

Java Simulator for Modeling Concurrent Events

Weston Jossey
College of Computer and Information Science
Northeastern University
Boston, MA 02115
wjossey@ccs.neu.edu

Simona Boboila
College of Computer and Information Science
Northeastern University
Boston, MA 02115
simona@ccs.neu.edu

ABSTRACT

Synchronization appears in several computer science scenarios that involve concurrent access to shared resources. Due to their prevalence and importance, a good understanding of synchronization concepts is essential in learning operating systems. This paper presents a simulation framework that students can use to explore concurrency in the context of classical synchronization problems. The simulator features a thread scheduler, which enforces an ordered execution of concurrent threads. Furthermore, the simulator collects meaningful statistical data that are used to generate the equivalent Markov model of the simulation. Students can use the steady state probabilistic model as a validation method of how well their solution behaves compared to the theoretical Markov model. We illustrate how the simulator can be utilized to model, implement and experiment with the well-known barber shop problem.

1. INTRODUCTION

Synchronization of concurrent events is one of the main topics discussed in any operating systems curriculum. In computer science, synchronization refers to the coordination of simultaneous threads or processes in order to avoid race conditions [20]. Synchronization is conceptually difficult to grasp, because reasoning about parallel executions and the interactions among them is hard. This makes the task of presenting the concept to students particularly challenging.

Most textbooks introduce synchronization techniques such as mutexes, semaphores, and monitors and show how these techniques are used to implement classical synchronization problems: producers and consumers, readers and writers, sleeping barber or dining philosophers [16, 20, 21, 22]. However, using synchronization primitives adds a new layer of abstraction and difficulty to any synchronization problem. Before juggling with different synchronization techniques, it is beneficial for students to have a good understanding of the synchronized program flow and thread interactions.

This paper presents Biju¹ (BasIc Java concUrrency) [1], a simulation framework that can be used to model concurrent executions [10]. The simulator provides a framework that facilitates solving some synchronization problems, without

¹A famous bear from a Romanian zoo. Original from the Carpathian Mountains, Biju was the main attraction, the 'bijou' (jewel, in French) of the zoo at the beginning of the 20th century. Nowadays, his offspring, also called Biju, inhabit the same home.

the need to use semaphores, monitors or other synchronization primitives. In this way, the simulator can be efficiently used in the first steps of teaching synchronization concepts. It allows students to focus on thread coordination and program flow, rather than language specific synchronization techniques.

A correctly synchronized program execution is similar to a queueing theory model: several processes arrive at a queue, wait in the queue and are finally serviced by other processes [7, 12]. Thus the program execution can be described by a series of steady states and the associated state probabilities. As a result, one can theoretically determine the equivalent Markov model [15] of a synchronization problem and use it for validation against the empirically determined state probabilities. In a Markov model, the current state fully captures all the information that influences the future evolution of the system [15]. It is represented by a graph, where the vertices are the states and the edges are probabilistic transitions from one state to another. Biju provides the validation facility based on Markov models. Using the simulator, students can implement and experiment with various synchronization problems. Moreover, the simulation framework provides the means to generate meaningful statistics that can be used to validate students' implementation against the Markov model.

The rest of the paper is structured as follows: Section 2 presents related work in the field of educational software for teaching computer science. Section 3 introduces the simulator. Section 4 describes how the simulator can be used to solve the classical barber shop problem. Also, it illustrates the Markov model validation technique as a method to analyze some implementation issues (e.g. deadlocks). Section 5 concludes the paper.

2. RELATED WORK

Exploring efficient methods of teaching computer science is an ongoing task. Teaching frameworks and simulators have been developed to assist students and professors in several fields of computer science. In the area of data structures and algorithms, Kurtz [9] resorts to simulation to teach recursion and binary tree traversals, while a comprehensive collection of tools that help students visualize how algorithms work can be found at [6]. Aubidy [8] and Hatfield [13] present simulators of basic internal computer operations, which can be used to teach computer organization and architecture. Xu [14] describes a teaching framework targeted to intermediate-

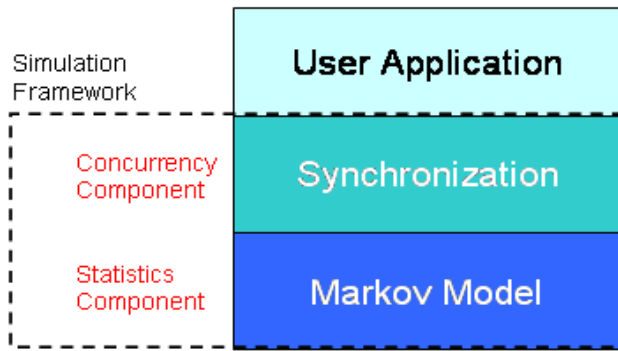


Figure 1: A layered structure of the simulation components.

level undergraduate programming course, that would help students learn the fundamentals of compiler design and language processing. Educational software has also been developed in the area of formal languages and automata, to simulate and experiment with Turing machines, finite state and pushdown automata, and other related topics [3, 5, 11].

