ROLE PLAYING IN CONCURRENT PROGRAMMING AS A WAY OF DEVELOPING ACTIVE LEARNING

Osvaldo Clúa[1] and María Feldgen[2]

Abstract *¾ Concurrent programming is not only a programming paradigm, but a powerful structuring tool for applications that are logically comprised of asynchronous components. Although it is a conceptually simple abstraction, concurrent programming is challenging because processes interact with themselves in several obscure ways. When students are caught in the synchronization traps, they tend to blame the compiler; the Operating System, the synchronization primitives, but they are not ready to discover their programming errors. We tried several methods of sequencing the material and guided problems without result; students failed to attack the concurrent issues with the necessary ingenuity. During the last term we tried a more integral "kinesthetic" way, with promissory results.*

Index Terms ¾ Active learning, Concurrent programming Software design, Teamwork.

INTRODUCTION

Concurrent Programming deals with defining programs whose actions may be performed simultaneously. Though the concurrent paradigm seems the natural way of modeling many target domains, it is by no means easy to use. The different actors, threads, processes or objects, interact with themselves in different and perhaps obscure ways. Most software design techniques yield sequential program structures. But concurrency is a very natural concept, and when explained seems to pose little difficulty to our students proficient in sequential programming. But when they put hands on the work, the unforeseen ways of interactions appear as program malfunctions. The non deterministic nature of actors interactions present themselves as random failures. Our students tend to blame the compiler, the operating system, the programming environment, everything but their design. In time we reached the conclusion that the problem was that students do not design their programs with concurrence in mind. In fact, they were not able to relate the process interactions that occur in concurrent programs to resource management and communication situations explained in the course. In this paper we present a brief history of our teaching experiences, how we reached this conclusion and the way we found to (hopefully) solve this problem.

TEACHING CONCURRENT PROGRAMMING

In the middle 80's Concurrent programming was introduced as part of a regular course called Computer Systems III, whose aim was to offer a suitable follow-on to the Operating Systems course. The course included items such as Data Base Machines, Real Time Data Acquisition, RISCS machines and others. It was developed as a number of non-formal concepts around the notions of *semaphores, Critical Sections* and *monitors,* in line with the Belfast Seminar at Queens University [1] and the classic Hansen's book [2]. It was targeted to Systems and Electronic Engineering major as an elective course. By 1996, a major revision of Engineering grades was made in our University, following in part ACM/IEEE *Computing Curricula 91*. It introduced some new policies such as early specialization in certain fields. As a result, Concurrent Programming began to be offered as a separate one semester elective course. The first attempts of implementation were made following the above mentioned concepts. There were programming assignments to be resolved in paper but we had no way of making hands on experience for our students due to the lack of adequate programming support structures as Computers. Being an Engineering course, we saw this semi formal approach frustrating for both, students and faculty.

By 1989 we decided to introduce some programming in the form of the use of "C" and semaphores. In those days there was only one UNIX machine in our Department, with 4 terminals and students had to wait in a long queue until they could get one. "C" details on semaphores are so toilsome, and technical literature at those times was so superficial that very few could really complete a programming assignment. The practical difficulties were so overwhelming that we could barely determine the result of the experience. The following years we left the programming assignment as an elective option and continued with the semi formal approach.

By 1991, Smallada, the work of Prof. Michael B. Feldman from the Department of Electrical Engineering and Computer Science of The George Washington University, Washington DC [3] became a usable option. It is a DOS and MAC based subset of Ada focused in concurrence. We adopted it and

[1] Osvaldo Clúa, Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, Argentina, oclua@ieee.org
[2] María Feldgen, Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, Argentina, mfeldgen@ieee.org

used it as the basis of our programming assignments with total success. Students were able to build concurrent programming and to play with different scheduling options [4]. After two semesters totally based on Ada we began to reintroduce some of the formal constructs, and we realized that though students could actually build concurrent programs, they did not understand the underlying concept. In 1996 we began to distribute among our students a GNU/LINUX version, installed on FAT and zipped to give them access to a working UNIX Systems. We replaced the formal primitives with C and the outcome did not change. They were able to build concurrent programs in Ada (GNAT) but they could not do the same assignment in C.

