SHAPING THE FOCUS OF THE UNDERGRADUATE CURRICULUM

Marcin Paprzycki and Janusz Zalewski
Dept. of Math & Computer Science
The University of Texas of the Permian Basin
Odessa, TX 79762-0001
(915) 552-2258/2260
{paprzycki_m, zalewski_j}@utpb.edu

Abstract. *This paper outlines an approach to reshape the existing undergraduate CS curriculum. Based on software engineering and parallel computing concepts, the details of the new curriculum are presented in terms of educational objectives, core courses, innovations in the teaching method, and early experiences.*

1 Introduction

This paper discusses the approach taken to reshape the existing undergraduate computer science curriculum. There are two primary educational objectives that drive this process:

- Make our graduates more competitive on the job market.
- Increase the chance of admission to the graduate schools for those who want to pursue their studies.

The first objective is to be met by a proper selection of focus, concentrating on certain areas in depth, rather than by giving a general background in many areas of computer science. In addition to that, the areas chosen should be attractive enough, so that current and future job market demands are met by students absorbing sufficient knowledge of respective subjects.

The second objective determines the need to ensure compatibility with existing undergraduate programs from graduate schools. This criterion translates into a necessity of preparing students sufficiently well to meet the admission requirements.

In addition to the two, above mentioned, primary objectives, there are a number of secondary objectives, such as: limit changes in the current CS curriculum to the necessary minimum, ensure compatibility with Computing Curricula '91 [36] in view of possible accreditation, etc. The secondary objectives will not be discussed in this paper.

In the following sections, we present the details of meeting these objectives, including mathematical background (Section 2), selecting core courses and their contents (Section 3), and innovations in the teaching method, such as the use of laboratories, tools, and other concepts to enforce the transmission of knowledge (Section 4). This is followed by a brief description of early experiences, in Section 5, and conclusion in Section 6.

2 Meeting the Objectives

Both our primary objectives are very pragmatic in the sense that they respond to real-life situations. We have to keep in mind, however, that due to their dynamic nature they may change in future, so the curriculum being developed must be flexible enough to easily accommodate prospective changes.

2.1 Competitiveness Objective

This is a very practical objective which is the major driving force of the new curriculum. To meet this goal, we analyzed local (state) [35] and regional [14] demand for computer science majors. Several companies in various Texas areas and neighbor states were called and inquired about their current needs for CS graduates as well as their needs in the immediate future.

The overwhelming majority of responses (near 70%) indicated the demand for graduates familiar with various stages of the software development process. This indication, supported by an observation of local businesses, was an important factor in leading us to the conclusion that every effort should be made to base the curriculum on Software Engineering concepts and introduce them, as early as possible, in the CS curriculum.

To determine the most promising area of computer science, with respect to the potential growth within the next 5-10 years, we studied the list of feature articles in forty most recent issues of IEEE Computer magazine (from January 1992 to April 1995). We found that two fields prevailed in the number of research articles: Software Engineering, and Parallel Computing. Parallel Computing alone is represented by the largest number of single-topic articles, beating the next category, Software Engineering, by some 20%. Along with distributed computing, computer networks, and data communication, parallel computing takes more than 30% of the entire number of articles printed for the studied period.

To relate the above findings to current developments in computer industry, one of the authors spent summers 1994/95 in the Silicon Valley area, and the other one in Europe, trying to collect evidence on current practice and the most immediate future of computer technology. Observations of correct developments at companies such as Sun Microsystems [23], Silicon Graphics [8], as well as presentations at conferences, such as Hot Interconnects [12] and others, convinced us that parallel computing in various forms is indeed an area of the highest potential. All these studies led us to the conclusion that Parallel Computing is another field which needs to be melted with our curriculum.

2.2 Preparedness Objective

Assuming that our curriculum should be based on Software Engineering and Parallel Computing, we have to meet the second primary objective. To see how our students need to be prepared to meet the requirements of the leading graduate schools, we studied several graduate programs with emphasis on software engineering, and analyzed undergraduate programs from top computer science schools (using their catalogs).

