Using Model Checkers in an Introductory Course on Operating Systems*

Roelof Hamberg
Embedded Systems Institute
P.O. Box 513, 5600 MB Eindhoven
the Netherlands
Roelof.Hamberg@esi.nl

Frits Vaandrager
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen
the Netherlands
F.Vaandrager@cs.ru.nl

ABSTRACT

During the last three years, we have been experimenting with the use of the UPPAAL model checker in an introductory course on operating systems for first-year Computer Science students at the Radboud University Nijmegen. The course uses model checkers as a tool to explain, visualize and solve concurrency problems. Our experience is that students enjoy to play with model checkers because it makes concurrency issues tangible. Even though it is hard to measure objectively, we think that model checkers really help students to obtain a deeper insight into concurrency. In this article, we report on our experiences in the classroom, explain how mutual exclusion algorithms, semaphores and monitors can conveniently be modeled in UPPAAL, and present some results on properties of small, concurrent patterns.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, deadlocks, mutual exclusion, synchronization*; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Theory, Verification

Keywords

Operating system course, Model checkers, Concurrency, Mutual exclusion, Semaphores, Monitors

1. INTRODUCTION

Each year, thousands of Computer Science students are exposed to introductory courses on operating systems and study one of the numerous textbooks in this area, for instance Tanenbaum & Woodhull [22], Stallings [20], Nutt [17], or Silberschatz & Galvin [19]. All these textbooks contain one or more chapters on principles of concurrency, with a discussion of fundamental concepts such as mutual exclusion algorithms, semaphores, monitors, message passing, deadlock and starvation.

For beginning students concurrency is a difficult subject. To begin with, it is hard to visualize dynamic concurrent behavior in a static book. As a reader one often needs four hands, like the Hindu god Vishnu, to simultaneously point at the different control locations of a concurrent program, as well as at the explanatory text. Usually, no correctness proofs are given in textbooks on operating systems. Authors do not want to bother their readers, that is the students, with tedious formal proofs, since this would distract attention from the key issues they want to get across. But contrary to their intuitive intention, this does not make life easy for for students. Students know concurrency is tricky, that deadlocks, race conditions and starvation scenarios are hard to avoid, and that program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [7]. But how then should they convince themselves of the correctness of concurrent algorithms and programs?

Also for instructors, grading assignments on concurrency poses major challenges. Students often come with "creative" solutions to concurrency problems in which, for instance, numerous semaphores are used in intricate ways. How to determine whether such solutions are correct? Many instructors will admit that frequently they give a student the maximal score, simply because they have not been able to spot any mistake. But this does not mean these solutions are correct!

Many experts agree that concurrency is the next major revolution in how we write software [21]. Applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains that have now started becoming available and will continue to materialize over the next several years. For example, Intel is talking about someday producing 100-core chips; a single-threaded application can exploit at most 1/100 of such a chip's potential throughput. This implies that concurrency should be a major topic in any course on operating systems. Race conditions, deadlock

*All the models discussed in this article are available through the homepage of the second author `http://www.cs.ru.nl/F.Vaandrager/`.

and starvation are not just things studied in a distant past by operating system pioneers such as Dijkstra: our students need a thorough understanding of these issues in order to be able to build the applications of tomorrow.

Model checking is emerging as a practical engineering tool for automated debugging of complex reactive systems such as embedded controllers and network protocols [4, 11, 1]. In model checking, required or hypothesized properties of the system are expressed as (temporal) logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specified property holds or not. Extremely large state-spaces can often be traversed in minutes. This year, the ACM has named Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis the winners of the 2007 A.M. Turing Award for their original and continuing research on model checking. We think that after 20 years of research on model checking this technology has become sufficiently mature and it is time to change the way in which we teach principles of concurrency:

1. Using the input language of model checkers it is straightforward to express concurrency algorithms in terms of networks of communicating state machines. Algorithms are usually explained using pseudo code and/or text. However, for understanding algorithms it greatly helps to see how pseudo code and text correspond to precise automaton models and assertions about these models. By specifying state transitions, we make explicit which operations are atomic and which operations are not, a key issue in concurrent programming.

2. Using the (graphical) simulators provided by some modern model checkers it becomes easy to visualize the dynamics of concurrent algorithms, in particular traces of the evolving system in which mutual exclusion is violated, starvation occurs, etcetera.

3. Students may convince themselves of the correctness of algorithms without having to spend time on tedious, manual correctness proofs, which are of independent interest but belong in a different course: here the verification is done fully automatically by the model checker.

During the last three years, we have been experimenting with the use of the Uppaal model checker in an introductory course on operating systems for first-year (second semester) Computer Science students in Nijmegen. We decided not to tell students about the theory and algorithms behind model checking, but to focus on how a model checker can be used to explain, visualize and solve concurrency problems. A model checker is just like a pocket calculator: a tool that does the math for you. In fact, the few hours spent teaching the tool helped us to speed up learning other OS algorithms and methods.

Uppaal [1, 2] is an integrated tool environment for specification, validation and verification of systems modeled as networks of timed automata. It is available for free for non-profit applications at `www.uppaal.com`. The language for the new version Uppaal 4.0 features a subset of the C programming language, a graphical user interface for specifying networks of extended finite state machines (EFSMs), and syntax for specifying timing constraints. We selected the tool because of its nice graphical user interface, which makes it very easy to use. In fact, after less than one hour of training students are able to the first assignments on mutual exclusion algorithms.

Our experience is that students very much enjoy to play with model checkers because it makes concurrency issues tangible. Even though it is hard to measure objectively, we think that model checkers really help students to obtain a deeper insight into concurrency. Last year, for instance, students participating in our course discovered several deep mistakes in a published textbook [8], simply by modeling and analyzing proposed solutions from the book using Uppaal.

In this article, we report on our experiences in the classroom, and explain how a variety of concurrency related concepts can be conveniently modeled in Uppaal. Section 2 discusses models of some basic mutual exclusion algorithms, Section 3 is devoted to models of semaphores and concurrency problems that use semaphore, and Section 4 presents models involving monitors. Finally, in Section 5, we present some conclusions and discuss related work.