Our next trial was revisiting our first approach leaving Ada outside, we based our courses in the formal primitives, programmed them in "C " and enforced students to use them in the programming assignment. Then the "bug" showed in its whole dimension. Students do not design their solution, they just follow some sort of stepwise refinements from a working example through a working solution.

HOW STUDENTS WORKED OUT OUR PROGRAMMING ASSIGNMENT

Constructivists tell us that any new knowledge our students construct in response to new experiences, will be incorporated into the framework of knowledge they have already constructed [5]. This means that the learning we attempt to provide will be rooted on what students already know. Our students are proficient in sequential programming, so they incorporate concurrent programming primitives as *extensions of sequential programming statements*. They believe they understand the concept and they follow on with the next concurrent concept. When students are asked for a concurrent programming assignment, they solve it "sequentially", then they add the "concurrent stuff". And when things do not work, they rely on the tools the environment provides to *debug* their code. They do not attempt to make a concurrent design and they do not have the tools (previous experience) to do it. Indeed we do not know if the next time they will be asked for a concurrent assignment things will be done in a different way, curricula times do not allow us for this kind of post course checking.

In the light of this view, we could state by 1997 that students do not design concurrent programs, they debug their sequential programs in order to make concurrency work. A tool such as Ada is designed in order to make this approach work. Both, Ada 83 [6] and 95 [7] encourage an object based design using information hiding and abstract data types. Our students know both techniques since the time they approved CS2, so they felt self confident in this environments they follow the language rules, use the debugging tools to visualize and correct deadlocks and finally arrive at a solution. The solution often lacks of elegance of an appropriate design and looks like a pyramid of fixes. The clean design of Ada and the fact that our students really know data abstraction kept this fixes only in the concurrent part. Ada was designed supporting programming by extension, and this fact was proven by our students.

With the "C and primitives" approach the support of programming by extension was no more at hand. And mandatory the primitives prevented students from falling in the more obscure traps of "C". Actually we attempted to use raw "C+ IPC" but we found things such as a busy wait testing the value of a semaphore as it was a (shared) variable. Inspecting the code the fact that students were not designing but debugging their code became evident. When they were asked to tell us about their "developing cycle" their answers confirmed our observations. Of course, some preconceived notions such as "programming as a trial and error activity" [8] have a lot of influence in their attitude.

ADDRESSING THE CONCURRENCY CONCEPTUAL PROBLEM

As Ben Ari stated clearly in [5], we must ensure that a viable hierarchy of models is constructed and refined in the learning process. And we must deal with the sequential programming model they have. In 1998 and 1999 we tried different approaches in order to gain understanding of where the problem resides. In order to let them do some cooperative learning [9], during one course we asked for a programming assignment which would be used as the starting point of their final examination. Though there was some progress in the way they understood the behavior of their solution; the "debugging instead of designing" bug was still there. During the assignment we had to make specific programming demos in order to show them that compiler and libraries were working as intended. They were so self confident with their proficiency in sequential programming that they lacked the necessary ingenuity to challenge their design.

We also introduced a sequenced way of exposing the materials we teach. One of the common uses of sequencing refers to the ordering of curriculum from the more concrete topics to the more abstract, according to the work of Jean Piaget. It has been documented as improving student understanding [8] [10] and the syllabus remains essentially the same until today's courses. We begin our course with lectures devoted to a programming way of solving the mutual exclusion problem, followed by the hardware solutions of the problem in the same sequence as some Operating Systems classic books such as Tanenbaum or Silberschatz. After it we introduce file locking as a mutual exclusion technique. It is followed by Inter Process Communication primitives and then, with the adequate framework about concurrent processes, pipes, messages and semaphores are introduced. These issues are spiraled by solving some classical problems again and again using the different approaches. Spiraling allows students to see how new concepts are developed based on previously mastered ones [11]. Actually, we begin with the abstract primitives implemented in "C" and work our way down to the concrete "C" implementation when the primitive's input, output and effect are well understood. Ada tasking, threads and Java are left for a second advanced course.