In operating systems, some relevant contributions are Nachos [4], an educational operating system simulator, and Robbins’ Java-implemented simulators [17] that illustrate processor scheduling, producer consumer synchronization, starving philosophers, fork-pipe, disk head scheduling, address translation and concurrent I/O. Our work is closest to Robbins’ simulations of synchronization problems. However, his simulators are targeted to particular problems, such as producer consumer [18] or starving philosophers [19], while our goal is to provide a simulation framework that would enable users to implement and experiment with a larger set of synchronization problems. Furthermore, Robbins’ approach offers higher usability (e.g. it facilitates user-application interactions through a simulation window), while in our case the focus is on functionality (e.g. we provide a validation method).

3. SIMULATOR

The building blocks of Biju are concurrency simulation and Markov modeling. The current section first gives a high-level description of the simulator, and continues with a few implementation aspects.

3.1 Description

Figure 1 presents a layered structure of the components that are integrated in a simulation scenario. The first layer is the Markov Model, which provides the theoretical representation of the problem. While several problems can be modeled with Markov chains, we are only interested in a subset of them, namely synchronization problems. Synchronization represents the next concept (second layer) that the students need to understand before starting the implementation of a particular problem (third layer). Markov Models and Synchronization concepts have direct correspondents in the simulation framework: Statistics Component and Concurrency Component, respectively.

The Statistics Component of the simulation framework pro-

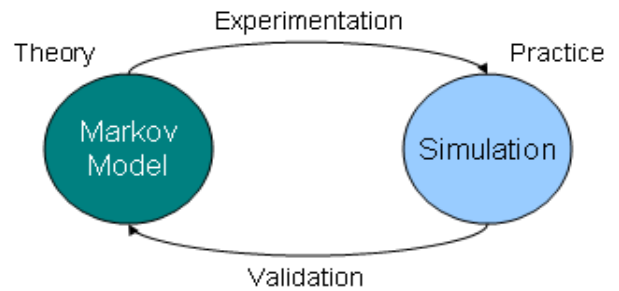


Figure 2: Relation between the theoretical model and simulation.

vides the functionality to generate and visualize the equivalent Markov chain and the steady state probabilities. As shown in Figure 2, the Statistics Component aids students in getting a perspective view of how theory and practice relate. Starting from the Markov model of a particular synchronization problem, students can implement and run various simulation experiments to see how their implementation behaves in practice. Next, students can use the simulation framework to generate statistical data and compute the steady state probabilities obtained during simulation. Finally, the simulation results can be validated against the original theoretical model.

The Concurrency Component features a Simulatable environment and a Scheduler. The Simulatable environment provides a “middle-man” for communication between the thread scheduler and the running thread, as well as four implemented methods to serve as the Java equivalent of notifyAll, notify, wait, and sleep. The Scheduler allows for only one thread within the simulation to be running at a single time. We give a detailed description of the Concurrency Component in Section 3.2.

3.2 Implementation

The simulator is a Java application that can be run locally on a computer. It is event driven and uses a virtual time which is incremented during the simulation process.

The underlying implementation of Biju is a basic thread scheduler. All threads that wish to be run under Biju need extend the Simulatable abstract class, which requires that the programmer implement a simulatable method. The simulatable method is analogous to a run method that would be implemented for a Runnable interface.

3.2.1 Behind the Scenes

The scheduler, which is the core part of Biju, is a sequential scheduler that permits only one thread to execute at any particular time. It achieves this by locking threads from execution using traditional Java concurrency techniques after they “register” with the scheduler. Registration is handled automatically by the abstract class Simulatable; thus, the programmer need not have a notion of how Biju’s underlying implementation works.

The four methods inherited by any class that extends `Simulatable` are `Notify`, `NotifyAll`, `Sleep`, and `Wait`. `Notify`, `NotifyAll`, and `Wait` are identical in their implementations with their Java namesakes. The `Sleep` method is also identical to its namesake, except it takes in a `Long`, rather than a `Double`.