2. MUTUAL EXCLUSION

Software solutions for the mutual exclusion problem are rarely used in practice, since at the hardware level mutual exclusion can be realized using test-and-set or equivalent instructions. Nevertheless, most textbooks present various concurrent programming solutions for mutual exclusion that have been proposed in the literature, since this provides an excellent way to introduce students to some fundamental issues in concurrency. In our course, we have been using Uppaal to visualize and analyze the behavior of a number of mutual exclusion algorithms. As an illustration we discuss here two models of Peterson's algorithm.

In its original formulation, Peterson's algorithm [18] is stated for two processes `P(0)` and `P(1)` that work in parallel on a single resource. In pseudo code, the algorithm for process `P(pid)` reads as follows:

```
while(true)
  {
   flag[pid] = true
   turn = 1-pid
   while( flag[1-pid] && turn == 1-pid );
           // do nothing
   // critical section
   ...
   // end of critical section
   flag[pid] = false
  }
```

The algorithm uses three variables, `flag[0]`, `flag[1]` and `turn`. A `flag` value of `1` indicates that the process wants to enter the critical section. The variable `turn` holds the pid of the process whose turn it is.

Figure 2 shows a Uppaal model of process `P(pid)`. As one

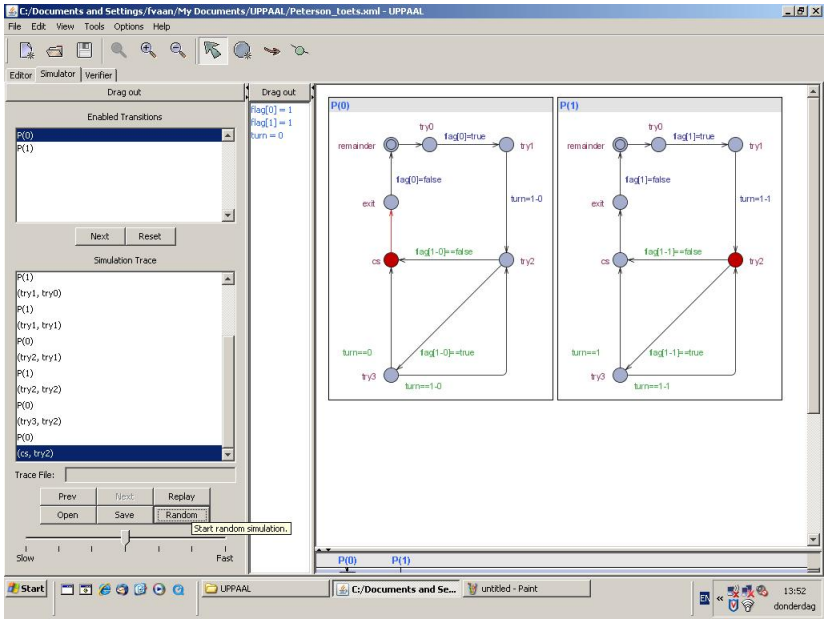

Figure 1: Screen dump of Uppaal simulation of Peterson's algorithm.

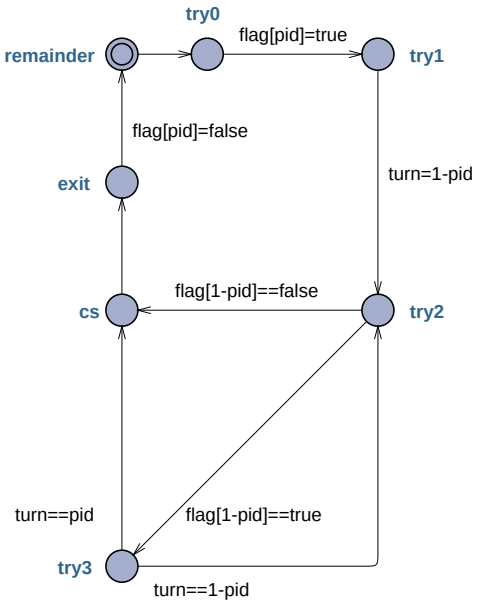

Figure 2: Model of Peterson's algorithm.

can see, the translation between pseudo code and UPPAAL is straightforward. Basically, there is a location in the automaton for each line of code. However, a fundamental aspect of the algorithm that is explicit in the UPPAAL model but left implicit in the pseudo code, is that evaluation of the condition `flag`$[1-$`pid`$]$&&`turn` $==1-$`pid` is not atomic. It may happen, for instance, that first process `P(0)` reads variable `flag[1]`, subsequently process `P(1)` takes a number of steps, and only after that process `P(0)` reads variable `turn`. The model therefore contains two locations to capture the evaluation of the condition: in location `try2` process `P(pid)` reads variable `flag`$[1-$`pid`$]$ and in location `try3` it reads variable `turn`.

Figure 1 shows a screen dump of a simulation of Peterson's algorithm in UPPAAL. Red dots indicate the current control location of each process. During simulation a user may manually select possible transitions, or perform a random simulation. A useful feature of UPPAAL is that counterexamples that have been found by the verifier can be replayed within the simulator. Using UPPAAL, it is trivial to verify that Peterson's algorithm indeed satisfies *mutual exclusion*, that is, for all reachable states (`A[]` in temporal logic notation) it holds that `P(0)` and `P(1)` can not be in their critical section at the same time:

`A[] not(P(0).cs and P(1).cs)`

UPPAAL immediately finds a counterexample to the claim made in Wikipedia about the algorithm[1] that "If `P0` is in its critical section, then `flag[0]` is `1` and either `flag[1]` is `false` or `turn` is `0`". If we ask UPPAAL to check the corresponding property

```
A[]  P(0).cs  imply
        (flag[0]==1 && (flag[1]==0||turn==0))
```

it produces the obvious counterexample — which can be replayed in the simulator — in which first `P(0)` enters the critical section and then `P(1)` performs its first assignment.