We saw a real progress in the concurrency understanding, but there was little progress in the concurrent programming design. Of course, debugging was also improved, but it became apparent that analysis is not synthesis as stated by Bloom's knowledge levels [12].

FREEDOM FOR DESIGNING

In order to improve the primitives understanding, we began to think in a visualization tool. In [13] there is a survey of such tools. The authors classify them into *algorithm animation* and *program behavior visualization*. (The work is later in time than our survey, but its "previous work" section summarizes their findings in a very useful way). Our conclusion was that visualization tools will be used (again) as debugging tools, not too far from the above mentioned classification.

Stevens [14] in order to help the understanding of some "C" programming primitives, "converted" the primitives in command line commands. Cleverly used they allowed to see the effects of each one. We prepared such a set of command-lined primitives for our set of abstract semaphores and pipes primitives, and then an idea appeared. To use the primitives as designing tools and make use of the fact that the classroom is a concurrent system. We call this set an "inspecting and action" tool set.

The setup is straightforward: we make teams of students in a number according to the problem to solve (for example 5 dinning philosophers), they share a screen but each is in charge of an X-Window. They have the command line primitives and other command lines in order to inspect the values of semaphores and shared areas, and a "personal" window that can be used for inspecting. They have to behave as a process cooperating with the other processes.

They are given a few assignments or "games". Designing a solution for the problem is, of course, the overall goal, but they must al.so try to make the algorithm fail by using different time sequences, they are also asked to defeat their partners algorithm. The algorithm is played by following some written rules they have to design, and issuing the primitives on each window. When a process is blocked, its window has no prompt, so its owner cannot make his/her following move until he/she has the prompt again. Using some tools as *expect's (x)kibitz* [15] the students and instructors can follow the action from another terminal. The combination of structured primitives with well defined effects and the freedom of choosing which action can follow, is the right amount of tools and freedom to make them design the solution. The concurrency concept was clearly used from the beginning because of the number of players.

THE ROLE PLAYING ASSIGNMENT AT WORK

A typical class with a designing assignment works as follows. In previous meeting students get lectures on the semaphore and shared memory primitives, and on how to use the programming and inspecting tools. They have previous background on *shell programming* and on the use of *expect*, *kibitz* and *xkibitz*. One of the classical concurrent problems is explained. Students make teams with one or two members more than the expected number of processes. This allows them to use "guardian" or "scheduler" processes. The assignment is twofold. In the first part of the Lab time (two hours) they have to design a solution for the problem. In the second part, they get some other teams solution and they have to test it. The solution is a script with primitives they have to perform and other behaviors they have to simulate (for example, "eat spaghetti").

Each team chooses the assembly of computers, interconnected by X-Window with a server. They may choose to use only one computer with all the windows or they may choose to have each one a computer with any number of shared private windows at their will. Each member has a window that is where they issue the primitives and act the algorithm. They take notes of the proposed algorithm and finally they hand it to the assistant. Usually, the assistant asks for a performance of the algorithm, but making the students to change the roles they played during development. This is a hidden auto evaluation activity.

This methodology of work helps each member to use his/her own learning style [16] and still make contributions to the overall designing process. Highly kinesthetic oriented members tend to share a computer and change places depending on who is going to "move". More reflective or individual learners prefer to have their own seat (and their own disposition of papers and notes) and have their terminals duplicated in the general one. Active learners use the private windows to test their ideas before exposing them to the group. It also helps to accommodate intuitive or sensing learners, and to change the approach every time they want. As the team approaches a final result we note that students tend to move from using only one computer to a more spread assembly and finally back to the single computer, but this attitude varies a lot.

And for the concurrent part, students are doing the concurrence from the beginning of the process. They learn to think concurrently (if such thing is possible) by seeing true concurrence in action. And it is not a simulated or visualized concurrence, but a real one, where the "processors-processes" are themselves.

In the second part of the assignment, where they are asked to test others team's solutions, they can assemble their computers with the same freedom that in the first part. Sometimes we make them test a solution for the same problem and sometimes for another concurrent problem. We note also a tendency for the centralized-individual-centralized movement.