The scheduler has a hierarchy of scheduling concerns. First and foremost, the scheduler attempts to allow for threads which are already running to continue execution. To do this, the scheduler maintains a list of all threads that are currently "running." The second order of precedence is that of all the threads which are currently sleeping. These threads are stored in a priority queue, with the next thread to awake from sleep at the top of the priority queue. Since the scheduler is not capable of interrupting threads mid-computation, switching occurs when the executing thread within the simulation invokes `Sleep` or `Wait`.

If there are no threads running, or sleeping, then the scheduler has reached a deadlock; thus, it will never terminate. This scenario means that either all of the simulation threads have completed executing, or there was an issue in the implementors solution that caused deadlock. In this case, deadlock can occur when all threads remaining are in a wait hold where their underlying condition cannot be satisfied.

3.2.2 Simulation Time

Simulation time is the amount of elapsed time relative to the simulation, without any relation to the real clock-time. With `Biju`, simulation time only changes when a sleeping thread is awoken, causing the simulation time to increment by however long that given thread was intended to sleep for. One could make the analogy that a "Sleep" is equivalent to telling the scheduler that it takes an explicit amount of time for a certain computation to complete. By utilizing simulation time, rather than real time, we are able to measure computation times more accurately, because the overhead of the programming environment is not taken into account.

3.3 Visualizing Statistics

When confronted with complicated concepts that are difficult to visualize, it can be useful to utilize visualization tools to augment generated statistics. In the case of `Biju`, the open source visualization tool `Graphviz` [2] is deployed to produce meaningful visual representations of the statistical models. Unfortunately, due to slow development and cross-platform compatibility issues, automating the process is highly difficult. With the current version, manual steps need to take place to coalesce all of the information.

`Graphviz` utilizes a language called `DOT`, which is a simple assignment based language that `Graphviz` can interpret to generate graphs. The statistical framework, which is bundled with `Biju`, automatically generates a `DOT` representation of the final statistical results. The generated output is printed to the console once the simulation has completed. `Graphviz` has a number of ways in which it can parse text, but the most common form is to just save the text in a file with the `.dot` extension, and then use the command line `Graphviz` tool to generate either a `jpg`, `png`, or the file format of user's choice. Example results are presented in Section 4 (Figure 4).

```
chairs = N    /* number of seats */
waiting = 0   /* number of customers waiting */
sleeping = false;

barber()
  /* loops forever,
   * sleeping and cutting hair */
  while true
    while waiting == 0
      sleeping = true
      wait()
      sleeping = false
    sleep()    /* cuts hair */
    waiting --
    notify()   /* calls the next
               * waiting customer */

customer()
  if waiting < chairs
    waiting ++
    if sleeping
      notify() /* wakes up barber */
    wait()
  else
    leave()   /* no more empty chairs */
```

Figure 3: Barber shop problem, pseudo-code.

4. BARBER SHOP PROBLEM

We illustrate how the simulator can be used to implement and experiment with one classical synchronization problem: the barber shop [22]. The problem is based on the following scenario: When there are no customers in the shop, the barber sits in his chair and sleeps. When a customer arrives, he checks to see if there are any empty chairs. If all of the chairs are occupied, the customer leaves. Otherwise, he either awakens the barber or, if the barber is cutting someone else's hair, waits in one of the empty chairs.

4.1 Solution

In the analogous synchronization problem, the barber and the customers are each represented by threads, and the problem becomes a queueing problem: the customer threads wait in the queue until they are serviced by the barber thread. The simulator features a bakery algorithm implementation: when a notification is sent, the thread that joined the waiting queue first is woken up, which enforces a FIFO order of handling waiting threads.

Figure 3 gives a pseudo-code solution of the barber shop problem. As mentioned in Section 3, `notify()`, `wait()`, and `sleep()` are provided through the simulation framework. An equivalent implementation with monitors requires two condition variables, one for the customers and one for the barber. Similarly, an implementation with semaphores requires three semaphores: one for the customers, one for the barber, and one (mutex) to protect the critical region in which the number of waiting customers is updated. The simulation framework eliminates the need to use synchronization

