LEARNING PROCESS SYNCHRONIZATION INTERACTIVELY

Henri Delebecque

LEARNING PROCESS SYNCHRONIZATION INTERACTIVELY

Henri DELEBECQUE

Supélec, Gif sur Yvette (FRANCE)
henri.delebecque@supelec.fr

Abstract

Teaching Computer Systems Architecture at Supélec involves both lectures and projects. We have defined and implemented a pedagogical simulator intented to teach different synchronization problems. Its main aim is to help students better understand the problems involved in variable sharing between two processes, and allow them to solve the problems, using semaphores. The simulator offers an instruction library based on a pedagogical processor used in the Computer Systems Architecture course. Students will drag-and-drop the instructions into the two instruction lists used for the processes. The overall program should fulfil the global requirements, and produce the right value for the shared variable, regardless of the execution order of the instructions. Students use our code analysis software to check the correctness of their design. If the algorithm detects a failure in the shared variable use, it builds an execution flow that exemplifies the failure. The students can follow the flow, step-by-step, by browsing through a series of on-the-fly computed panels, each one displaying the critical values of the processor registers as well as process description record fields and the current instruction field or a shared variable. The explanation panel includes arrows that point to every information transfer and a short text that describes the current overall status. The analysis and explanation builder software has been designed according to a very simple instruction set of our pedagogical computer. Its principal feature is to reduce the complexity of the student code diagnosis algorithm. The simulator is currently accessible online to our 400 students who have given us a very positive feedback on it. We have now been asked to extend its application to other problems. The possibility of such an extension has been included in the simulator pre-requirements, first by offering an open instruction library, second by designing an analysis tool able to work either with various sets of instruction sequences, or with more than two processes, multiple semaphores and/or shared variables. The most innovative aspect of our pedagogical experience is the ability to explain very precisely the faulty behavior of the code to our students while allowing them to build any sequence of instructions without the burden of conforming to a strict programming language syntax. This allows them to focus on the problem they have to solve, and probably explains why they like it so much. Furthermore, our simulator is expandable to other typical synchronization problems or, potentially, to other instruction sets.

Keywords - Pedagogical simulator, computer science, process synchronization.

1 INTRODUCTION

Concurrent programming is difficult both to teach and to learn. Even in simple cases, it involves understanding many concepts, such as the semantics of a processor instruction, the various states of a process, the way in which an operating system describes a process, etc. Moreover, practical experimentation is difficult, due to the non determinism induced by the scheduler.

We will start by describing how Computer Science is taught at Supélec, then we will define the pedagogical aims of our project. This will later on allow us to describe the models and concepts involved. The next paragraph deals with the simulator design, and the last one with related work. We will end with the future developments.

2 PEDAGOGICAL CONTEXT

"Computer Science" is a two-year compulsory course for all students at Supélec, and includes both lectures and practical sessions. The first year courses teach the basic concepts of software

engineering and the programming models. Students become familiar with the usual topics of software development and learn Java as a high level programming language.

One of the two second year courses enables students to master both the fundamental blocks of hardware computer architecture, namely busses, memory chips and modules, interruptions as well as the operating system concepts, such as a process, a thread, a scheduler, memory and input/output management. All students finishing their second year at Supélec have to pass an exhaustive examination in the hardware and operating system domains, during two lab sessions. During the first session they are asked to build a simple computer using a pedagogical hardware simulator we have designed[1]. In the second one they have to design a multi-threaded simulation in Java in which they have to carefully synchronize the threads in order to correctly manage the shared global variables.

The concepts involved in this latter session are multiple, complex, and completely new to our students. Moreover, the global behavior of a set of threads is obviously non-deterministic, since it depends on their scheduling which can change from one execution to another.

The debugging of a set of concurrent threads is considered as a tricky task, since inserting textual outputs (the means often used by students to pinpoint the successive values of a variable) leads to unreliable results in a multi-thread context. Moreover, using breakpoints included in the current IDE (Integrated Development Environment) debugging tools can change the faulty behavior or completely remove errors.