There exists a number of CS/CIS programs that require only a minimal mathematics component. Such approach may lead to satisfying our first objective (even though it would be open to discussion), but definitely does not satisfy the second one. We have no doubt that a strong mathematical background is important to our undergraduate CS program. Therefore we stress the necessity of strong mathematical preparation by including, as a requirement, several mathematics courses.

The mathematics support in our institution was quite standard and consisted of requiring all CS majors to take the three-course calculus sequence followed by two courses selected from the four core courses of the mathematics major: statistics and/or probability, linear algebra, intermediate analysis and algebraic structures.

Recently it became clear to us that after finishing the calculus sequence our students (computer science, as well as mathematics majors) are not prepared well enough to take further courses in mathematics or computer science. The primary deficiencies were in the areas of basic mathematical knowledge: logic and set theory, and theorem proving. To alleviate the problem a new course (Introduction to Mathematics, MATH 305) was developed. This course replaced one of the two mathematics support electives. The content of this course is concentrated around basic methods of proof and is supposed to provide the student with fundamental skills and tools in this area.

3 Implementation in Core Courses

To meet the objectives as suggested above, it is not enough to add a course or two (say, on Parallel Computing, as discussed in [19, 22]) to the curriculum already filled to the limits; a radical change in most of the currently existing courses is required. The approach presented here is an extension to the proposed incorporation of parallel computing into the curriculum without a separate parallel computing course [27]. This approach, somewhat similar to the one discussed in [33], employs a top-down, software engineering view of teaching parallel computing and comprises five relatively well separated layers of knowledge:

- understanding parallel applications
- parallel algorithms
- methodologies for parallel software construction
- implementation techniques
- parallel hardware architectures.

Below a brief description of our current core courses is given, in terms of the new contents, including more important modifications due to the change of focus. Software engineering contents and parallel computing contents is discussed for each course.

1. Computer Science I (CPSC 121)
The basic paradigm for this course is the three-phase software development cycle: Design/Code/Test. All fundamental computer science concepts are taught using this vehicle. All classes, including lectures, are taught in the lab and approximately one third of lecture/lab time is spent on each phase. Especially important, from the practical viewpoint, is to teach the right approach to program testing, beginning from the simplest examples, such as a program solving a quadratic equation (which nevertheless contains at least 7 test cases, to test it adequately), to less obvious

examples, such as a program which runs correctly but contains a fault.

The choice of a programming language is very important for the beginners, although teaching a programming language is a by-product of this course. This is based on a firm practical observation that coding is not the most critical phase in the software life cycle. Once the design is done well, coding can be made trivial. Therefore, in the first place, we teach concepts of software development. Our course is based on Pascal, because we believe it is the simplest vehicle to teach such concepts. It avoids the dangers and difficulties of C and Ada, for example, and provides a good background to learn any other structured programming language quickly.

No explicit parallelism is taught at this level; only preparatory concepts are introduced that are well matched with software engineering principles. The list of topics that need to be covered in this respect includes: local/global variables, information hiding (ADTs), separate compilation. Examples of concurrency and task separation are discussed when the concept of an algorithm is introduced (for example, using flowcharts). The idea of a computer with more than one processor is discussed.

2. Computer Science II (CPSC 122)
This course is based on two fundamental paradigms: use of software development tools other than compilers and debuggers, and explicit programming of parallelism/concurrency. Our first attempt is to use a tool for structured program development, based on Nassi-Shneiderman diagrams, XperCASE (reviewed in [18]).

To introduce parallelism/concurrency, a more gradual approach is needed. The typical CS I course covers the programming concepts approximately up to the introduction of the first structured data type – single-dimensional arrays, although it is often possible also to introduce the concept of a matrix. Our experiences show that the new approach does not slow down the pace of the instruction, so it should be possible to introduce the students to the early glimpses of parallelism, near the end of the semester in CS I. If not, CS II starts with basic vector and matrix operations (vector addition, dot product, matrix-vector multiplication, matrix-matrix multiplication) to introduce the preliminary concepts of parallelism, and possibly the notion of task/problem granularity (see [27] for examples of the appropriate laboratories).