An important property of Peterson's algorithm is *bounded waiting*: a process will not wait longer than approximately one turn for entrance to the critical section. In order to

[1] `http://en.wikipedia.org/wiki/Peterson's_algorithm`, version 9-7-2008.

state and prove this property in UPPAAL, we add timing constraints to the model: an upper bound `l` on the execution time of instructions, and an upper bound `c` on the critical section time. Figure 3 shows the enriched model. The idea is that each process has a local clock `x`, which is

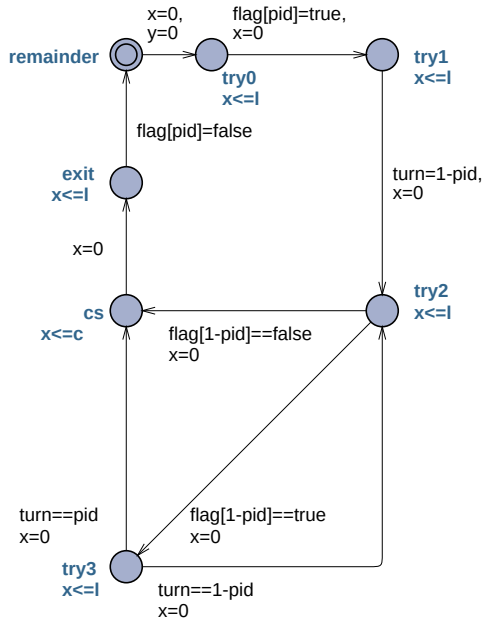

Figure 3: Peterson's algorithm with timing.

reset before entering a location. The invariant `x <= l` in location `try0` ensures that the time spent in this location is at most `l`. This models the upper bound `l` for performing the instruction `flag[pid] = true`. In a similar way we have added time bounds to the rest of the model. For arbitrary integer values of the parameters `l` and `c`, UPPAAL can establish that the time from when a particular process enters `try0` until it enters `cs` is at most $c + 10 * l$. This is done by introducing a local clock `y` for each process, which is reset whenever the process enters location `try0`. UPPAAL can then check that:

```
A[]  (P(0).try0||P(0).try1||P(0).try2||P(0).try3)
      imply P(0).y <= c+10*l}
```

This property says that a process stays at most $c + 10 * l$ time units in the trying region. Since a process can only leave the trying region by entering the critical section, this implies that a process must enter the critical section after at most $c + 10 * l$ time units. If we change the bound to $c + 10 * l - 1$ then the property no longer holds, and UPPAAL produces a counterexample. This result is consistent with Theorem 10.14 from Lynch [15], which establishes an upper bound of $c + \mathcal{O}(l)$. Our result is stronger in the sense that we give a precise upper bound on the number of instructions. The result in Lynch [15] is stronger in the sense that it holds for all values of `c` and `l`, whereas we have only checked a couple of instances.

UPPAAL is not able to prove general liveness properties for the *untimed* model of Figure 2, such as the temporal logic formula `P(0).try0` $\leadsto$ `P(0).cs` (whenever process `P(0)` enters the trying region, it will eventually enter the critical section). Such properties can easily be checked using other model checkers such as SPIN [11] and SMV [4, 3]. These tools however miss the graphical user interface of UPPAAL and are not so easy to use for people without background in formal methods. It would be useful (and not too difficult) to establish a link between UPPAAL and SPIN or SMV that would allow one to model check temporal logic formulas for untimed UPPAAL models.

We do not expect first-year students to come up independently with UPPAAL models such as those in Figures 2 and 3. However, these models are very helpful to explain the operation of the algorithm through the UPPAAL simulator. Students find it easy to understand the models and to modify them in order to answer various questions about the algorithm such as "Is Peterson's algorithm still correct if we change the evaluation order of the conditions `flag`$[1 - $`pid`$]$ and `turn` $== 1 -$ `pid`?" Also, once students understand the UPPAAL model of one mutual exclusion algorithm, they are able to also model other mutual exclusion algorithms. For instance, in less than half an hour most students manage to construct a model of Hyman's algorithm [12] and discover — using UPPAAL — why this algorithm is flawed.

3. SEMAPHORES

Semaphores [5] constitute a classic method for restricting access to shared resources. They are widely used in practice and are the primitive synchronization mechanism in many operating systems. Solutions that use semaphophores are portable and usually efficient. Even though they have been criticized for being too unstructured and difficult to use, all major textbooks on operating systems discuss semaphores and their use in solving classic problems of synchronization. In this section, we explain how we modeled semaphores in UPPAAL, and how model checkers can be used to analyze solutions to synchronization problems.

In UPPAAL, transitions between states may be labeled by output actions or input actions. A transition with an output action `a!` from one automaton may synchronize (occur simultaneously) with a transition with a matching input action `a?` from a different automaton. A semaphore `s` is modeled as an automaton that interacts with its environment via three types of synchronization actions: `semWait`[`s`][`p`]?, `semSignal`[`s`][`p`]? and `semGo`[`s`][`p`]!, where `p` is a process identifier. Figure 4 gives a schematic representation. A semaphore maintains an integer variable `count` to record the number of shared resources still available, and a list `queue` with names of processes that are waiting. Whenever a process `p` wants to access a resource protected by `s`, it performs a synchronization action `semWait`[`s`][`p`]! that synchronizes with a matching action `semWait`[`s`][`p`]? of `s`. If `count` is positive, then `s` will immediately react with a synchronization action`semGo`[`s`][`p`]!, and `count` is decremented. Upon performing the matching action `semGo`[`s`][`p`]?, process `p` may access the resource. If `count` is less than or equal to 0, then process identifier `p` is stored in `queue` and `count` is decremented. A process `p` releases a resource protected by `s` via a synchronization `semSignal`[`s`][`p`]!. After a matching `semSignal`[`s`][`p`]? transition, the semaphore increments `count`. If `count` was nega-

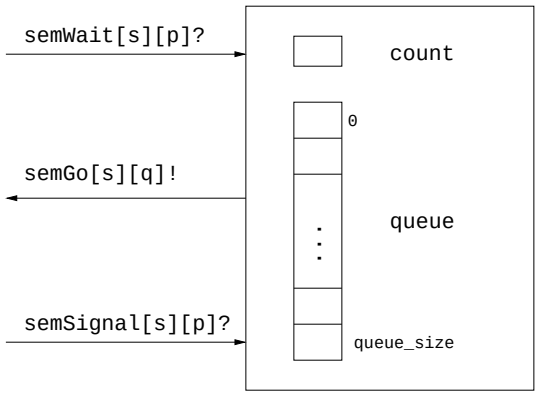

Figure 4: Schematic view of semaphore model.

tive before this transition then, in addition, the first process identifier `q` is removed from `queue` and activated via an action `semGo[s][q]!`. We assume that processes are activated in FIFO order.