For all these reasons, we have decided to help our students in two ways. First, a specific tutorial proposes a design of a consumer-producer pair of processes, with requirements of an increasing difficulty. Next, we have designed an interactive pedagogical simulator that allows them to solve a rather trivial problem of sharing a variable between two processes, one raises the initial variable value by a constant, while the other, concurrently, decreases the same variable from which is subtracted another constant This article will describe the simulator, both from the pedagogical, ergonomic and technical point of view.

During their third and last year, all students have to specialize in one of fourteen majors, each in a particular field (e. g. communications systems, power electronics, or computer science). Students who choose to major in computer science will have to master the main fields of modern computer systems, attending further theoretical lectures and doing lab work.

3 PEDAGOGICAL AIMS

We have designed the pedagogical material for the Computer Science courses with the five following pedagogical goals in mind. The first three of them represent general pedagogical principles in our school of engineering, while the last two are specific to the Computer Architecture domain.

3.1 Students should focus on the concepts

One of the main purposes of the simulator is to make students able to design and rectify the instruction sequences of the two processes by themselves, since they have to use it outside the mandatory sessions. The Human-Machine interface should be intuitive, and the simulator has to give visual feedback of the students actions, while allowing them to focus on the concepts they should learn. This option obliges the designers to avoid tools that ask students to give instructions textually, as the editing task requires too much time and effort if the students follow strictly a programming language syntax.

3.2 We have to keep students in a stable pedagogical background

To help students in their design, we have chosen to keep for the simulator the same instruction set computer they have seen in the computer architecture lectures that start the Computer System Architecture course. Obviously, we have added WAIT and SIGNAL, i.e. the instructions required for manipulating semaphores. The synchronization tool which executes such instructions has been described by DJIKSTRA[2].

The semaphore is the main synchronization mechanism described in lectures, practically used during a tutorial devoted to process synchronization problems as well as during the hands-on session described in paragraph two and detailed later. The lecturer presents other techniques, but focuses on semaphores in order to shorten the time spent on this subject.

3.3 The simulator should be accessible online

Since specifying and developing such a simulator following the pedagogical and technical requirements described here is time-consuming, we have decided to use it for all second-year students, regardless of the campus they belong to (Gif-sur-Yvette, near Paris, Rennes in Brittany, or Metz in the east of France). This option has obliged the designers to define an online tool hosted by “Moodle”, the pedagogical platform chosen by the school. To implement the tool, the development team has chosen Java, since it is the language widely mastered in the school. The simulator is an applet, as this option requires only a lightweight client and allows trivial updates. The disadvantage of using a Java applet, in terms of security constraints and limited processing resources are of less weight in our case since the simulator neither needs to access to the computer's client resources, nor implies heavy computing.

3.4 The simulator should be able to manage a large panel of problems

To achieve the same aim of economy, we specify a very open and expandable simulator. Instead of limiting its diagnosis capabilities to the initial exercise, for which crude methods are sufficient, we have chosen to design an analysis algorithm, reusable for forthcoming complex problems. Conversely, after studying many open source tools that can help us in diagnosing the student proposal, we have chosen to design a custom analysis tool. We justify this option by analyzing the available tools.

First, three kinds of tools exist: the static ones that work before (and without) code execution ([3], [9]), tools that detect the race conditions during execution ([5],[6]), and the post-mortem ones that analyze the log trace of execution to diagnose problems ([7], [8]). The last two techniques must either exhaustively investigate all the execution paths, or accept that some errors remain undetected. The first option will lead to heavy computing, even in the case of simple programs (two 10 lines long programs may generate about 1 million different execution paths). It is irrelevant if the process codes contain infinite loops, a very common case in synchronization problems. The second option is forbidden in a pedagogical context, where students need a perfect diagnosis of their solutions.