Explicit parallelization is discussed using programming examples of introducing concurrency to elementary searching and sorting algorithms [25]. These programs, together with the earlier examples of matrix operations are used to introduce concepts of load balancing and algorithms' performance characteristics.

3. Programming Language Survey
Our current requirements include a second programming language: C, Ada, Fortran, or Cobol. We feel that this should be changed and the course should be extended to teach broad spectrum of implementation techniques. This can be either a course on Programming Languages or an Operating Systems course, provided either one can teach respective concepts of concurrency, its models (shared memory and message passing), and basic interprocess communication primitives, such as semaphores, monitors, etc.

Our approach to teaching implementation techniques may be controversial but is based on an extensive industrial experience: it does not matter very much, on this level, which language we use – what is important are concepts. One part of this attitude is mixing languages in a single project. In practice, there are very few meaningful applications (if any) written exclusively in one language. Therefore programming in a variety of languages should be an essential component of computer science education.

4. Discrete Mathematics (CPSC 312)
We consider this course to be the one giving theoretical background for formal software development as well as for the design and analysis of parallel algorithms. The necessary topics include: propositional calculus, predicate calculus, formal logic, sets, relations, functions and sequences, combinatorics, graph theory, introduction to Petri nets, introduction to cellular automata.

To cover the entire material of this sort, it is not enough to have one course, therefore we introduced a prerequisite course, Introduction to Mathematics (MATH 305), which focuses on logic and theorem proving. It is worth noting that in recent years several textbooks were published that quite well match our course contents requirements, for example [11, 13]. The one currently used in MATH 305 is [16].

5. Digital Computer Organization (CPSC 310)
Traditionally this course is oriented on introduction to computer architecture, which has to teach all basic hardware-related concepts, so not much time is left to introduce basic ideas of parallel architectures, especially without hardware lab. If this is possible, however, we recommend to cover any of the following topics: instruction-level parallelism, cache, bus arbitration, coherence problems and their hardware solutions, hardware lock, parallel I/O. The book by Dowd [5] is a very good example of what approach could be taken here.

Since this is the only hardware-oriented course in our core, to make it more effective we plan to base it on the extensive use of tools. This includes courseware covering ordinary topics [4], as well as the introduction of CAD software, such as [3], in a way the SPICE simulator dominates courses on analog circuitry [30]. If elements of parallelism are present, respective courseware and CAD tools should also be used, as much as possible, for example [7].

6. Data Structures (CPSC 320)
This course has been a good vehicle to introduce or strengthen, if introduced earlier, the concepts of parallel algorithms (as an extension of sequential ones), their design, analysis and implementation. All the examples of programming projects include the parallel component. The emphasis in contents is on optimization algorithms, which comprise the most widely used class of algorithms in practice.

The software engineering approach taken in this course needs to employ object-oriented concepts (based on earlier introduced notions of abstract data types), to teach how to produce reusable software components. Formal methods should be also applied in this course to teach systematic and rigorous development of such components, and to prove the correctness of algorithms.

Another important paradigm in this course should be the introduction to performance analysis of parallel algorithms, including formal definitions of speedup, etc. As far as possible, various courseware items already developed should be used, such as CD-ROMs [9, 10] or DS Guide [29].

7. Software Engineering
This course, actually called Information Systems Design (CPSC 315), is designed to teach the full-fledged software development cycle and an application of software development methodologies. Since the prerequisite in data structures is assumed, it is not a course for beginners. Students must have been already introduced to the basic concepts of software engineering.

The additional emphasis in teaching is on team work [39] and the use of tools [40]. Team project is an essential part of this course, to let the students interact in groups to meet a common objective. The extreme solution is to employ a parallel model of software development, where teams work in parallel, from the beginning, on various phases of the simplified development cycle (say, requirements, design, coding, testing, and verification), and change roles if the circumstances permit. This is usually a project in concurrent software development for some real-life situation, such as a real-time application [38, 41] or a business application. The use of an integrated toolset for software development is critical to the success of this course.