Figure 5 displays our UPPAAL model of a semaphore. Students

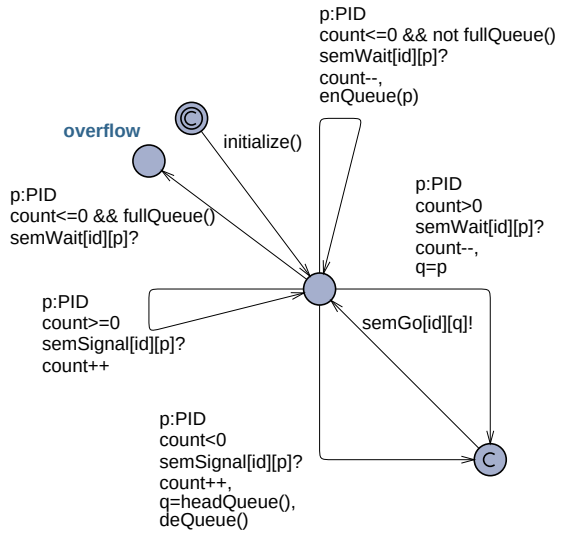

Figure 5: Model of a semaphore.

do not need to understand the details of the code; they just use this model as a black box when solving synchronization problems. The semaphore template has three parameters: (1) `id`, the unique identifier of the semaphore, (2) `init_val`, the initial value of the semaphore, and (3) `queue_size`, the maximal number of processes in the waiting queue. Since UPPAAL does not support dynamically growing data structures, we need to fix an upper bound on the size of the queue. In our model, the queue is implemented as an array of `queue_size`. If, due to a `semWait`, a new element needs to be added to a queue that is full, the automaton jumps to a special `overflow` location. UPPAAL needs to verify that `overflow` can not be reached. Since currently in UPPAAL it is not possible to initialize a parametrized array, we need a special transition to do this.[2] By making the initial location "committed" we ensure that the initialization takes place before any other activity in the system. In several transitions we use a select field `p : PID`. This indicates that we have instances of these transitions for each `p` in the set `PID`, that is, for each process identifier.

Note that in our modeling approach the usual `semWait(s)` operation from the textbooks translates into two consecutive transitions labeled with actions `semWait[s][p]!` and `semGo[s][p]?`, respectively. Each `semSignal(s)` operation by `p` is encoded by a transition with synchronization action `semSignal[s][p]!`.

A UPPAAL model for the binary semaphore is obtained as a small and obvious variation of the general semaphore model.

3.1 Producer/Consumer Problem

Now we have models of semaphores, we can start playing with them! Figure 6 shows a model of the incorrect solution to the infinite-buffer producer/consumer problem using binary semaphores, as discussed by Stallings [20] on pages 221-224. The model is obtained in a straightforward man-

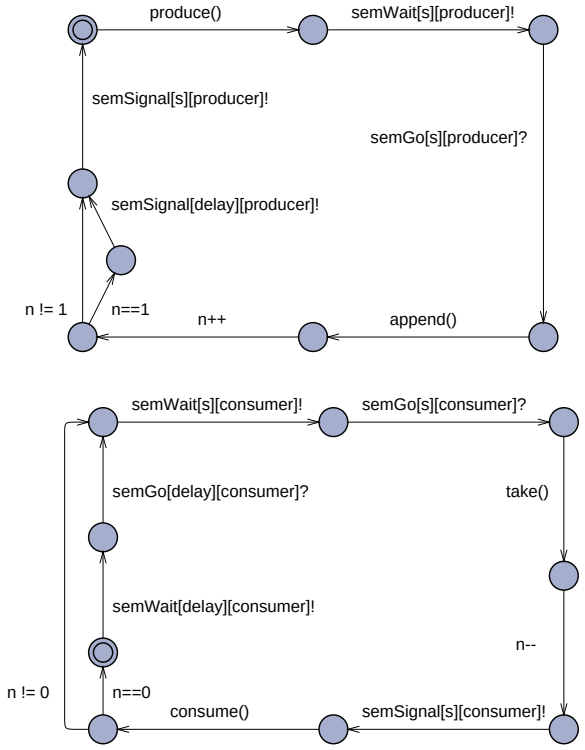

Figure 6: Models of producer and consumer for incorrect solution to producer/consumer problem.

ner from the code presented in [20][Fig. 5.9]. The integer variable `n` keeps track of the number of items in the buffer. The binary semaphore `s` is used to enforce mutual exclusion; the binary semaphore `delay` is used to force the consumer to `semWait` if the buffer is empty. As Stallings [20] points

[2]This imperfection of UPPAAL actually has a positive consequence: due to the extra transition the automaton has a striking resemblance with a bee!

out, the problem with this solution is that variable `n` may become negative, that is, the consumer may consume an item from the buffer that does not exist. By checking the query `E <> n < 0` with the verifier ("there exists a path to a state in which `n < 0`"), UPPAAL produces a counterexample almost instantaneously. Essentially (modulo permutation of independent transitions), this is the same counterexample as the 21 step example presented by Stallings in Table 5.3. The ability of UPPAAL to replay counterexamples in the simulator greatly helps in understanding what goes wrong. Note that the model checker is not able to explore the full (infinite) state space of this model.

We also modeled the solution to the bounded-buffer producer/consumer problem with general semaphores presented in [20][Fig. 5.13]. This model is shown in Figure 7. The

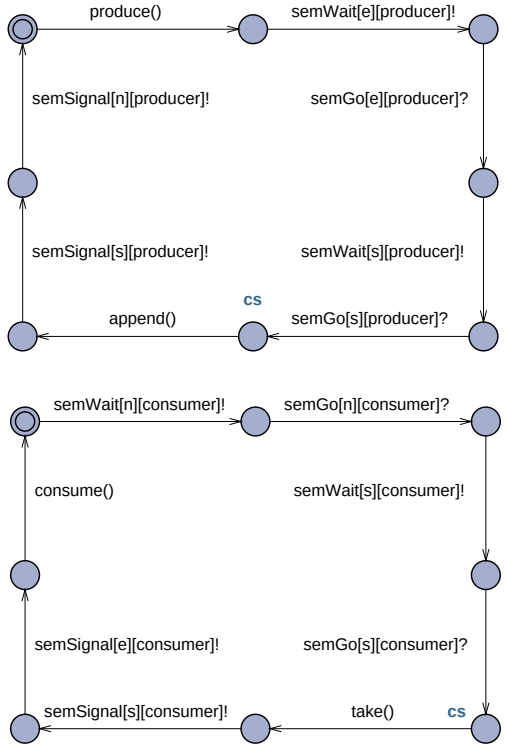

Figure 7: Models of producer and consumer for correct solution to bounded-buffer case.

variable `n` is now a semaphore. Its value still is equal to the number of items in the buffer. The semaphore `e` keeps track of the number of empty spaces. Stallings [20] claims correctness of this solution, but does not prove it. Even for large values of `sizeofbuffer` up to 10,000, UPPAAL can prove mutual exclusion and absence of deadlock automatically within a few seconds. After introducing an auxiliary variable `buffer` that is incremented by function `produce`() and decremented by function `consume`(), UPPAAL can prove that always `buffer >= 0` and `buffer <= sizeofbuffer`, that is, the consumer never consumes an item that does not exist, and there is no buffer overflow.