What remains is the static analysis option. One kind of static diagnosis tool uses high level languages, such as C or Java, extending their type system ([9], [10], to prevent data races. They are of less interest in our case, since the very simple programming model our students use in Computer Systems Architecture, based on processor instructions, differs considerably from the semantics of these languages, and does not allow for a typing system. Moreover, these tools try to prevent problems by adding locks. The objective is for the simulator to detect rather than remove errors contained in the students’s work. Other kinds of tools performing effective detection ([12], [13]) cannot be used for our simulator, since they only supply a diagnosis, but do not give a typical sequence of an instruction execution that leads to the failure. Moreover some of them tend to report "false warnings", probably due to the fact that they tend to analyze high-level languages which manipulate pointers. Manageable in the industrial systems domain where engineers can deal with it, a wrong detection is not acceptable in a pedagogical context.

3.5 The simulator has to explain pedagogically the faulty behavior of the code

As mentioned before, the most crucial goal of our simulator is to carefully explain to our students the mistakes they have made, especially since they work alone. The simulator builds this explanation using the terms and concepts used during lectures and tutorials, and displays all the information required for understanding the faulty behavior of the code.

Moreover, the students are able to browse through the sequence of slides displaying the following states of the system while the processor executes their specific code, either forward or backward, at their own rate. Each slide shows the processor register values (its state), the processor state stored by blocked or waiting processes, the value of any shared variable and the state of each process. A process can be “running” (if the processor is executing its code), “waiting” (it is waiting to run), “blocked” (if it waits is waiting for a SIGNAL to be executed by another process) or “terminated”, as shown in Fig. 1.

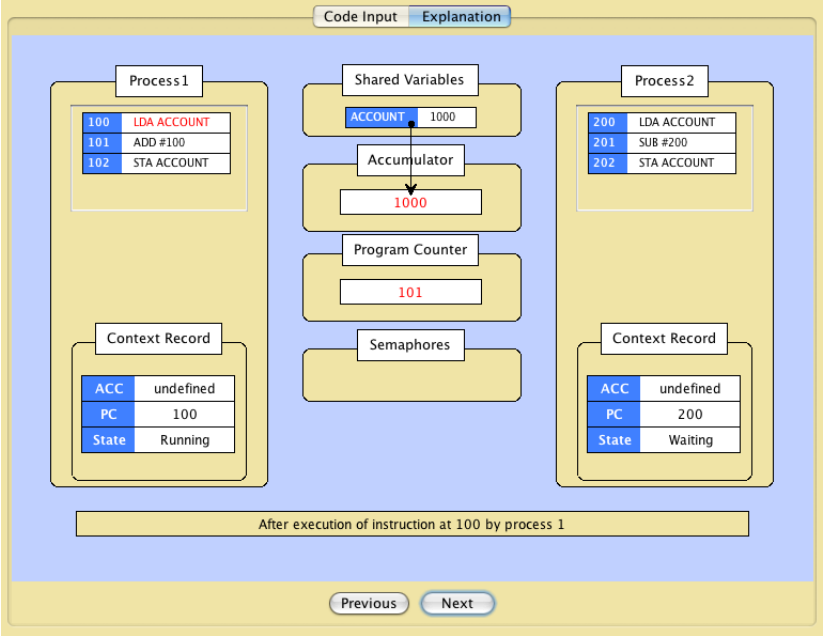

Figure 1. A step of the execution

4 THE CONCEPTS AND MODELS INVOLVED

The Computer System Architecture course takes only twelve lectures, four tutorials of 1.5 hour each, and two 8-hour practical sessions to teach all the building blocks of a modern computer system. It uses a custom pedagogical processor, built step-by-step in the lectures devoted to computer architecture. More than 20 years ago, Supélec professor decided to define their custom processor rather than to use the well-known Tanenbaum architecture [14] since the Supélec processor allows the lecturer to reduce the time spent on explaining both the external and internal processor architectures.