8. Senior Research Project (CPSC 495)
The objective of this course is to develop a meaningful application, including parallelism or concurrency, to make evident the student's ability to perform individual work under supervision and demonstrate his/her maturity. The project includes elements of parallel computing as well as software engineering.

The way we are approaching this course and select meaningful applications is by talking to researchers and educators from other science disciplines (biology, chemistry, geology, math, physics, and also business and economics) to learn about their needs in parallel computations. This prepares students better to understand computational problems of these disciplines and respond to their future needs, which leads to blending such needs with methods of solution that computer science can offer. The idea of integrating the needs with CS tools in the undergraduate curriculum was explicitly suggested in [20] and is actually being implemented elsewhere [31].

4 The Method

At this point, we feel we have a well designed curriculum. The dynamic process of teaching is responsible for its ultimate success, however. This requires the introduction of several innovations in the method, some of which are discussed below. The natural categorization divides them into new teaching concepts and new teaching aids.

4.1 Teaching Concepts

There are four basic ideas we are trying to base our method upon: scientific experimentation, expanding from examples, recurring concepts, and technical writing.

The way the CS laboratories are usually designed and conducted does not satisfy our requirements as far as the scientific experimentation component is concerned. The principle of scientific method, which leads to a discovery, is to introduce changes in the environment and make observations that must be presented in a form of numerical results. Then the results are analyzed and conclusions are drawn that may lead to a discovery. Therefore the basis of our labs is experimenting in computer science, that is based on performance evaluation. For example, to measure performance of parallel algorithms there is a number of important parameters influencing the algorithm's behavior when executed on a computer with multiple processors. In the simples,t case it could be the size of the problem n and the number of processors P. Then a thorough investigation of the two-dimensional performance space will be expected from the students, e.g., for a certain range of values of n, study the program behavior when executed on $1, 2, \ldots, P$ processors.

What we consider crucial to the success of this approach is a set of carefully selected examples: demos, exercises, assignments, and team projects. Our former experience proved that this way of gradual introduction of new concepts significantly enforces their understanding [26, 27]. The way it works is that students are presented with a demo program showing how a certain concept works. Then they are given a task of modifying slightly the existing program, under instructor's supervision, to achieve a predefined effect. Based on this knowledge, a more involved homework or open-lab assignment is given to them. If solved, it is followed by a more sophisticated problem requiring a solution as a team project.

Another vehicle that worked well in our experience is that of recurring concepts from [36]. For instance, in parallel computing, one can use the notion of multiple entities competing for a single resource, on virtually all levels of software development hierarchy, in a variety of forms, such as: job partitioning (algorithm level), resource allocation (design level), task scheduling (implementation level), bus arbitration (architecture level). Several other notions can be used that way, across the entire curriculum, for example, performance evaluation (of algorithms, development methods, implementations, and architectures), heterogeneity (mixing programming environments, mixing languages, and mixing architectures).

The way our assignments and laboratories are structured allows us to build upon them. It is a fact in our environment that a typical undergraduate student has a limited writing ability. Typical English courses do not help much as far as technical writing is concerned (and only very few schools have a Technical Writing course requirement). In our curriculum we introduce students to the basic ideas of technical writing as early as in CS 1. We do so by appropriately structuring the laboratory exercises and homework assignments [27, 25], and requesting students to write technical reports based on their findings. Later in the curriculum, we require students to write a substantial number of literature-search type research papers (for more details see [28]). This approach is successful to the extent that allows students to write and publish a software review paper [18].

4.2 Teaching Aids

By teaching aids we understand all environmental factors that can be used to impact the effectiveness of teaching. In this paper, we consider three kinds of teaching aids: computer laboratory equipment, software tools, and audiovisual aids.