The above solution to the bounded-buffer producer/consumer problem is also presented by Tanenbaum and Woodhull [22].

The authors observe that when the order of the wait operations in the Producer code is reversed there is a deadlock. This observation can easily be verified using UPPAAL; the model for the producer is shown in Figure 8.

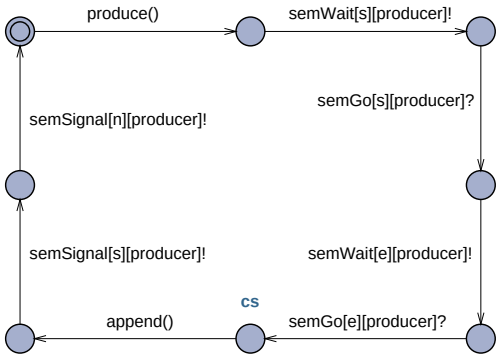

Figure 8: Model of the producer for the faulty solution to the bounded-buffer case.

3.2 Dining Philosophers

The first somewhat more involved example of a synchronization problem that is included in nearly all textbooks, is the problem of the dining philosophers. Dijkstra proposed this problem first [6] as an examination question about a synchronization problem and it surely has become a classic. Five philosophers think and eat in alternation, but in order to eat they need two forks, each of which is shared with a neighboring philosopher at a round table.

Without any precautions, deadlock arises by the sequence of events in which all philosophers pick up their left fork first. After that, they have to wait forever for the other fork: deadlock! Figure 9 shows a UPPAAL model of the naive solution to the dining philosophers problem. The model checker finds the deadlock in a fraction of a second.

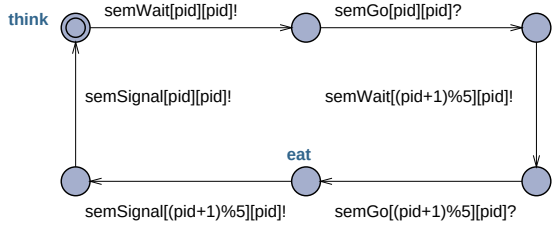

Figure 9: Model of naive dining philosopher.

To overcome deadlock, one common approach is to assume the presence of a doorman who only allows four philosophers at a time into the dining room. A model along these lines is shown in Figure 10. Now it is easy to prove absence of deadlock using UPPAAL. In order to establish that each philosopher who wants to eat eventually can do so, we impose an upper bound `U` on the time allowed for eating, using a local clock variable `x` for each philosopher. We assume that the time needed for the semaphore operations can be ignored[3].

[3]A symbol ∪ in a location indicates that the location is

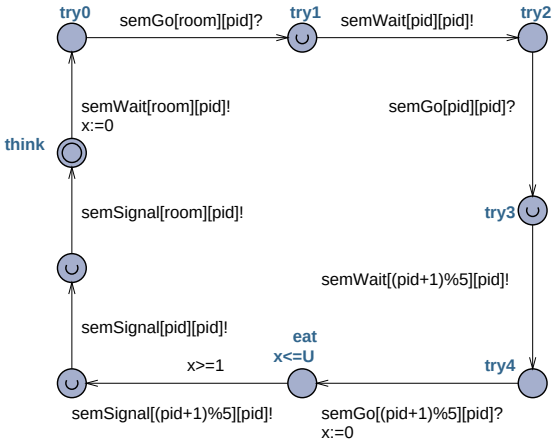

Figure 10: Model of dining philosopher in solution with doorman.

To exclude Zeno cycles[4], we also impose a lowerbound on the time needed to eat. With these assumptions, UPPAAL can prove that whenever a philosopher enters the location `try0` it will always eventually reach the location `eat`:

$$\texttt{Philosopher0.try0} \rightsquigarrow \texttt{Philosopher0.eat}.$$

In fact, we can establish an upper bound of $5 * U$ on the waiting time for a philosopher. The property

$$\texttt{A[]}\ \texttt{Philosopher0.try4 imply Philosopher0.x} <= B$$

holds for $B = 5 * U$ but not for $B = 5 * U - 1$. Since clock `x` is reset upon entering location `try0`, this means that a philosopher may have to wait in `try0`-`try4` for at most $5 * U$ time units before being allowed to enter location `eat`. UPPAAL proves the upper bound $5 * U$ in a fraction of a second, and only needs about 2 seconds to find the 62 step counterexample for $5 * U - 1$. Proving the upper bound by hand is hard and way too difficult for the large majority of Computer Science students.

Adding clock variables and timing constraints to the model requires some effort. Advocates of temporal logic may argue that it is much simpler to establish liveness properties with a model checker that supports general temporal logic such as SPIN [11] and SMV [4, 3]. However, if you happen to be a philosopher, knowing that you will be allowed to eat "eventually" is only of theoretical interest! Knowledge of the time bound $5 * U$ is useful in practice.

With the above model as a starting point, students may explore further properties. Does it make any difference if we add nondeterminism to the model and allow philosophers to pick up forks in any order? What is the maximal number of philosophers that can eat at any point in time? What happens if we change the number of philosophers? What happens if we no longer ignore the time needed to pick up forks? Etc.

"urgent" and no time may pass if the automaton is in this location.

[4]Infinite sequences of transitions in which time does not advance beyond a certain point.

3.3 The Room Party Problem

A particularly difficult synchronization problem, known as the "room party problem", has been defined by Downey in his "Little Book of Semaphores" (cf. [8, 9]). The situational sketch is as follows:

> A dean of students should keep order in the students' house. In order to do this, he can enter a room with too many students (in order to break up a too large party) or he can enter an empty room to conduct a search. Otherwise, the dean may not enter a room. If the dean is in a room, no additional students may enter, but students may leave. In that case, the dean has to stay until all students have left. There is only one dean, and no limitation on the number of students in one room. The challenge is to construct code for dean and students such that these constraints are satisfied.