Another advantage of this simple processor is that our students can design, built and test it in the first practical session mentioned earlier. In line with the pedagogical principles explained above, we use the same processor for all the tutorials, the lectures devoted to the acceleration mechanisms (cache memory and pipeline) and the presentation of the process synchronization fundamentals.

These fundamentals start from the very simple case of a variable shared between two processes, and describes how the lack of synchronization can lead to a data race giving a wrong final value of the shared variable. The lecturer then presents classic patterns, such as rendezvous, both incomplete or complete, between processes. Finally the lecturer describes the problems that occur when synchronization tools are misused, and the techniques and principles that allow students to avoid committing errors.

4.1 The processor

Our processor is equipped with an accumulator, eight instructions (LDA for loading the accumulator, STA for storing it into a variable, ADD and SUB for adding a value to it, or subtracting a value from it, and branching instructions, that allow the processor to jump to another part of the code). It defines only two addressing modes (direct for variables and immediate for constants). It is built using very simple building blocks, presented in the Logic System lectures, such as 16 bits wide registers, a simple arithmetic and logic unit and a 64K 16 bits wide memory, connected together using tri-state buses. A 3 hour lecture starts by describing these blocks and specifying the processor requirements. The lecturer then builds step-by-step the internal data path, and ends by defining the sequencer operations flowchart. An additional tutorial asks students to apply this theoretical knowledge while building another kind of processor, to allow them to separate the general concepts found in any processor from the specific characteristics of a particular one.

4.2 DJIKSTRA's Semaphores

The semaphore is the main locking mechanism described in lectures, used intensively during the tutorial devoted to synchronization problems, and the practical 8-hour session that asks student to design, build and test a barbershop simulation in Java.

The tutorial focuses on the analysis and design of two processes, one producing messages it puts into a mailing box, while the second extracts and processes them. The first process should block itself as soon as the mailbox is full, while the other is not allowed to extract a message from an empty box. These requirements describe a classic case of process cooperation, called the producer-consumer problem. We then enlarge the initial specification first by allowing multiple producers and consumers, then by asking an idle consumer process to process an alternative task instead of leaving the process blocked.

5 THE SIMULATOR DESIGN

To fulfil the requirements described in the paragraph 2, we have specified the simulator in terms of three dimensions: the pedagogical, the ergonomic, and the technical one.

5.1 Pedagogical aspects

During the Computer Systems Architecture our students have to assimilate a lot of new concepts in a rather short time. We present these concepts by increasing complexity, the understanding of the last one implying that the earlier ones have been well assimilated. Students who use the simulator work on a voluntary basis without coaching. The simulator should help them as much as possible, either during the process code building, or the diagnosis explanation.

5.1.1 Contextual and immediate help

The simulator offers ready-to-use instructions, stored in an easy to browse panel, instead of asking the student to enter them manually, as shown in Fig. 2.