The fundamental question in improving the effectiveness of teaching any computer science course is the existence and usefulness of a computer laboratory. In addition to traditional concepts of a computer lab, based on a network of personal computers and/or workstations, we feel that in our environment, where parallelism is a fundamental concept taught, a parallel computing laboratory is essential. With very limited availability of funds, we were forced to seek a cost-effective solution, which we found to be a transputer-based equipment. In choosing this type of lab, we also considered the extent to which we would be able to use experiences of other educators in pursuing the idea of a Closed Lab [21].

Another crucial environmental factor helping increase teaching effectiveness is the availability and classroom use of software tools, especially important in a curriculum based on software engineering principles. One of the authors developed such an integrated environment from off-the-shelf components, and used it successfully in classrooms, in a course focusing on real-time systems [40]. The components include high-level specification and design tools based on object-oriented techniques [32], compilers of concurrent languages, real-time kernels, and appropriate simulators. Portability of designs, simulation of designs and automatic code generation, portability of the source code across different platforms, mapping language constructs onto operating system kernel calls, all are important issues stressed in such a course.

In addition, we are trying to use various forms of audiovisual aids in a systematic manner across the curriculum, to enhance our courses. This includes traditional videotapes [1, 15, 34] and courseware [3, 4, 6, 7, 29] designed specifically for certain types of courses but also new media, such as CD-ROMs, multimedia and networks (accessible via World Wide Web) [9, 10, 37]. Our first lessons learned from the extensive use of non-printed media make us believe that a classical textbook may become obsolete, maybe in the near future.

5 Early Experience

Redesigning a curriculum is a time consuming process and usually takes several years. At this point in time (end of 1995), changes in lower-level courses from the core have been implemented. Below, we shortly describe one such change, a semester long software development project in CS II, which has been designed to meet the competitiveness objective and respond directly to a challenge from industry (more detailed discussion can be found in [2, 24]). In particular, this project addresses the needs articulated most recently in [17]: "industry needs software practitioners who understand the dynamics of developing and reengineering large software systems, adeptly employing techniques such as analysis and reuse."

5.1 Project Description

The CS II course is currently relatively small and consists of one section of approximately 15 students. The semester long software development project was introduced two years ago as an attempt to confront the future computer scientists with a larger software artifact.

The starting idea for the project is Conway's Game of Life. This relatively simple simulation game is extended to a larger artificial life simulation project. This project is based on a group work, where the groups consist of 2-4 students – depending on the class arrangements and attrition.

The project proceeds in phases. Phase I, development of the original game of life, consists of the following steps:

1. Formulation of the software requirements
2. Writing the design
3. Instructor's comments on the design
4. Implementation
5. Testing and preparation of the User Manual
6. Across-the-class comparisons
7. Instructor's comments on the artifacts

At the beginning of the process, groups are created and students study the original paper by Conway. Then they are asked to discuss the basic system requirements with a very strong emphasis on the development of human-computer interface. This step is followed by each group writing the design (in the form that they were introduced to in the CS I course). These designs are then commented on by the instructor.

After receiving feedback on the designs, students work on improving them and implementing the code, testing the code and writing the user manual. At this moment, groups exchange the artifacts: design, user manual, and code. Depending on the size of the class, each group receives the artifacts prepared by all other groups or some subset (up to two or three) of other groups. Each of the groups has the task of: commenting on the design, commenting on the user manual, verifying the code against the design, running the code and testing it, commenting on the user interface – all tasks comprising a natural way of introducing independent verification and validation. Students are instructed to concentrate their attention on particular verification criteria, such as consistency and clarity. They are also informed that they should read the user manual from the point of view of the prospective users of the system, so that they should concentrate their attention on all things that are unclear and not explained well-enough. The improved documents are then commented on by the instructor.

Phase II, first modification of the system, comprises the following steps:

1. Specification of new requirements
2. Upgrade of the design document
3. Instructor's comments on the design
4. Modification of the code to include changes
5. Testing the new code and upgrading the manual
6. Across-the-class comparisons
7. Instructor's comments on the artifacts

The second phase is almost a repetition of the previous one, although this time students work not on the new code, but introduce modifications to their own, earlier developed, programming artifacts. This phase is then repeated as many times as there is time during the semester.

