and Rumbaugh, J. *Object-Oriented Modeling and Design with UML (Second Edition)*. Prentice-Hall, 2005.
2. Holzmann, G. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
3. Magee, J., and Kramer, J. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.
4. Jackson, D., Shlyakhter, I., and Sridharan, M. "A Micromodularity Mechanism." ACM Conference on Foundations of Software Engineering / European Software Engineering Conference. September, 2001.

Biographies

Michael J. Lutz has a Masters in Computer Science from SUNY Buffalo. He joined the RIT faculty in 1976, where he remained until taking a two year industrial leave in 1987-88. His experience on leave led him to head the team that created the nation's first baccalaureate program in software engineering at RIT. His interests include software design, mathematical modeling, and software engineering education.

Copyright © 2005, American Society for Engineering Education

James R. Vallino has a PhD in Computer Science from the University of Rochester. He joined the RIT faculty in 1997 after finishing his PhD work in augmented reality. Previously, he worked in industry for over 16 years doing software development in embedded systems, industrial automation, and communications. He is interested in software design particularly for real-time and embedded systems, and software engineering education.

Copyright © 2005, American Society for Engineering Education