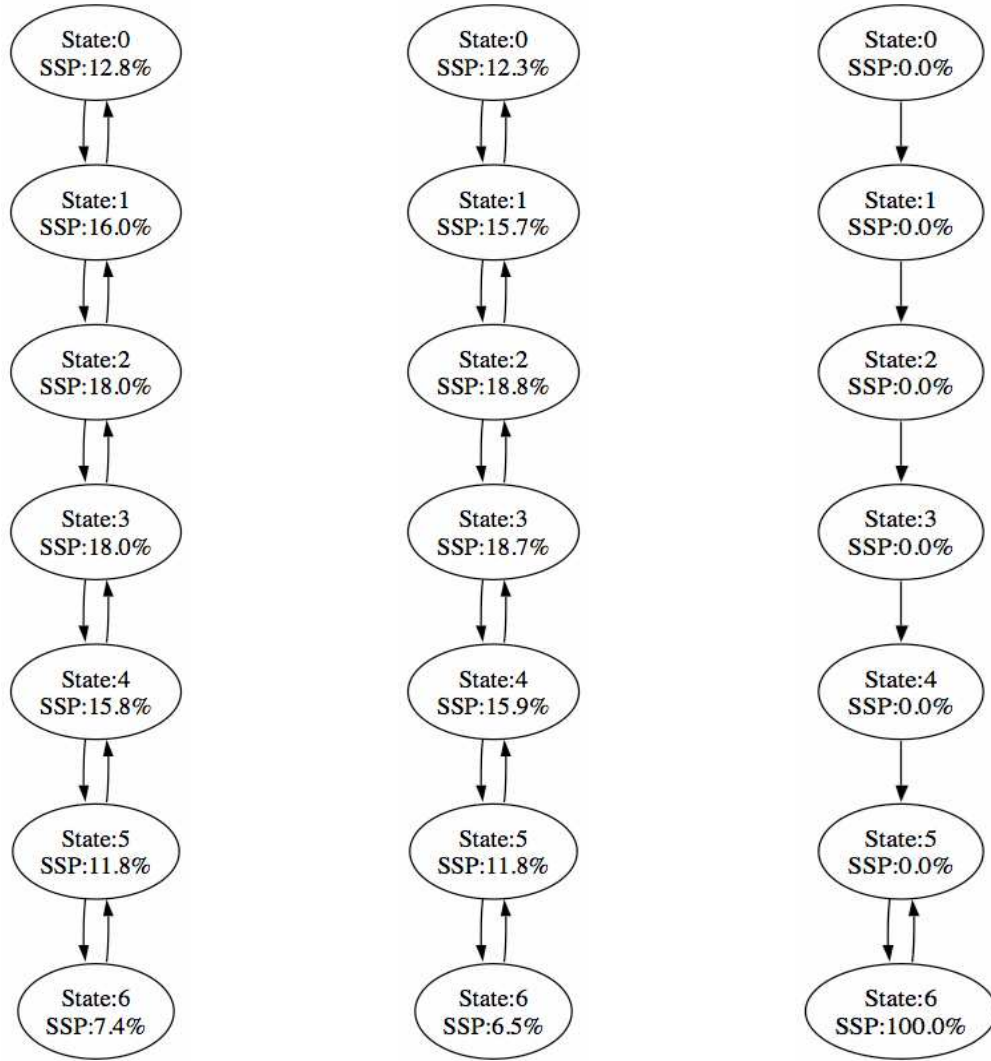


Figure 4: a) Theoretical Markov model. b) Simulation results. c) Deadlock.

primitives. Having an underlying synchronization mechanism which is transparent to students puts the focus entirely on the central issue of threads coordination. That eases students' effort of reasoning about synchronization. Students can concentrate on understanding, designing and implementing the synchronization algorithm, while the simulator handles the instrumental aspects of the implementation transparently.

4.2 Validation

In our barber shop implementation example, there are six chairs in the shop (five in the waiting area and one for getting the haircut). As a result, there are between zero and six customers in the shop at any time, waiting to be serviced, which leads to seven states in total. In our simulation, the barber spends 1.25 seconds for a haircut, having a constant service rate of 0.8. There are ten customers for which the trip to the shop takes 10 seconds. Since the number of customers is finite, their arrival rate is affected by the number of customers already waiting in the shop. With these pa-

rameters, the theoretical Markov model of the barber shop is illustrated in Figure 4a).

Biju collects statistical information during execution and computes the corresponding empirical steady state probabilities. Figure 4b) shows simulation results obtained with the barber shop implementation described before. We see that the state probabilities are well correlated with the expected probabilities from the theoretical model. The similarity between expected and simulated results can be used by students as a measure of how well their implementation of the synchronization problem behaves. A closer correlation means a higher confidence in their solution.