A first solution of Downey, published in [8], is displayed in Table 1. It employs a `mutex` to protect the variables `students` and `dean`, which denote the number of students in a room and the state of the dean, respectively. The other

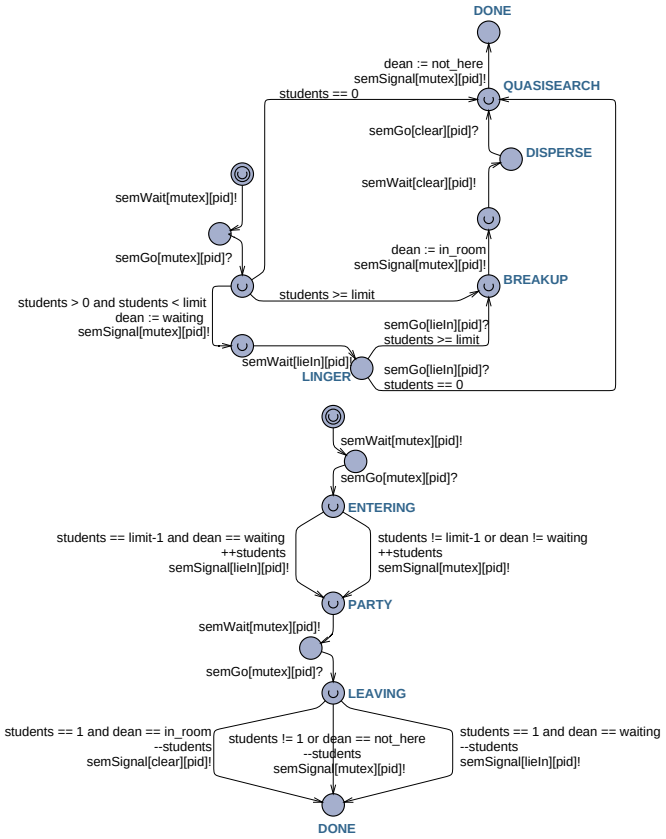

Figure 11: Model of first solution to room party problem.

two semaphores `clear` and `lieIn` are used as rendez-vouses between a student and the dean. The model for Downey's

dean code:

```
mutex.wait()
  if students > 0 and students < 50:
    dean = 'waiting'
    mutex.signal()
    lieIn.wait() # and get mutex
  # students must be 0 or >= 50
  if students >= 50:
    dean = 'in room'
    breakup()
    mutex.signal()
    clear.wait() # and get mutex
  else: # students = 0
    search()
dean = 'not here'
mutex.signal()
```

student code:

```
mutex.wait()
  students += 1
  if students == 50 and dean == 'waiting':
    lieIn.signal() # and pass mutex
  else:
    mutex.signal()
party()
mutex.wait()
  students -= 1
  if students == 0 and dean == 'waiting':
    lieIn.signal() # and pass mutex
  elif students == 0 and dean == 'in room':
    clear.signal() # and pass mutex
  else:
    mutex.signal()
```

Table 1: First solution of Downey to the room party problem.

dean code:

```
mutex.wait()
  if students > 0 and students < 50:
    dean = 'waiting'
    mutex.signal()
    lieIn.wait() # and get mutex
  # students must be 0 or >= 50
  if students >= 50:
    dean = 'in room'
    breakup()
    turn.wait() # lock turnstile
    mutex.signal()
    clear.wait() # and get mutex
    turn.signal() # unlock turnstile
  else:  # students = 0
    search()
dean = 'not here'
mutex.signal()
```

student code:

```
mutex.wait()
  if dean == 'in room':
    mutex.signal()
    turn.wait()
    turn.signal()
    mutex.wait()
  students += 1
  if students == 50 && dean == 'waiting':
    lieIn.signal() # and pass mutex
  else:
    mutex.signal()
party()
mutex.wait()
  students -= 1
  if students == 0 && dean == 'waiting':
    lieIn.signal() # and pass mutex
  elif students == 0 && dean == 'in room':
    clear.signal() # and pass mutex
  else:
    mutex.signal()
```

Table 2: Second solution of Downey to room party problem.

first solution is given in Figure 11. It turns out that this solution does not prevent students from entering the room when the dean is there to break up a party. Analysis with UPPAAL reveals a trace of some 20 steps that shows how the dean has to release the `mutex` after breaking up the party without being able to prevent students to enter the room. This bug was actually found by student Marc Schoolderman during his assignment work as a part of his course on operating systems. He also proposed an alternative solution and showed this obeyed the required properties with the aid of UPPAAL. A discussion with the author however resulted in yet another proposal from Downey's side, published in [9]. A turnstile `turn` is added to the code in the Table 2, specifically designed to keep students from entering while the dean is in the room.

But, alas, also this model does not satisfy the required property mentioned above. A trace (of 64 steps in this case) shows that one student may have received and released the turnstile to enter, but still is waiting for the mutex, which he gets from the dean while the latter is still in the room. Such counterexamples are hard to find just by looking at the code. The UPPAAL model with which this analysis was done, is shown in Figure 12. Note that the structure of the code of Downey is very well visible in the UPPAAL model.

Students participating in our course discovered several other mistakes in [8], simply by modeling and analyzing proposed solutions from the book using UPPAAL. The author uses semaphores in a very structured manner, using solutions for basis synchronization patterns, and we do not think that these problems could easily have been avoided using different synchronization primitives. Our conclusion is that the intrinsic complexity of these synchronization problems requires the use of model checkers to ensure correctness.