5.2 Addressing Software Engineering Needs

The project described above matches the software engineering education needs as specified in [17] well.

1. Software reuse
After the first phase, students stop programming from scratch, as they introduce changes to the existing programming artifacts: the design, code, and user manual. This is an even more realistic model of what happens in real life than simple changes in some of the programming artifacts proposed in [17]. Here the developers are faced with a situation where the entire set of software related documents needs to be reengineered. It is also possible, although we have not done it so far, to reuse software artifacts as a starting point during the next semester, so that the software reuse and analysis of existing artifacts will be utilized to its fullest. Going in this direction even further, it is also possible that students from one group introduce changes to the products generated by another group.

2. Group work and communication skills
Obviously our project proceeds as a group project and as such it matches the natural situation in industry. Students from one group take active part not only in developing their own software, but also act as analysts, verifiers, and users of the products of other groups. This allows them, first, to see the issues of software quality from all sides and, second, to evaluate their own product in a view of what the other groups were proposing.

Group work leads naturally to the expansion of communication skills. In addition, since the groups actively deal with the products prepared by other groups they are involved in a group-to-group communication.

3. Increasing project size

Due to the introduction of this project, students have a chance, already in the CS II course, to be confronted with the management of a project which is between 2000 and 3000 lines long. This experience is very unique when, during the final stages of the project, they are introducing changes/additions to the 2500 lines of existing code. Only then they really begin to appreciate the importance of issues like quality of the documentation, code modularity, the appropriateness and number of comments in the code.

5.3 Further Observations

During the two first semesters of using the project in the CS II course, a number of observations have been made. In both semesters a different number of software cycles have been reached. Also, since the students themselves decide which way to proceed with the project, different types of upgrades have been proposed. Typical changes include: introduction of random movement, introduction of two genders, introduction of food on the field which is then eaten at a certain rate by the creatures, introduction of the aging process. In each group, a different resolution of interactions between various parameters have been observed. In all cases, a relatively large working code with all the supporting documents have been produced.

The primary problems are related to group work. It was observed that as long as the group works hard and as a unit, even if the group consists of relatively weak students, they were able to reach satisfactory results. However, any kind of group disfunctionality leads to serious problems with project completion and quality. Typical problems are related to some of the group members not doing the required work on time, some of the group members not being ready to cooperate with others (for various reasons), groups falling apart due to attrition.

We do believe that a project of this type should be introduced at CS II level, even if this means that some of the material typically covered in such a course will not be covered. The fact that students actively develop and maintain a software system (relatively large in comparison to their former experience) is extremely important from the point of view of matching the needs of future employers. Also, the fact of introducing students relatively early to group work will definitely be beneficial to them.

6 Conclusion

In this paper we show how to remodel the undergraduate CS curriculum to meet a set of well-defined objectives: competitiveness on the job market and preparedness to enter graduate schools.

It is extremely important that the CS graduates are well prepared both to join the work force as well as to pursue more advanced levels of education. Very often these two goals were perceived as contradictory, since the requirements of future employers were considered incompatible with the requirements of graduate schools.

We are proposing and implementing a new approach to shaping the undergraduate CS curriculum focused on two areas: software engineering and parallel computing. This approach allows us to produce graduates that are well prepared to face the current and future computational challenges, whether they decide to pursue their careers by going for employment or by staying at school.

For this approach to work well, substantial changes are needed in the core courses, math prerequisites, and in the teaching methods. We presented the first experiences and progress made, so far, in all these areas.

7 Acknowledgement

Work supported in part by a grant from ARPA, via USAF Phillips Laboratory, Solicitation No. F29601-94-K-0046. A preliminary version of this paper was presented at the Forum on Parallel Computing Curricula, Wellesley, Mass., March 31 - April 1, 1995.