CONCLUDING REMARKS

It is early to conclude if this kind of work performs better in allowing each student to deal with the design of concurrent applications. Our institution allows for 18 months after the course to sit for the final examination and this period has not yet expired. Students are asked to hand in a complex programming assignment before they take the final test, but they can handle it one month after classes and take the test 17 months after. Also, the unending political and economical struggle in our country does not allow us to compare adequately among long periods [17]. Anyhow, the longer the period from the course to the test, the more difficult it is to evaluate the course methodology's influence.

The overall work students handed in so far have a superior quality. Concurrency is part of the design and not a post fix of an otherwise sequential program. Class time become a lot more fun for both, students and faculty. And lecture time seems to be more meaningful because of the kind of discussions we have.

We believe that a blend of visualization tools to make hidden features apparent with this inspection and action tools for facilitating the design is the most adequate to allow students to create their own framework for incorporating new concepts. But this model or framework has to be explicitly addressed [5] if we want to preclude the building of preconceived notions harmful for the student performance.

REFERENCES

[1] Hoare, C.A.R, Perrot, R.H. (editors), Proceedings of the 1971 Seminar at Queens University: *Operating Systems Techniques.* Academic Press, London, 1972.

[2] Hansen, Per Brinch, *The Architecture of Concurrent Programs*, Prentice Hall, 1978.

[3] Feldman, R. et alt., Smallada, available at *Simtel Software Collection,* http://www.bsdi.com

[4] Feldgen, M; Clúa, O.; Bettini, V. et al., "Contenidos y Métodos para la Enseñanza de la Programación Concurrente", *Proceedings of the ICIE 95,* Facultad de Ingenieria, UBA, Buenos Aires, 1995, p 89.

[5] Ben Ari, Mordechai, "Constructivism in Computer Science Education", *Proceedings of the ACM SIGSCE* 98, Atlanta, GA, USA 1998, p 257.

[6] *Rationale for the Design of the Ada Programming Language,* United States Government, 1986, Chap 13, "Tasking", available at http://sw-eng.falls-church.va.us/AdaIC/

[7] *Ada 95 Rationale:* United States Government, 1994, Chap 9 "Tasking" available at http://sw-eng.falls-church.va.us/AdaIC/

[8] Powers K., Powers, D., "Making Sense of Teaching Methods in Computing Education", *Proceedings of the IEEE Frontiers in Education FIE 99,* San Juan, Puerto Rico, 1999.

[9] Johnson, D. W., Johnson, R.T., *Learning Together and Alone. Cooperative, Competitive and Individualistic Learning,* Allyn and Bacon, 1994.

[10] Baldwin, D., "Discovery Learning in Computer Science", Proceeding of ACM SIGCSE 96, Philadelphia, PA, 1996.

[11] Doran, M., Langan, D., "A cognitive Approach to Introductory Computer Science Courses, *Proceedings of ACM SIGCSE* 95, Nashville, TN, 1995, p.218.

[12] Bloom, B. et al., *The Taxonomy of Educational Objectives McKay,* 1956.

[13] Bedy, M., Carr, S., et al, "A Visualization System for Multithread Programming", *Proceedings of ACM SIGCSE 2000*, Austin, TX, 2000.

[14] Stevens, W., *UNIX Network Programming, Vol. 2, Inter Process Communications,* Prentice Hall, 1999.

[15] Libes, D., "Kibitz - Connecting Multiple Interactive Programs Together", *Software - Practice & Experience,* Wiley & Sons, West Sussex, England, V. 23, V 5, May, 1993.

[16] Hein, T. and Budny, D., "Teaching to Student's learning styles: Approaches that Work", *Proceedings of the IEEE Frontiers in Education FIE 99,* San Juan, Puerto Rico, 1999.

[17] Feldgen, M., Clúa, O., "Social Influence in Student Attitude, Twelve years of computer network teaching in Argentina". *1999 IEEE Frontiers in Education*, San Juan, Puerto Rico, 1999.