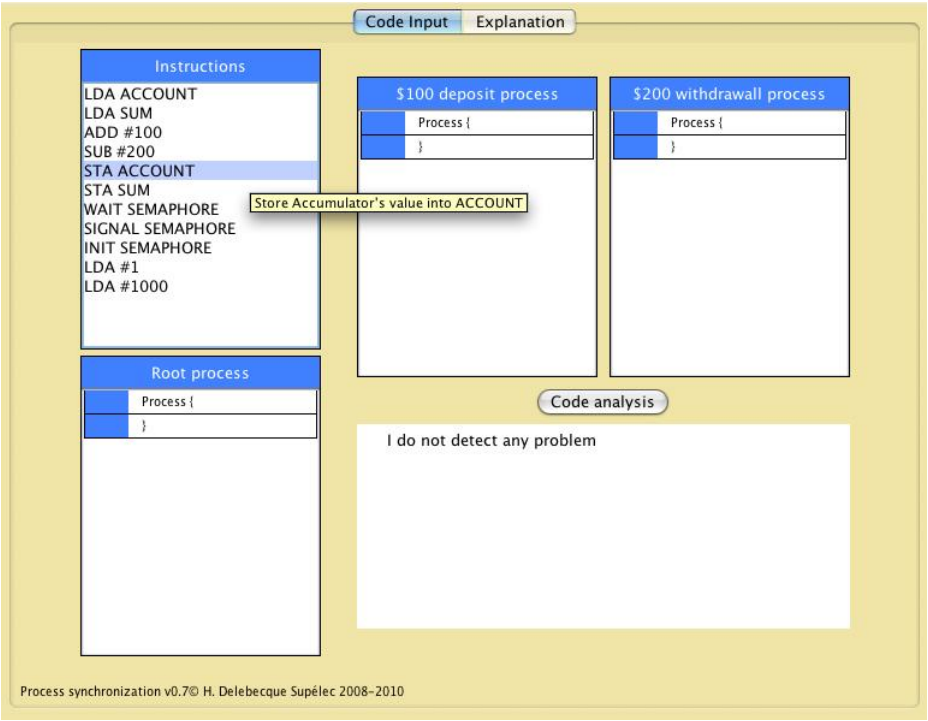

Figure 2. The processes code input panel

The panel guides students in their design, even if they do not remember the computer instruction set. Practically, the simulator displays the semantics of any instruction, if the student leaves the mouse

motionless on a library instruction. During the explanation phase, the simulator exemplifies the impact of an instruction execution by displaying the instruction just executed and the values it has modified in red, and by drawing arrows from the origin of the value to the object that received it, as shown in Fig. 3.

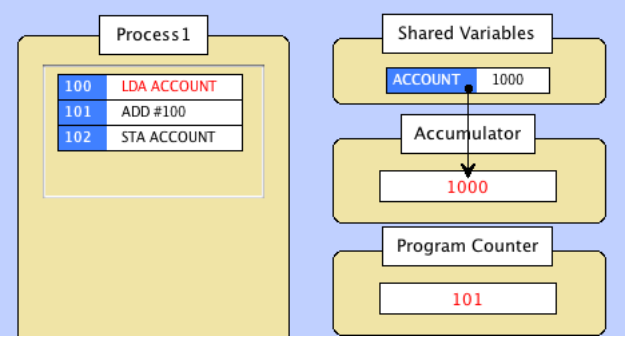

Figure 3. Clear explanation of an instruction execution

The students do not have the burden of looking for the semantics of an instruction, and can focus on the task they should perform: correctly synchronizing the two processes, by understanding where and why their proposal has failed.

5.1.2 Explanation of the scheduling consequences

We also need to prove to our students why a specific arbitrary process scheduling can lead to a faulty behavior by giving a global state description, as shown in Fig. 4.

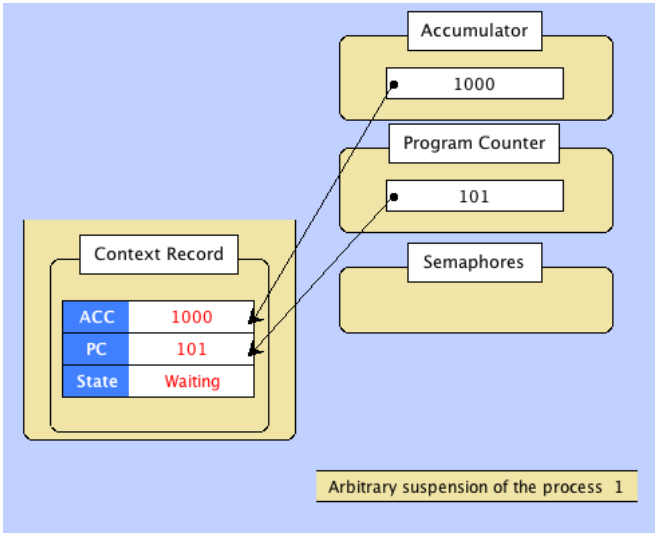

Figure 4. Impact of a process state change

This description clearly explains the cause of any process state change. For example, the simulator tells the students if the process is “blocked” by performing a wait on a semaphore having a counter valued less or equal to zero, if it “terminates”, or if it turns “waiting” due to the scheduling policy.