References

[1] A *Z* Readers Course (5 videotapes). Pavic Publications, Sheffield, UK, 1992

[2] Bailey D., A. Cheek, M. Paprzycki, Developing an Artificial Life Simulation Package, Proc. 12th Annual Conference on Applied Mathematics, Edmond, Oklahoma, February 1996 (to appear)

[3] Capilano Computing Systems, LogicWorks – Interactive Circuit Design Software. Benjamin/Cummings, Redwood City (CA), 1994

[4] Digital Techniques. Computer-Aided Instruction. Heathkit Educational Systems, Benton Harbor (MI), 1994

[5] Dowd K., High Performance Computing, O'Reilly and Associates, Sebastopol (CA), 1993

[6] Eaton J.K., L. Eaton, LabTutor. Stanford University, Stanford (CA), 1992

[7] Futurebus+ Concepts. Self-paced Instruction Tape. Digital Equipment Corp., Burlington (MA), 1993

[8] Galles M., The Challenge Interconnect, Hot Interconnects: A Symposium on High-Performance Interconnects. Stanford University, Palo Alto (CA), August 5-7, 1993, IEEE Computer Society TC on Microprocessors and Microcomputers, Washington (DC), 1993

[9] Gloor P. et al. (Eds.), Animated Algorithms: A Hypermedia Learning Environment for "Introduction to Algorithms" (CD-ROM). MIT Press, Cambridge (MA), 1994

[10] Gloor P. et al. (Eds.), Parallel Computation – Practical Implementation of Algorithms and Machines (CD-ROM). Telos/Springer-Verlag, Santa Clara (CA)/New York, 1994

[11] Gries D., F.B. Schneider, A Logical Approach to Discrete Math. Springer-Verlag, New York, 1993

[12] Hot Interconnects III: A Symposium on High-Performance Interconnects. Stanford University, Palo Alto (CA), August 10-12, 1995, IEEE Computer Society TC on Microprocessors and Microcomputers, Washington (DC), 1995

[13] Ince D.C., An Introduction to Discrete Mathematics, Formal System Specification, and Z. Second Edition. Clarendon Press, Oxford, 1992

[14] Job Choice 1995 in Science and Engineering. A Guide to Employment Opportunities for College Graduates, College Placement Council, Bethlehem (PA), 1994

[15] Kennedy K. et al., Parallel Computation: Practice, Perspectives and Potential. CRPC Short Course (7 videotapes). Center for Research in Parallel Computation, Rice University, Houston (TX), 1994

[16] Kurtz D.C., Foundations of Abstract Mathematics. McGraw-Hill, New York, 1992

[17] Lawlis P.K., K.A. Adams, Computing Curricula vs. Industry Needs: A Mismatch, pp. 5-19, Proc. 9th Annual Ada in Software Engineering Education and Training (ASEET) Symposium, Morgantown (WV), 1995

[18] McBee L. et al., XperCASE(SPX) Lite. Computer, Vol. 27, No. 7, pp. 118-119, July 1994

[19] Metaxas P.T., Fundamental Ideas for a Parallel Computing Course, ACM Computing Surveys, Vol. 27, No. 2, pp. 284-286, June 1995

[20] Misra M., An Undergraduate Course in Parallel Computing for Scientists and Engineers. Proc. NSF Workshop on Parallel Computing for Undergraduates, pp. 22/1-13, Colgate University, Hamilton (NY), June 22-24, 1994, C. Nevison (Ed.)