4. MONITORS

The monitor was introduced in the 70s by Hoare [10] as an alternative programming-language construct that provides equivalent functionality to that of semaphores but is easier to control. A number of programming languages, such

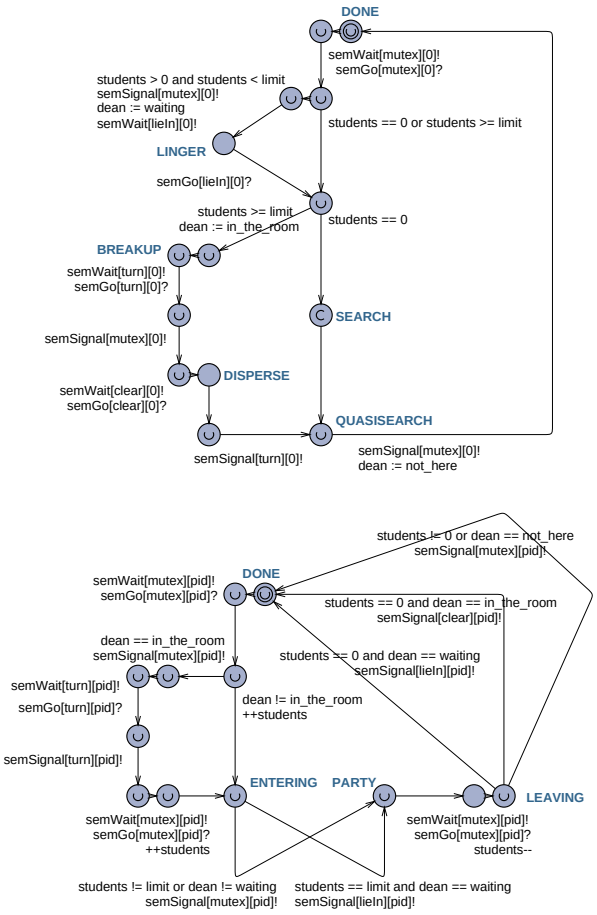

Figure 12: Model of second solution to room party problem.

as concurrent Pascal and Java, have implemented monitors. The basic version of Hoare was refined by Lampson and Redell [14] in the 80s. A monitor is a software module that consists of a number of procedures, some initialization and local data. Processes can enter the monitor by invoking one of the procedures, while only one process may be executing in the monitor at any time. Other processes that have invoked the monitor are blocked until the monitor becomes available. We describe here a UPPAAL model for a restricted (structured) usage of the Lampson and Redell monitor in which each procedure has the following structure:[5]

```
return_structure procedure(invoke_variables)
{
  while condition(this_procedure)
              then wait(my_condition);
  execute procedure;
  update conditional variables;
  notify all conditions;
}
```

[5]General monitors can also be modelled in UPPAAL, for instance by explicitly describing an implementation using semaphores.

These restrictions render the monitor more robust against missing events and make the procedures more independent of each other, because they do not have to know which conditions to trigger precisely. It comes at the cost of more iterations, but their number is manageable (cf. [14]).

We have modeled a monitor `id` as shown in Figure 13. We assume a set `PID` of process identifiers and a set `METH` of method names. The conditions on the procedures and their updates appear in the UPPAAL template as two model dependent functions `condEval` and `condUpdate`. There are two possible events from the central "standby" location. The first is the reception of a monitor invocation `monitorCall`$[\mathtt{id}][\mathtt{p}][\mathtt{m}]?$, for some $\mathtt{p} \in \mathtt{PID}$ and $\mathtt{m} \in \mathtt{METH}$, which puts the pair $(\mathtt{p}, \mathtt{m})$ at the end of the queue to be handled, or jumps to an overflow location if the queue is full. The other event is the execution of a procedure call, which is enabled by `allCondEval`(). This function checks if `condEval`$(\mathtt{p}, \mathtt{m})$ evaluates to true for some pair $(\mathtt{p}, \mathtt{m})$ in the queue. If so the corresponding transition is taken urgently[6]: the first pair $(\mathtt{q}, \mathtt{u})$ in the queue that is enabled, is removed from the queue, executed, and the corresponding conditional variables are updated through `condUpdate`$(\mathtt{q}, \mathtt{u})$. The closing notification statement of the Lampson and Redell monitor is taken into account by the evaluation of `condEval`() each time the central location is entered.

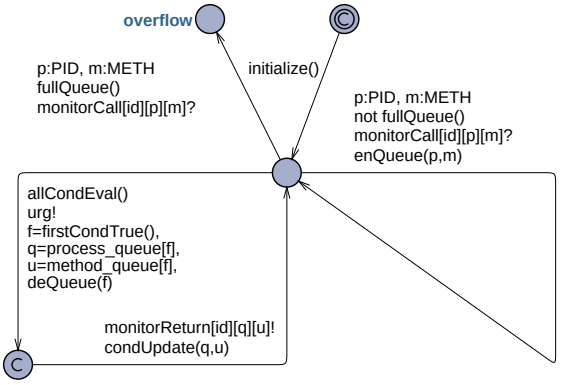

Figure 13: Model of a monitor.

4.1 Dining Philosophers Revisited

Several textbooks present a solution to the classical dining philosophers problem with a monitor, for instance Stallings [20], Nutt [17] and Silberschatz & Galvin [19]. In the last two books, the presented solution involves a test procedure that is not side-effect free, an objectionable way of programming by itself, cf. Figure 9.11 at page 230 of [17]. Moreover, both mention that the solution is deadlock free, but not starvation free, and leave finding the solution to the latter problem as an exercise to the reader.

In Figure 14 the philosopher part of the model is shown. Table 3 shows the model dependent code in the monitor template. Here `N` denotes the number of philosophers. This is similar to Nutt's solution, but the condition test has been made side-effect free, while the notifications are automatic

[6]In UPPAAL, we declare `urg!` to be an urgent broadcast channel to which no-one listens.

```
meta int eaters = 0;
const int thinking = 0, eating = 1;
int status[N];

bool test(PID p) {
  return ( (status[(p+N-1)%N] != eating) && (status[(p+N+1)%N] != eating) );
}

bool condEval(PID p, METH m) {
  if ( m==pickUpForks ) { return test(p); }
  if ( m==putDownForks ) { return true; }
}

void condUpdate(PID p, METH m) {
  if ( m==pickUpForks ) { status[p] = eating; eaters++; }
  if ( m==putDownForks ) { eaters--; status[p] = thinking; }
}
```

Table 3: Model dependent code in model of dining philosopher.

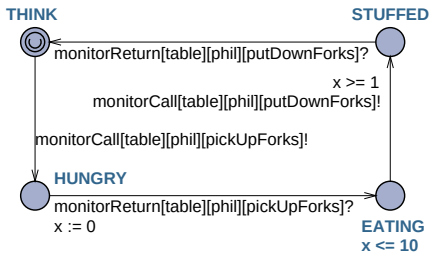

Figure 14: Template of philosopher in the model of Nutt's solution with a monitor.

by the return to the central state in the monitor template. The query

$$\texttt{Philosopher(0).HUNGRY} - - > \texttt{Philosopher(0).EATING}$$

indicates a trace to the starvation problem immediately.