In the Fig. 4 example, the explanation panel both mentions an arbitrary suspension (decided by the diagnosis algorithm to show a faulty code behavior) and stresses the transfer of values this process state change implies. Displaying this information is a crucial point for the good understanding of the race condition. More precisely, the faulty behavior of the code is due to the fact that two different accumulator values exist: on the one hand the real one and on the other the value stored in the context record, restored into the accumulator as soon as the process resumes. This is the reason why the explanation computing phase should carefully chose which process should start first and when to suspend it. We found here a clear justification for avoiding tools that give only a global diagnosis, without an execution path leading to the race condition

.

5.2 Ergonomic aspects

Since our students may have to change the code they proposed after having understood their mistake, and in order to offer the quickest way to build the first version, we have decided to allow them to drag-and-drop the instruction from the library into one of three instruction lists. The first one (in the right bottom corner of Fig. 2) contains the initialization code of the root process. The other two receive the instructions that have to add $200 to (and withdraw $100 from) the shared variable that implements the bank account. For the same reason, students can move an instruction in any instruction lists, either by drag-and-dropping it, or by combining keyboard keys. They can also remove an instruction by selecting it and pressing the delete key.

The applet avoids modal dialogs that tend to keep the user captive, and allows students to navigate freely between the input and explanation panels. This gives them the ability to look at information not displayed on the explanation panel (such as the root process code, or instructions they have not yet used), while keeping the computed explanation slides sequence available for further study. To avoid mistakes, a short user's manual fills in the explanation panel as soon as the students change anything in their code.

5.3 Technical aspects

The specification and design of either the user interface, or the two algorithms required by the simulator took about 4 months of full-time work in order to build a version that meets only partially our requirements, but still can be put on-line.

The user interface specification of the input panel has been the first and easier step, while the explanation slides sequence design had to wait until the complete definition of the race-condition detection algorithm principles. As mentioned before, we first looked for existent tools and algorithms, before deciding to write our own. Once we chose this option, we gained more knowledge about the available information necessary to explain the student mistake. Consequently we are now able to specify how to compute the explanation slides sequence, and decide what information each slide will display.

The code analysis begins by looking for not initialized variables and semaphores. Any error of this kind shortens the diagnosis by explaining the problem. The race condition algorithm starts by detecting any code section that begins with a transfer from a variable value into the accumulator, modifies the accumulator, and then stores it in the same variable. We call these zones varZones. The algorithm then checks if such sections are included in zones beginning with a WAIT, and ending with a SIGNAL working on a same semaphore, we called semZones. The algorithm marks varZones outside any semZone as prone to race conditions. During this analysis, the algorithm manages semZones included in another semZone by storing them in its list, and by assigning them an incremented level.

All the zones detected during the first phase are used both to detect race conditions and to produce the sequence of explanation slides. The race condition detection algorithm begins by checking conflicts between the root semZone of the two processes. It checks the kind of semZone (either one starting with a WAIT or a SIGNAL), and also checks whether the first semZone excludes executing the other one until its own end or not. In the latter case, it first asks the other semZone to check for conflicts with its included semZones (in a recursive way), then with its included varZones. Checking varZone conflicts works by comparing the variable affected by both zones. Identical variables lead to race condition which inputs for the building phase of the explanation slides

.