[21] Nevison C.H. et al. (Eds.), Laboratories for Parallel Computing. Jones and Bartlett Publishers, Boston (MA), 1994

[22] Nevison C.H., Parallel Computing in the Undergraduate Curriculum, IEEE Computer, Vol. 28, No. 12, pp. 51-56, December 1995

[23] Normoyle K. et al., Systems Architecture for the 90s: Introducing UPA From SPARC Technology Business, Hot Interconnects III: A Symposium on High-Performance Interconnects. Stanford University, Palo Alto (CA), August 10-12, 1995, IEEE Computer Society TC on Microprocessors and Microcomputers, Washington (DC), 1995

[24] Paprzycki M., CS II: An Applied Software Engineering Project, J. of Computing in Small Colleges, Vol. 11, 1996 (to appear)

[25] Paprzycki M., R. Wasniowski, J. Zalewski, Parallel and Distributed Computing Education: A Software Engineering Approach, Proc. 8th SEI Conf. on Software Engineering Education, pp. 187-204, New Orleans (LA), March 29 – April 1, 1995, R.L. Ibrahim (Ed.), Springer-Verlag, Berlin, 1995

[26] Paprzycki M., J. Zalewski, Introduction to Parallel Computing Education. J. of Computing in Small Colleges, Vol. 9, No. 5, pp. 85-92, May 1994

[27] Paprzycki M., J. Zalewski. Teaching Parallel Computing without a Separate Course, Proc. NSF Workshop on Parallel Computing for Undergrads., pp. 19/1-18, Colgate University, Hamilton (NY), June 22-24, 1994, C. Nevison (Ed.)

[28] Paprzycki M., J. Zalewski. Should Computer Science Majors Know How to Write and Speak?, J. of Computing in Small Colleges, Vol. 10, No. 5, pp. 142-151, May 1995

*****Undergraduate Continued On Page 50*****

******************************Undergraduate From Page 44********************************

[29] Pinter-Lucke J., DS Guide (Courseware). Intellimation, Santa Barbara (CA), 1993

[30] PSPICE Electrical Circuit Simulator. Student Version 5.0. Prentice Hall, Englewood Cliffs (NJ), 1992

[31] Rebbi C. et al., Lecture Notes of the Workshop on Parallel Computing for Undergraduates. Boston University, Center for Computational Science, May 25-27, 1994

[32] Robinson P.J., Hierarchical Object-Oriented Design. Prentice Hall, Englewood Cliffs (NJ), 1992

[33] Rosenberg A.L., Thoughts on Parallelism and Concurrency in Computing Curricula, ACM Computing Surveys, Vol. 27, No. 2, pp. 280-283, June 1995

[34] Shatz S., Concurrent Software Analysis. A Videotape. IEEE Computer Society Press, Los Alamitos (CA), 1991

[35] Texas' Hottest 500 Employers, Future Outlook, Vol. 2, No. 2, pp. 23-34, 1994

[36] Tucker A.B. (Ed.), Computing Curricula '91. Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM/IEEE, New York, 1991

[37] Umar V.M. (Ed.), Computational Science Education Project. Universal Record Locator (URL): http://csep1.phy.ornl.gov/csep.html

[38] Wann K-C., J. Zalewski, Scheduling Messages in Real Time with Application to the SSC Message Broadcast System, IEEE Trans. on Nuclear Science, Vol. 41, No. 1, pp. 213-215, February 1994

[39] Zalewski J., IEEE Draft P-1074 Mapped on a Parallel Model: A Teaching Vehicle for Software Development. Proc. Workshop on Directions in Software Engineering Education, pp. 125-134, L.

Undergraduate Continued On Page 64

*******************************Undergraduate From Page 50*********************************

Werth and J. Werth (Eds.), 13th Intern. Conference on Software Engineering, Austin (TX), May 12-16, 1991

[40] Zalewski J., Cohesive Use of Commercial Tools in a Classroom. Proc. 7th SEI Conf. on Software Engineering Education, pp. 65-75, San Antonio (TX), January 5-7, 1994, J.L. Diaz-Herrera (Ed.), Springer-Verlag, Berlin, 1994

[41] Zalewski J., Boiler Water Content Controller Based on EWICS Safety Model. Proc. Intern. Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems, Ottawa, Canada, June 28-29, 1993. University of Waterloo, Institute of Risk Research, 1994

[42] Zalewski J. (Ed.), Advanced Multimicroprocessor Bus Architectures. IEEE Computer Society Press, Los Alamitos (CA), 1995