A possible solution to the starvation problem involves the introduction of a doorman, as explained for instance by Downey [8] in terms of semaphores. The model is easily extended as shown in Figure 15 and the extension of the code in Table 4. The liveness property (absence of philosopher starvation) is readily checked with UPPAAL.

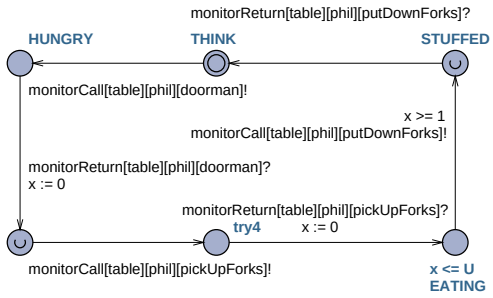

Figure 15: Template of a philosopher with the introduction of a doorman in the solution.

5. CONCLUSIONS AND RELATED WORK

Lamport [13] asks: "Programs are not released without being tested; why should algorithms be published without being model checked?" Similarly, we conclude "Why should algorithms be explained without the use of a model checker?" As discussed in this article, key advantages of using model checkers are: (a) unambiguous definition of algorithms and their properties, (b) visualization of concurrent behavior, and (c) fully automatic proof of correctness properties. Model checking technology has become easy to use and sufficiently powerful to handle nontrivial instances of all the concurrent algorithms that are typically discussed in introductory courses on operating systems or concurrent programming. The behavior of these algorithms is tricky, and authors, instructors and students should simply not trust solutions that have not been model checked. However, we emphasize that key elements for successful use of a model checker with first-year students are (a) the availability of a powerful graphical user interface for editing and simulation, and (b) a smooth and short learning curve. Here UPPAAL clearly stands out.

Mutual exclusion algorithms are popular benchmark examples for model checkers, see for instance [3], and the analysis results of this article are not new, except for the time bound for Peterson's algorithm. Our results on model checking semaphores and monitors are new, to the best of our knowledge. In this article we have not described UPPAAL models of the use of message passing as a synchronization primitive; adding this would be routine.

Closely related to our work is the book of Magee and Kramer [16]. This book provides a nice approach to concurrent programming using state models and Java. State models are described in a textual, process algebraic language called FSP and can be visualized and analyzed using an LTL model checker called LTSA. The consistent combination of state models and Java makes their approach ideal for a course on concurrent programming. Via the use of Java applets, the authors offer appealing visualizations of concurrent behavior, in addition to the visualization of state machines offered by LTSA. The FSP language, however, is much less expressive than the UPPAAL language, and for instance does not

```
bool condEval(PID p, METH m) {
  if ( m==pickUpForks ) { return ( test(p) ); }
  if ( m==putDownForks ) { return true; }
  if ( m==doorman ) { return ( eaters < N-1 ); }
}

void condUpdate(PID p, METH m) {
  if ( m==pickUpForks ) { status[p] = eating; }
  if ( m==putDownForks ) { eaters--; status[p] = thinking; }
  if ( m==doorman ) { eaters++; }
}
```

Table 4: Model dependent code in model for dining philosopher with doorman.

support shared variables. This makes it less straightforward to handle mutual exclusion algorithms, like we did in Section 2. Also, the EFSM graphical notation of UPPAAL even allows one to visualize the behavior of complex industrial sized models, whereas only relatively small models can be visualized using LTSA. Magee and Kramer [16] present a model of semaphores which, in our opinion, is overly abstract: a wait operation is modeled by a single transition (rather than with a pair of a `semWait` and `semGo` transition) and information about the order in which processes have been blocked is not preserved. Typically, liveness and real-time properties of concurrent algorithms crucially depend on the order in which processes that are blocked on a semaphore are activated again. Implementations usually adopt a FIFO order. This means that in the approach of Magee and Kramer [16] it is, for instance, not possible to prove liveness or real-time properties for the solution of the dining philosophers with a doorman, like the $5 * U$ bound we derived in Section 3.2.

As a spin-off, using model checkers in an introductory course also provides a great opportunity to increase the impact of formal methods research. More students will learn about and appreciate model checking technology. Once students have seen how useful these tools are, they will much faster decide to use them later on when facing similar problems. The more theoretically inclined students become motivated to study the algorithms behind model checkers.

6. ACKNOWLEDGMENTS

The second author has been supported by NWO/EW project 612.000.103 on Fault-tolerant Real-time Algorithms Analyzed Incrementally (FRAAI). We like to thank our students for their enthousiasm and help with constructing models, in particular Justus Freijzer, Martijn Hendriks, Bart Kerkhoff, Bart Meulenbroeks, Marc Schoolderman and Koen Vermeer. We also thank the anonymous referee for some comments that helped us to improve the presentation.

7. REFERENCES

[1] G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.

[2] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings QEST*, pages 125–126. IEEE CS, 2006.

[3] N. Bogunovic and E. Pek. Verification of mutual exclusion algorithms with SMV system. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8. Vol 2*, pages 21–25. IEEE CS, 2003.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[5] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*. Academic Press, 1968.

[6] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[7] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

[8] A. Downey. *The Little Book of Semaphores*. Green Tea Press, second edition, 2005.

[9] A. Downey. *The Little Book of Semaphores*. Green Tea Press, 2.1.2 edition, 2007.

[10] C. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[11] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2004.

[12] H. Hyman. Comments on a problem in concurrent programming control. *Commun. ACM*, 9(1):45, 1966.

[13] L. Lamport. Checking a multithreaded algorithm with $^{+}$cal. In *Proceedings DISC 2006*, LNCS 4167, pages 151–163. Springer, 2006.

[14] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. In *Proceedings SOSP*, pages 43–44, 1979.

[15] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Fransisco, CA, 1996.

[16] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley, 2006.

[17] G. Nutt. *Operating Systems: A Modern Perspective*. Addison Wesley Longman, 2nd ed, 2000.

[18] G. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[19] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison Wesley Longman, 5th ed, 1997.

[20] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, 5th ed, 2005.

[21] H. Sutter. The free lunch is over — a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[22] A. Tanenbaum and A. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd ed, 1997.