The statistical data and the visualization facility of Biju can also be used to analyze incorrect program behaviors. The following is an example of bad implementation, where race conditions occur. In an erroneous implementation, a new customer might forget to wake up the sleeping barber before he begins to wait in his chair (Figure 5). This would

```

/* If the barber is asleep,
 * the customer wakes him up */
if(sleeping == true)
    Notify();

```

Figure 5: Code that can lead to deadlock. If the code is missing, or 'sleeping' is incorrectly tested for false value instead of true, the barber can not wake up.

inevitably lead to deadlock: the barber waits to be woken up, while the customers that sit in the chairs wait to be serviced. None of them is able to move to the next execution state because they are blocked. Figure 4c) shows how the deadlock affects the state probabilities. The barber shop remains "trapped" in state six (which represents a full shop) throughout the entire simulation.

Other bad implementations of the barber shop problem may lead to starvation: if customers are not served in an orderly manner, some of them may never get a haircut even though they have been waiting. Since the scheduler implements a FIFO notification algorithm for waiting threads, starvation can generally be overcome with the simulator.

5. CONCLUSIONS

Biju is a simulation framework designed to help students in their effort to understand and experiment with concurrency concepts. The simulator can be used in the context of an undergraduate operating system course to implement and analyze various synchronization problems. An important insight that students can gain with the simulator is the correlation between practical (simulation) results and theoretical (estimated) results.

While the current version of Biju is fully functional, there are a number of additional features that would also benefit end-users. First and foremost, integrating Graphviz directly with Biju would eliminate all manual generation of graphs. This addition would make Biju even more streamlined and efficient, further cutting out any middle-man that could lead to compatibility frustrations. Another possible extension is the ability to programmatically input a state model with transition rates, which could also be used for verification purposes, so long as the generated matrix has a solution. This would resemble the JFLAP model [3] of constructing finite state automata. Altogether, it is intended that this framework remain small and targeted towards a specific task; thus, future work will be more geared towards usability, rather than additional functionality features.

6. REFERENCES

- [1] Biju simulator - google code page.
- [2] Graphviz - graph visualization software. <http://www.graphviz.org/>.
- [3] Jflap, software for experimenting with formal languages topics. <http://www.cs.duke.edu/cs423/jflap/>.
- [4] Nachos - an educational operating system simulator. <http://www.cs.washington.edu/homes/tom/nachos/>.
- [5] Turing machine simulator. <http://www.ironphoenix.org/tril/tm/>.
- [6] Visualizations of data structures and algorithms. <http://www.cs.princeton.edu/wayne/cs423/demos.html>.
- [7] I. Adan and J. Resing. *Queueing Theory*. Department of Mathematics and Computing Science, Eindhoven University of Technology, 2001.
- [8] D. Aubidy, Kasim M. Al. Teaching computer organization and architecture using simulation and fpga applications. *Journal of Computer Science*, 2007.
- [9] D. J. Barry L. Kurtz. Using simulation to teach recursion and binary tree traversals. *SIGCSE*, 17:49 – 54, 1985.
- [10] M. Ben-Ari. *Principles of Concurrent and Distributed Programming, 2nd edition*. Addison-Wesley, 2006.
- [11] J. Bovet. Visual automata simulator. <http://www.cs.usfca.edu/jbovet/vas.html>, 2004-2006.
- [12] R. B. Cooper. *Introduction to Queueing Theory, 2nd edition*. North Holland, 1981.
- [13] R. M. L. J. Hatfield, B. Incorporating simulation and implementation into teaching computer organization and architecture. *Frontiers in Education*, 19:FIG – 18, 2005.
- [14] F. G. M. Li Xu. Chirp on crickets: teaching compilers using an embedded robot controller. *SIGCSE*, pages 82 – 86, 2006.
- [15] S. P. Meyn and R. Tweedie. *Markov Chains and Stochastic Stability, 2nd edition*. Cambridge University Press, 2008.
- [16] G. Nutt. *Operating Systems, A Modern Perspective, 3rd Edition*. Addison-Wesley Publishing Company, 2003.
- [17] S. Robbins. Simulators for teaching computer science. <http://vip.cs.utsa.edu/simulators/>, 1999-2007.
- [18] S. Robbins. Experimentation with bounded buffer synchronization. *SIGCSE*, 32:330 – 334, 2000.
- [19] S. Robbins. Starving philosophers: experimentation with monitor synchronization. *SIGCSE*, pages 317 – 321, 2001.
- [20] A. Silberschatz and P. B. Galvin. *Operating System Concepts, 8th edition*. Addison-Wesley Publishing Company, 2009.
- [21] W. Stallings. *Operating Systems: Internals and Design Principles, 6th Edition*. Prentice Hall, 2008.
- [22] A. S. Tanenbaum. *Modern Operating Systems, 2nd edition*. Prentice Hall, 2001.