6 RELATED WORK

The earlier 3.4 paragraph mentions many tools devoted to the detection and/or correction of race conditions in concurrent programs. The aim of the current paragraph is to investigate pedagogical tools dedicated to the teaching of concurrent programming. The first one, Visual OS, is an interactive, exploratory environment for learning process synchronization. It allows for visualizing classic synchronization cases (the critical section problem, a Consumer/Producer pair, a Reader/Writer pair, and the well-known dining philosophers problem). Unfortunately, this tool is described very briefly in only one article [15], and its development seems to have stopped since then. Moreover, another tool

with the same name exists, oriented towards a whole operating system graphical simulation, which is not usable for our purposes. Another project, Convit [16] is a Java applet like our simulator. It can be used to ensure easy maintenance and a lightweight set-up. It allows students to test their solutions to various problems. It provides means to debug, simulate and to change the source code, such as the global deadlocks detection. However, the teacher should create all examples and interactive problems, since this supposes compiling the concurrent program parts code into Java. The Convit programming language, even if it is simple, remains a high level language. It is a mix of Pascal and C. However, Convit lacks the explanation feature of our simulator, recognized by our students as very important in understanding why and where their code has failed.

7 CONCLUSION

More than 400 of our students have been using our pedagogical simulator for 2 years. They have given us a very positive feedback on it. They massively ask for its extension to more complex cases, such as consumer-producer pairs, or rendezvous. As mentioned earlier, this capacity is one of its key features, since it presents an open instruction library, and uses an analysis and explanation sequence building algorithm able to cope with more than one semaphore, or more than two processes. Moreover, we intend to upgrade the diagnosis algorithm to detect all cases of starvation (i.e. when all or part of a code cannot be executed due to a lack of a SIGNAL), and we intend to include deadlock detection. It can be also enhanced to work on higher level languages, as soon as these can be compiled into instructions of our pedagogical processor.

References

[1] Henri Delebecque. Hands-on Experience for Teaching Computer Architecture using a hardware Toolkit. EAEEIE Conference Ulm Germany 2000

[2] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM 8, 9 (1965), 569

[3] Dawson Engler & Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles. Pages 237-252, New-York, NY, USA 2003 ACM Press

[4] J. Choi, A. Loginov & V.Sarkar. Static data race analysis for multithread object-oriented programs, 2001

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro & T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. ACM Transactions on Computer Systems. 1997.

[6] C. Von Praun & Thomas Gross. Object-Race Detection. Conference on Object-Oriented Programming, Systems, Languages and Applications October 2001.

[7] M. Ronse & K. De Bosschere. RecPlay: A Fully Integrated Practical Record/replay System. ACM Transactions on Computer Systems, Vol.17 N°2, May 1999, Pages 132-133.

[8] J. D. Choi & H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. Proceedings of SIGMETRICS Symposium on Parallel and Distributed Tools Pages 48-49. August 1998.

[9] C. Boyapati & M. Rinard. A parameterized type system for race-free java programs. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications 2001. Pages 56-69.

[10] C. Boyapati, R. Lee & M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented programming, Systems, Languages and Applications. Pages 211-230. New-York USA 2002. ACM Press.

[11] A. Sasturkar, R. Agarwal, L. Wang & S. Stoller. Automated type-based analysis of data races and atomicity. Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel programming. Pages 83-94. New-York USA 2005 ACM Press.

[12] V. Kahlon, Y. Yang, S. Sankaranarayanan and A. Gupta. Fast and Accurate Static Data-Race Detection for Concurrent Programs. Computer Aided Verification 19th International Conference. July 3-7, 2007, Berlin, Germany

[13] F. Dabrowski and D. Pichardie. A Certified Data Race Analysis for a Java-like Language. Conference on Theorem Proving in Higher Order Logics August 17-20 2009 Munchen, Germany

[14] A. Tanenbaum. Structured Computer Organization (5th Edition). Prentice Hall 2005.

[15] A. Bhatti. Visual Tool for Teaching Synchronization in Operating Systems. SIGCSE Bulletin Vol 32. June 2000.

[16] H-M Järvinen, M. Tiusanen and A. Virtanen. Convit, a Tool for Learning Concurrent Programming. AACE E-Learn 2003 Conference